Phased Type Checker For Java

A Type Checker For a Subset of Java Built On Scope Graph Semantics

Omar Thabet

Supervisor(s): Casper Bach Poulsen, Aron Zwaan

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Omar Thabet
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Aron Zwaan, Thomas Durieux

Abstract

Traditional type-checking programs are typically designed for specific programming languages, resulting in complex and tightly coupled imperative implementations. One of the challenges faced by type checkers is ensuring consistent name-binding resolution in the presence of new names and symbols added to the environment.

To address this issue, Statix, a domain-specific meta-language for specifying static semantics using scope graphs and constraints aims to reduce the gap between the language specification and the implementation of the type checker. Statix rules define the static semantics of language constructs by employing constraints over type terms and constraints that define and query a scope graph. To demonstrate this concept, Ministatix, an implementation of the Statix core, is utilized to create a sound type checker for a subset of Java. Ministatix resolves name bindings in an interleaved manner as the scope graph is constructed.

This paper examines an alternative method of using scope graphs for type checking, which involves type checking in separate phases in order to ensure stable queries in the scope graph. Thus avoiding complexities associated with name-binding resolution in Ministatix. We provide an explanation of the required phases and compare the phased type checker to Ministatix in terms of supported Java features, testing, code declarativity and readability, and feature extendability.

Key Words: **Type Checking, Phased Type Checking, Scope Graphs, Stable Variable Name Resolution, Java Type Checker**

1 Introduction

According to Harper, R. (2016). [3] type checking is a mechanism that ensures that certain kinds of mismatches cannot arise during the execution of a program and plays a crucial role in ensuring the correctness and reliability of programming languages. It involves verifying the consistency of types within a program, thus preventing potential errors and ensuring proper program behavior. Traditionally, the resolution of names during type checking is accomplished by generating constraints while traversing the Abstract Syntax Tree (AST) based on the language specifications. However, there is often a noticeable gap between the language specification and its corresponding type checker, as these checkers are typically imperative and tightly coupled to specific programming languages [7]. This tight coupling makes it challenging to prove the correctness of the type checker and reuse its components for other languages.

To address these challenges, Scope graphs were introduced by Neron et al. (2015) [5], as a powerful concept that captures the scoping patterns in programs. Unlike traditional approaches that rely solely on the AST for name resolution, scope graphs provide a language-independent representation of scoping relationships. By mapping AST nodes with uniform name resolution behavior to 'scope' nodes within the graph, scope graphs offer a structured and efficient way to analyze variable visibility and type resolution.

In the context of type checking, Scope graphs serve as a structured representation of scoping relationships, enabling accurate navigation and analysis of variable and type references within a program. They enhance the precision and correctness of type-checking processes by tracing variable references to their declarations and enforcing proper usage of types.

The use of scope graphs in type checking offers several advantages. Firstly, they enable precise analysis of scoping patterns, even in languages with complex name-binding rules, ensuring consistent and correct resolution of variables throughout the program. Additionally, scope graphs facilitate the detection of potential type errors by providing a comprehensive view of the program's scoping structure, enabling type checkers to identify inconsistencies and violations of type rules.

Furthermore, scope graphs serve as a foundation for implementing advanced type systems and type inference algorithms. The rich information in the scope graph can be leveraged to support complex type systems, including polymorphism, inheritance, and module-level scoping. Moreover, the structured representation of scoping relationships facilitates the development of efficient type inference algorithms, enabling the automatic deduction of types in programs [1].

Type checking presents a significant challenge when it comes to ensuring stable querying of variable names during the type-checking process. Premature evaluation of queries can lead to unstable answers that are invalidated by subsequent additions to the environment or symbols. Consequently, type-checking programs require careful consideration of the execution order to address this challenge effectively.

To address these challenges in the context of scope graphs, Van Antwerpen et al. (2018) introduced the Statix framework [7], a domain-specific language (DSL) for type system specification that utilizes scope graphs for declarative name resolution. Statix enables high-level specification of type systems using declarative inference rules, providing well-defined semantics for reasoning over type systems while abstracting operational details. Subsequently, Rouvoet et al. (2020) developed Ministatix [6], an implementation of Statix-core [1] (a refined and simplified version of the Statix meta-language) which was used to demonstrate the usage of scope graphs to type check a subset of Java [2]. Notably, Ministatix utilizes critical edges [6] to characterize missing dependencies of queries, which are edges that are part of the final graph but not yet present in the current partial graph.

