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 background 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. Finally, we contend the novelty of the project in the context of existing work in the domain as well as provide an overview of the project milestones and approach.

Keywords gradual typing, Octave

1 Introduction

Static and dynamic type systems for languages have their own distinct advantages. For instance, a static type system enables early error detection and enforces a certain extent of code style within a collaborative setting. On the other hand, the lightweight workflow associated with dynamic typing is highly suited for rapid prototyping and iterative approaches. Over the past several decades, researchers in the programming language community have been working on integrating aspects of both static typing and dynamic typing with the goal of allowing programmers access to advantages of both type systems. Gradual typing—originally proposed by **?**]—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 [**?**], TypeScript [**?**] and Reticulated Python [**?**].

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 [**?**]. 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 [**?**], Olmos et al. clearly illustrate the benefits of introducing static type checking and shape analysis into Octave. In their work, static type and shape inferencing is used to directly reduce the number of type and shape checks needed at runtime and in doing so they can improve the efficiency of existing code. Functions and operators are highly overloaded in Octave to provide a simple interface for the end-users; however, the inevitable tradeoff for this flexibility is computational performance at runtime. The runtime system is responsible for type checking, array shape determination, function call dispatching, and handling possible runtime errors. Hence Octave suffers from a number of computational inefficiencies that may be improved provided additional static type and shape context during the compilation step.

2.3 Octave Has Reference Parsers

It is also convenient that Octave itself is open-source and serves as a great point of reference. Specifically, the parser

for Octave is available online, while that of MATLAB is proprietary and inaccessible. That being said, there are indeed many open source projects related to MATLAB and Octave parsing, which will help us parse Octave source code. For instance, the matlab-parser is an ANTLRv4 parser for MATLAB programming language, which converts the MATLAB source code to a concrete syntax tree representation, which is processed to an abstract syntax tree and control flow graph [**?**]. Another MATLAB parser we can potentially reference, the mparser, is based on an ANTLR v3 grammar is also able to do source code translations to an abstract syntax tree [**?**].

2.4 Octave Type Domain Is Interesting

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

3 Advantages of Gradual Typing

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

3.1 Gradual Typing Incorporates Advantages from Static and Dynamic Typing

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

In a gradual type system, programmers are able to decide which regions of code in a computer program are statically or dynamically typed, making different pieces suitable for different stages and scenarios [**?**]. 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 [**?**]. Hence, computer programs written in a gradually-typed languages take advantage of static typing in some regions while enjoying the flexibility of dynamic typing in other regions, integrating the benefits from both type paradigms in a single program simultaneously.

3.2 Gradual Typing Can Be Implemented Efficiently

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

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

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

3.3 Gradual Typing Enables Evolution of Programs

Due to flexibility of dynamically-typed programming languages, it is common in industry that programs are initially developed in a dynamically-typed programming language for reasons like fast prototyping, idea validation and scripting. However, dynamically-typed programming languages usually expose programs to safety and security issues, which sometimes lead to fatal errors and even monetary losses in production environments. Furthermore, the runtime performance of dynamically-typed programs is unable to compete with that of statically-typed languages as static type systems are able to check many aspects of computer programs well before they execute. In order to resolve the issues mentioned above, prototyping programs in industry are sometimes refactored to statically-typed languages, in order to reduce the possibility of errors as well as to enhance runtime performance.

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

Therefore with the aid of gradual typing, programmers are able to gradually evolve their programs from untyped to fully-typed with type annotations without ever breaking the programs.

4 Related Projects

4.1 Octave and MATLAB

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

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

4.1.1 Optimizing Function Overloading in Octave

Octave and MATLAB were designed to be easy to use and have syntax that follow closely with mathematical notation. However, with dynamic type checking, there are issues such as efficiency and the quality of the generated code. As briefly alluded to before, **?**] specifically address overloading in dynamically typed Octave programs. It introduces static typing for overloaded functions by making it explicit and restricting the input and output types of functions to match the actual function call. The main disadvantage to this approach is that the resulting Octave syntax is no longer as close to mathematical notation as before. However, by inferring the types used in the program for static typing, the system does not require any additional annotations from the user; hence the user developing their program is not greatly impacted. Since transformed program is statically typed, the overall computational performance of the system is improved. Thus, this paper shows how mixing dynamic and static typing can improve the overall system for Octave.