In this study, we explore an alternative approach to using Statix by adopting a manually phased method for type checking. By manually scheduling the construction and querying of the scope graph, we can ensure stable queries without the

[1] https://github.com/metaborg/ministatix.hs

[2] https://github.com/MetaBorgCube/java.mstx

complexities associated with Ministatix.
The outline of this paper is as follows: In the Problem Description section, we present the purpose and reasoning for this study. The Syntax section explains the data types used to build a Java AST. The Scope Graph Operations section explains functions provided by the phased library for exploring and querying the graph. The Phased Type Checking section delves into the details of the inner workings of the phased type checker and its three phases while comparing its operations to Ministatix. In the Evaluation section, we discuss how the type checker was evaluated and tested. The Discussion section compares the capabilities of the phased type checker to Ministatix and the standard Java type checker. The Responsible Research section addresses ethical considerations. Finally, we provide a conclusion based on the findings of this study and discuss future research on this topic.

2 Contribution

This study makes the following contributions:

- Demonstration of the phased approach through the implementation of a type checker for a subset of Java.
- Discussion of the differences between the phased type checker and Ministatix.
- Evaluation of the testing methodology and comparison of the supported Java features in the aforementioned type checkers.
- Comparison of the declarativity and extendability of the code in the two type checkers.

3 Problem Description

The current Ministatix implementation of a type checker for a Java 8 subset handles the stable querying of the scope graph with considerable complexity. In this study, we aim to explore an alternative approach by running the type checker program in phases to establish stable querying. This approach involves constructing and querying the scope graph in a manner that ensures coherence with the Java specification. Our research questions include: How does this phased approach compare to the Ministatix implementation? How many phases are required for effective implementation? How easily can the program be extended to include additional Java features, and will this necessitate additional phases?

4 The Syntax and Supported features

The initial step in the type-checking process involves parsing the Java code into an Abstract Syntax Tree, which is represented using Haskell data types. The type checker then operates on this AST by performing multiple recursive passes, where each pass gathers or utilizes information from previous phases. Pattern matching plays a crucial role in this process, as it allows developers to write clear and readable code for handling complex data structures [4].

The AST representation used in our program is inspired by the Ministatix implementation. It includes components such as packages, compilation units, statements, and expressions.

```
data JavaType
  = IntType
  | LongType
  | FloatType
  | DoubleType
  | BooleanType
  | CharType
  | StringType
  | ObjectType String
  | ArrayType JavaType
  | Void
  deriving (Eq, Show)
```

Figure 1: Java Types are represented using the `JavaType` datatype

However, there are some notable differences between our implementation and Ministatix, as our program supports a different subset of Java features. These differences stem from the scope of our study and the focus on exploring the phased approach to type checking.

The `JavaType` data type, as depicted in Figure 1, serves as a representation of Java types within the program. It encompasses a subset of the Java primitive types, and classes represented using the `ObjectType` construct.

Expressions are represented by the `Expression` data type as displayed in Figure 2, it covers a wide range of expressions, including literals, variable identifiers, method calls, binary operations, unary operations, field access, and more. This subset of Java expressions provides sufficient coverage to test the functionality of the type checker and enables the construction of complex programs.

```
data Expression
    = LiteralE Literal
    | VariableIdE String
    | MethodCallE String [Expression]
    | BinaryOpE Expression BinaryOp Expression
    | UnaryOpE UnaryOp Expression
    | ThisE
    | NewE String [Expression]
    | FieldAccessE Expression String
    | MethodInvocationE Expression String [Expression]
    deriving (Eq, Show)
```

Figure 2: Java expressions are represented using the `Expression` datatype

Statements are captured by the `Statement` data type shown in Figure 3. It includes assignment statements, if statements, while loop statements, variable declaration statements, return statements, break and continue statements, and expression statements.

The difference between expressions and statements is that Expressions are evaluated to produce a value, On the other hand, statements are units of code that perform actions or control the program's execution flow. They are typically standalone lines of code, terminated by a semicolon (;).

```haskell
data Statement
  = AssignmentS Expression Expression
  | IfS Expression [Statement] (Maybe [Statement])
  | WhileS Expression [Statement]
  | VariableDeclarationS JavaType String (Maybe
      Expression)
  | ReturnS (Maybe Expression)
  | BreakS
  | ContinueS
  | ExpressionS Expression
  deriving (Eq, Show)
```