4.1.2 Telescoping Compiler for MATLAB

Another aspect in which the notion of static typing has been explored for MATLAB is through the method of telescoping. **?**] describe telescoping as "a strategy to automatically generate highly-optimized domain-specific libraries" and in their work suggest a method to create a telescoping system that generates high performance libraries from prototype MATLAB. In order to accomplish this, it needs to infer MATLAB types. The paper develops an approach of combining static and dynamic type inference roughly as follows: to start the inference, constraints are formed using a database of annotations containing one entry per procedure or operations. Unfortunately, the problem of inferring the types is NP-hard. In order to obtain a relatively more efficient algorithm, a few simplifying assumptions were made to reduce the same problem into an n-cliques problem which is NP-complete. One of the issues with this paper is that unanalyzed procedures will be ignored and those calls will not give any added information. This is a pitfall because all recursive calls are treated as unanalyzed procedures by this method. A key observation that we note from this paper is that typed MATLAB certainly has several benefits but the act of performing type inference can be an issue.

4.2 Gradual Typing Projects

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

4.2.1 Diamondback Ruby (DRuby)

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

Although both DRuby and our proposed project 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 [**?**]. 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*. **?**] introduce a new approach to apply type inference on a gradually-typed language, admitting parametric polymorphism. In addition, the annotation syntax for DRuby is similar to informal documentation of Ruby, which facilitates programmers to annotate their Ruby programs. We will use this syntax as a reference when designing our annotation syntax for Octave.

4.2.2 Typed Racket

Typed Racket [**?**] is the gradual counterpart of Racket, enabling incremental addition of annotations for static type checking. It uses contracts to type-check at the boundaries of statically-typed regions and dynamically-typed regions

at runtime. It also includes type inference, so certain type annotations may be omitted by the programmers.

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

Typed Racket has recently begun to support refinement and dependent function types as experimental features [**?**]. Dependent types were first introduced by Xi and Pfenning [**?**] to eliminate array bound checks with a type-based approach. If time permits, we intend to additionally investigate the use of dependent and refinement types in our proposed type system to Octave to allow for array bound and matrix dimension checks.

5 Project Distinction and Novel Aspects

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

By introducing optional syntactic forms for type specification, the goal is for users to gradually transition away from pure dynamically-typed codebases while also maintaining the flexibility to leave certain pieces untouched. While **?**] similarly allows for optional annotation, it accomplishes this through an aspect-oriented design which ultimately still executes dynamically and fails to address the matters raised in the motivation for our project.

Additionally, previous projects such as **?**] and **?**] 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 [**?**]. 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 typecheck our proposed gradually-typed Octave source files. To demonstrate the entire process of conversion from source code to a check result, we will refer to the test file [examples/-tutorial.rkt](link to file) and show various stages of output as it is processed through our application.

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 three main stages of parsing include 1) tokenization 2) parsing to a Racket-friendly entity 3) parsing the entity to an intermediate representation 4) desugaring into our internal 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 [BNF](link to our language file) 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 [**?**] to handle this stage for completeness.

Figure 1 includes examples of rules used in our BNF.

We begin this stage by performing both lexical analysis and parsing using tools from the [brag](http://docs.racket-lang.org/brag/) library. 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 2.

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](https://docs.racket-lang.org/guide/stx-obj.html)

```
octave
        : translation_unit
        | octave translation_unit
        ;

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

typed_identifier
        : IDENTIFIER
        | IDENTIFIER ':' IDENTIFIER
        ;

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

Figure 1. 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 2. Sample token list

which provides an easily traversable form for us to desugar. The syntax in Figure 3 form is an abstract representation of the input source file in terms of the defined BNF.

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] (https://docs.racket-lang.org/plai/plai-scheme.html) language 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.

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

Figure 3. Sample syntax structure

```
;; Intermediate Typed Octave Construct
(define-type I-TOC
  [i-null]
  [i-id-type (name symbol?) (type Type?)]
  [i-assn-decl (vars (listof i-id-type?))
               (expr (or/c Expr? I-TOC?))]
  [i-app (fun (or/c Expr? I-TOC?))
         (args (listof (or/c Expr? I-TOC?)))]
  [i-int-binop (op procedure?)
               (lhs (or/c Expr? I-TOC?))
               (rhs (or/c Expr? I-TOC?))]
  [i-bool-binop (op procedure?)
                (lhs (or/c Expr? I-TOC?))
                (rhs (or/c Expr? I-TOC?))]
  [i-func (name i-id-type?)
          (args (listof i-id-type?))
          (rets (listof i-id-type?))
          (body (listof (or/c Stmt? I-TOC?)))]
  [i-if-stmt (cond (or/c Expr? I-TOC?))
             (then (listof (or/c Stmt? I-TOC?)))
             (else (listof (or/c Stmt? I-TOC?)))])
```

Figure 4. abstract syntax representation of source code

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.

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

```
(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

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.