Figure 3: Java statements are represented using the `Statement` datatype

A Java program is represented as a list of `JavaPackage`, and each instance represents a Java package that contains Java classes. The `CompilationUnit` data type represents an individual file containing a single `ClassDeclaration` along with a list of imported classes. Each class contains can contain a list of contractors as well as a list of `Members`. Figure 4 shows an example of a Java program and its representation in Haskell.

```java
package PA;
import PB.B;

public class A {
 public void method(){
        B b = new B();
    }
}
```

```haskell
-- Haskell code:
program :: [JavaPackage]
program = [JavaPackage "PA" [classAUnit]]
 where
    classAUnit =
        CompilationUnit
            [ ImportDeclaration "PB" "B" ]
            (ClassDeclaration "A"
                        [MethodDeclaration Nothing "method" []
                            [ VariableDeclarationS (ObjectType "B")
                                "b" (Just $ NewE "B" [])]]
                        False
                            [])
```

Figure 4: An example Java Program with the equivalent AST representation in Haskell.

5 Scope Graph Operations

The type checker makes use of the Phased Haskell library [3] in order to construct and query the graph. We make use of two types of nodes, 'Scope' nodes and 'Sink' Nodes.

5.1 Scope Graph Structure

In order to indicate that a scope x is associated with a scope y with relationship L we use the function `edge x L y` that creates a directed edge labeled L from the node of scope x to the node of scope y, this indicates that if a variable is visible in scope y then it is also visible from scope x by following a path with the regular expression `(L)`. This allows for independent scopes that can be passed around and extended without the need for explicit aggregation, and remotely accessed without explicit distribution [6].

Moreover, there are two possible relationships between scope, scope y is a lexical parent of scope x when scope x is attached to scope y with an edge labeled P, Secondly, we indicate an import with labeled with I. The usage of multiple labels is useful as they allow for different name resolution rules according to the language specification [10].

Each scope can contain declarations of name bindings, which can pertain to classes, packages, variables, or methods. A declaration is represented by a sink, where each sink is attached to a scope with a labeled edge and contains data associated with the declaration. Sinks can be assigned one of four labels: D for variables, methods, and constructors; M for packages; Cl for classes; and T for scope types. For example, if we want to declare a variable x of type *String* within scope s we would use the notation `sink s D (VarDecl x t)`, this means that variable x is accessible within scope s by following a path that terminates with label D. Table 1 provides a comparison between the labels utilized in the phased type checker and those used in the Ministatix Implementation. Note that in the case of Ministatix's interleaved approach, distinct labels are necessary for declarations in order to differentiate them during the calculation of dependency constraints to ensure the monotonicity of query results as explained in subsection 5.3.

Label Purpose	Ministatix	Phased TC
Lexical Parent	Lex	P
Import	IMP-ST,IMP-OD	I
Package Declaration	PKG	M
Class Declaration	Type	Cl
Scope Type	THIS	T
Constructor Declaration	CTOR	D
Method Declaration	METHOD	D
Field Declaration	FIELD	D
Variable Declaration	VAR	D

Table 1: Comparison between labels used in the Ministatix and the Phased type checker

5.2 Querying the Scope Graph

Resolving name references in both type checkers is achieved using queries. A query is not a part of the scope graph itself, but rather an algorithm that traverses the scope graph based on specific query parameters. Both Ministatix and the Phased type checker queries require a starting scope and a regular expression that represents the path to be followed during the traversal.

The query results can be further filtered based on criteria such as the length of the path or the declaration data at the end of the path. This allows for prioritizing certain paths or

[3] https://github.com/heft-lang/hmg

filtering results based on specific name bindings and declaration data.

For example, say we want to query scope *s* for `Class x`, There are two possible paths that can lead to a class declaration: a locally declared class within the same package, following a path of the form `(P* Cl)`, or an imported class, following a path of the form *(P* I Cl)*.

5.3 Monotonic queries

When querying a name in an incomplete scope graph, it is crucial to ensure that the query result remains unchanged even when the scope graph is extended with additional scopes and declarations. For example, let's consider the name x_r is referenced in scope *s* which resolves to a declaration X_D in a lexical parent of scope *s*. However, suppose that at a later step, the scope graph is extended with a new declaration $X`_D$ within scope *s* itself. if "x" would have been queried after that step $X`_D$ will shadow $X`_D$, altering the query's outcome. This highlights the fact that relying solely on querying the complete portion of the scope graph is insufficient for achieving stable query results [1]. The challenge of stable querying is not unique to scope graphs; it is a problem that every type checker must address, either implicitly or explicitly [6]. In this work, the presented type checker handles type checking in phases giving queries inherited stability by ensuring that no new outgoing edges with the same label are attached to a scope after it has been queried. If such an edge is created, the program will raise a `Monotonicity Error`.