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

Figure 6. Abstract syntax for source code

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. 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 [Siek and Taha 2006]. For instance, int -> ? represents a function type whose domain is int and whose codomain 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 dynamic system should allow f to be applied on x, and cast x from type ? to type int in the runtime, whereas type ? is not equal to type int. Hence, an important modification of a gradual type system is to replace the type equality relation with the type consistency relation [Siek and Taha 2006], which is defined as follows. Here, the arrow type (-> T T) is an explicit function type, and the star type T * T is a pair type.

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

Figure 7. Type definitions

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

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

Figure 8. Type consistency rules

The type consistency relation needs to respect several properties. First, type consistency should *conservatively* extend the type equality relation [Garcia et al.'s AGT paper]. Described as a criterion for gradual typing, a gradual type system should behave exactly the same for fully statically-typed programs [Siek et al. 2015]. Hence, two basic types T1 and T2 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 ? and ? bool would imply that int bool, which would violate the criterion for gradual typing stated above. In our type-checker, we implemented the following helper function to check type consistency of two types.

```
(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 9. Helper function to check type consistency

6.3 Typing rules

Figure * 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) [Siek and Taha 2006]. The typing rules described in Figure * can be read âĂIJif the propositions above the line are true, then the proposition below the line will be trueâĂİ, and the type judgements âĂIJ |-e:TâĂİ can be read âĂIJe has type T in the context âĂİ. The metafunction fun extracts function information from a type (i.e., the ? type can be viewed as a function type ? -> ?).

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 1. 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 type bool.

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

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

$$\cdots$$

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

Figure 10. 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 11. Statement typing rules

6.3.1 AST

Statements will be defined as all of the statements we want to support in type checking such as expressions, declarations, assignments, functions, and if statements. The AST will look similar to Figure 12.

We will dig deeper into a few as either an id, constants, application, binop functions, or compop functions. Since this

```
(define-type Stmt
  (U Expr decl assn func if-stmt))
```

Figure 12. AST node for statements

AST will be used for type checking, it can be useful to break up the binop functions to specific functions relating to the type they handle. For example, we separate binop into int-binop and bool-binop where the name tells us which type it supports. We also break up comparisons into int-compop and string-compop to be more specific about the types. This is what the AST for Expr will lookof the statement nodes in this tutorial to understand how to represent them. We will start with the Expr node as it is one where many nodes in the AST will link back to.

We will define an expression like in Figure 13.

```
(define-type expr
  (u id constant app int-binop bool-binop
     int-compop string-compop))
```

Figure 13. AST nodes for expressions

Let's take a look at how int-compop is represented in Figure 14 to have an idea of how the other nodes of Expr will be represented.

```
(struct int-compop
  ([op  : (-> Integer Integer Boolean)]
   [lhs : Expr]
   [rhs : Expr]))
```

Figure 14. AST node for `int-compop`

For int-compop, we have a function that takes in two expressions that needs to be evaluated into integers and returns a boolean that represents whether or not the two integers are equal. Since we know what types this compop function needs to handle, we can introduce it as (-> Integer Integer Boolean) where the last type in the -> notation is the output and the rest are inputs. Given this function, we now need to be able to represent the two integers passed in. They can be represented as expressions that have to evaluate to an integer in order to be consistent. This is beneficial because once we have a type checker for Expr, we can recursively use that type checker on all the other AST nodes that use Expr.

Another important AST node under the statement node is the function node. In Octave, we need to pass in the input and output parameters to a function. With type checking, that means both the input and output parameters can be type annotated. To represent that, we have a (Pair Symbol Type) where Symbol is the parameter name and Type is the type of

the parameter. Since functions can take multiple parameters, we will represent the arguments as a (Listof (Pair Symbol Type)). The body of the function may also need to be type checked. We can achieve that by creating a (Listof Stmt) node to represent the body. This way, we can recursively call the type checker for statement to verify the body of a function.

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

Figure 15. AST node for `func`

Lastly, we need to define what a Type node is. This node specifies the types we will handle type checking for. This includes integers, boolean, string, dynamic, none, arrow, and star which are the types we explained earlier in the type consistency section.

Figure 16 includes an example of how the AST node for Type can be represented.