Another example is shown in Figure 5, if the method body is evaluated before the field *x* is found, when we query for the variable name *x* the query will return no results since Out of order declarations are not allowed in Java as explained in subsection 6.3, however when the field is discovered and added to the class scope, this will change the result of the query.

```
public  class ClassA {
    public boolean method(){
        boolean y = x > 0;
        int x = //
    }
    public int x = 0;

}
```

Figure 5: An Example That could cause a Monotonicity Error if the order of the querying was done incorrectly.

6 Phased Type Checking

Unlike Ministatix, the type checker presented in this work follows a predetermined phasing approach, which consists of three distinct phases. The first phase involves traversing all the files in the program and exploring and registering all packages within the program scope. Each package has its own scope, and all the classes defined within that package are explored and registered within the package scope. This step creates declaration edges, also known as sinks, for all the packages and classes. In the second phase, the focus is on the fields and methods within each class. They are explored and registered within the corresponding class scope. However, only the left-hand side of the fields and methods is checked. This means that the initial values of fields and the bodies of methods are disregarded. Finally, in the third phase, the method bodies and right-hand sides of fields are analyzed. This phase involves type-checking expressions such as method calls and literals, as well as analyzing statements within method bodies, such as conditional statements.

6.1 The First Phase: Packages and Classes

The first phase of the type-checking process focuses on exploring packages and classes within the program. It starts with an empty scope graph, initially consisting of a single node representing the program scope.

For each package, a new scope is created and connected to the program scope using an edge labeled *P*. This newly created scope represents the package's own scope. Additionally, a sink labeled *M* is added to the program scope, containing the package name and a reference to its scope. In Ministatix, this would be represented by an edge labeled *LEX* from the package scope to the program scope, and the package declarations would be represented by a backward edge labeled *PKG* from the program scope to the package scope.

Moving forward, the exploration continues by examining each package individually. As each class is encountered, a new scope is created for that class and attached to the corresponding package scope using an edge labeled *P*. A sink labeled *Cl*, containing the class name and a reference to its scope, is added to the respective package scope. In Ministatix, the edge from the class scope to the package scope would be labeled *LEX*, and the class declaration would be represented by a backward edge labeled *TYPE*.

To facilitate the resolution of the `this` keyword in the third phase **??**, a sink with label *T* is attached to each class scope, containing the type of the class represented by that scope. Ministatix achieves a similar effect by using a circular edge labeled *THIS* attached to the class scope.

By systematically exploring and establishing scopes for packages and classes as depicted in Figure 6, this first phase lays the groundwork for the subsequent phases of the type-checking process. With all package and class declarations in place, they can be queried freely without concerns about changing query results.

6.2 The Second Phase: Imports and Class Members

After completing the first phase, where all packages and classes are explored and added to the scope graph, the second phase focuses on handling imports, constructors, fields, and methods within each class.

The first step of the second phase deals with imports. If `class A` imports `class B`, an import edge labeled

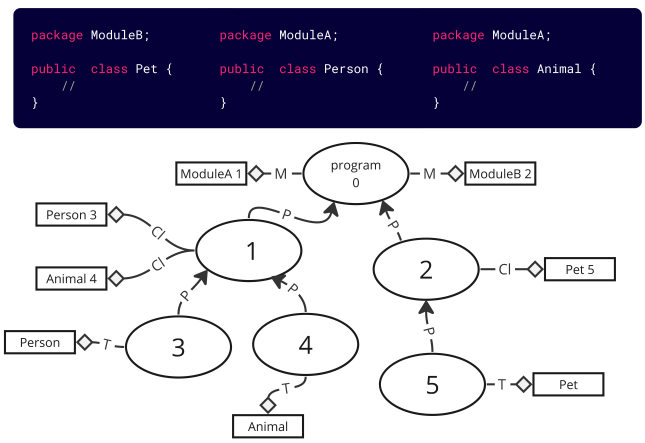


Figure 6: An example program with its equivalent scope graph after the first phase is completed.