```
(define-type Type
  (U 'int 'bool 'string 'dynamic 'none arrow
     (Listof Type)))
```

Figure 16. AST nodes for types

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.

6.3.2 Type checker tutorial

When we declare a variable or declare a function, we allow the user to specify what the type of the variables are. However, we need to be able to retrieve this information when we use assignment and application.In order to keep track of what the types of the variables are, we will 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 Symbol Type) to the environment. Symbol in the Pair represents the name of the variable and Type represents the type of the variable. Now, we can thread the environment through our program so it can be accessed by application and assignment. When we get to type checking an application or an assignment, 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.

As we explained earlier, our main AST nodes are statements and expressions. Our typechecker will be broken down into a typechecker for statements and a typechecker for expressions.

Note that the type of an expression will be used to typecheck 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 (or both matrices, for the final project). Therefore, the type-checker for expressions typechecks each sub-expression, performs additional checks and finally returns the type information of the expression.

In contrast, non-expression statements cannot be further operated on and therefore do not have types (denoted by the none type in the AST). However, it is worth noting that two statements, namely declarations and function definitions, will update the type environment. Specifically, a variable declaration will bind a new variable to a type in the environments. Function definitions are slightly more complicated. A function definition is a special declaration where the typechecker first type-checks the body of the function with the environment temporarily augmented with arguments and then bind the function name to its type in the environment. Therefore, the type-checker for statements returns the possibly updated type environment.

We will look at one of the more interesting cases such as type checking an assignment.

With an assignment, we first need to determine the types of the left hand side of the assignment. To do that, we recall that we have an environment that holds all the types of the variables seen so far. We create our list of expected types by searching in the environment and throwing an error if any of the variables can not be found. If an error occurs, that would mean this assignment was invalid since we have never declared this variable before. Once we have the list of expected types, we need to get the actual types. Since the right hand side of the assignment is an expression, we can call our helper for typecheck-expr which either returns one type or a list of types. We will get a list of types in the case where the right hand side of the assignment was a function application where the function can return multiple values. Octave allows the left side of an assignment to have less variables than the number of values on the right side of the assignment if it was a function application. In such a case, Octave will ignore the last n output values where n = (left length - right length). 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 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 side and the right side of the assignment.

```
[(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
                       "Unbound identifier")
                (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
             "Assignment arity mismatch")]
     [(consistent-list? expect-types actual-types)
      env]
     [else
      (error 'typecheck-stmt
             "Assignment type mismatch")]))]
```

Figure 17. Type-checker branch for assignments

Another interesting case is the case for function application.

Function application is an interesting case that is worth studying. 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. Therefore, to type-check a function application, we need to truncate the actual argument list if necessary, and append dummy dynamic types to type-check missing arguments. We check the type consistency between the actual argument list and the expected argument list using the type consistency rule for star types.

Another interesting case is that we may apply a function with type dynamic because the dynamic 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

```
[(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
                    "Argument type mismatch")))
       (error 'typecheck-expr
              "Try to apply a non-function"))))

(: 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 18. Type-checking case for function applications

type and the co-domain type of a dynamic type, which are both dynamic according to the metafunction fun is the typing rules in Figure *. 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 the type check cases are very similar and simpler than the two cases we described above. They can be implemented by calling the type checkers for statements and expressions as well as calling the consistency function to check the consistency of two types.

7 Plan for Next Steps

We intend to enrich and further introduce nuances into our gradual typing scheme by delving into features such as static dimension checking (as matrices are a core component of the language domain) and other more domain-specific aspects. More specifically, in order to meet our team's full project goals we plan on introducing matrices, as well as removing type annotations (i.e. translate back to normal octave).

Our Octave can include matrices to do mathematical calculations such as element operations, vector operations, and other ways to manipulate matrices. This will not be trivial because it's multidimensional, there can be dynamic types as well as static types.

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