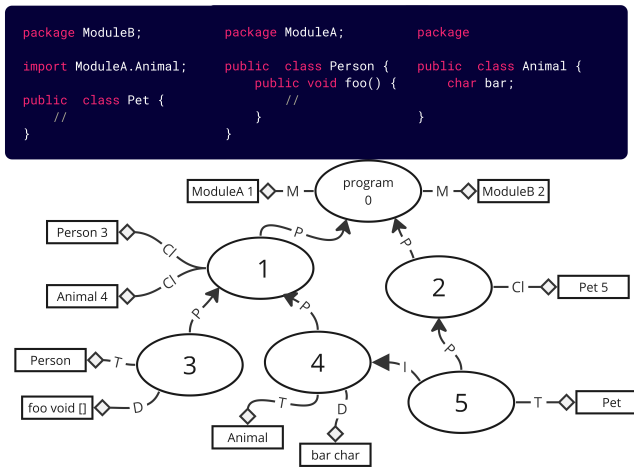


Figure 7: The extended graph from Figure 6 after the second phased is completed.

I is added from the scope of `class A` to the scope of `class B`. This ensures that the necessary connections are established between the importing class and the imported class. In Ministatix, the representation of imports in the scope graph is different, as it interleaves the resolution of imports with the discovery of packages and classes. However, In the final scope graph generated by Ministatix, imports are represented by edges with labels *IMP-ST* for specific type imports and *IMP-OD* for on-demand imports.

After the import edges have been added, the type checker proceeds to check the class constructors. Several conditions are verified during this process. If a class is declared as static but has a constructor, an error is generated since static classes cannot be instantiated with constructors. Conversely, if a class is non-static and contains explicit constructors, a sink labeled *D* is added to the class scope for each constructor. This sink contains the class name and parameter list. It's important to note that at this stage, the constructor bodies are ignored. If no explicit constructor is declared, a default constructor is added to the scope as a sink labeled *D* with the class name and an empty parameter list. This default constructor is equivalent to the inherited class constructor available for all non-static Java types [2]. In Ministatix, only a single constructor per class is supported. The constructor is declared with the label *CTOR* and its newly created scope is attached to the class scope using an edge labeled *LEX*.

The subsequent step in the second phase involves validating the class fields. Each field declaration must specify both the field name and its corresponding type. The type checker ensures that the field types are visible within the current scope. This is achieved by querying the scope using a specific path pattern. If the field type is locally available, the query follows the path pattern (P T). Conversely, if the type is imported, the path pattern becomes (P I T). Note that locally declared types take precedence over imported types [2]. If the field type passes the type-checking process, a sink labeled *D* is added to the class scope, containing the field name and its type. Ministatix employs a similar approach when validating field types. It queries the scope for the field type, taking into account inheritance support. Thus query parameters include additional paths, resulting in a path regular expression of *(LEX* (PKGC | IMP-ST | IMP-OD PKGC?)? EXT* IMPL* TYPE)*. This pattern encompasses local types, imported types, and inherited types.

Finally, the class method declarations are validated. This involves checking the types the visibility of the parameter list and, if applicable, the return type of each method declaration. Similar to the validation of field types, the type checker queries the scope to ensure the visibility of the required types. If the method declaration is valid, a sink labeled *D* is added to the class scope. This sink contains the method name, its return type, and its parameter list. In Ministatix, the equivalent representation is an edge labeled METHOD is used for method declarations.

The reason why sinks for constructor and method declarations contain the parameter list is to allow for method overloading, which is the ability to define multiple methods with the same name but different parameter types [2].

6.3 The Third Phase: Field Values and Method Bodies

The third phase of the type-checking process is dedicated to handling the right-hand side of declarations, including field initial values, constructor bodies, and method bodies. It involves iterating over all class members and performing type-checking on their corresponding expressions and statements.

To begin, new scopes are created for each method and constructor. The parameters of each function are added to their respective scopes as sinks labeled *D*, which contain the variable name and type. This allows for immediate visibility and accessibility of the parameters within their respective method scopes.

The initial values of field declarations are not supported in Ministatix, they are however supported in the phased type checker, the initial value is represented by an `Expression` which is type-checked and evaluated to a type; If there is a mismatch between the evaluated expression type and the declared field type, an error is thrown.

Type Checking Expressions

In the type-checking process, simple expressions like literals or arithmetic operations can be handled through basic pattern matching. However, more complex expressions often require querying the scope graph to resolve variable name references.

One important expression to consider is the `this` keyword. The `this` keyword is an expression that can be used to qualify method calls or field access expressions. It can also be used to reference the containing class. The `this` keyword is allowed within constructors, method bodies, or field initialization [2].

In order to resolve the `this` expression, the type checker queues the scope for the nearest sink with label *T* following the shortest path with the regular expression `(P* T)` depicted by the blue arrows in Figure 8. The result of the query will be the type information of the nearest class declaration, which then can be used to access the field or methods declared in the class scope.

In Ministatix, the handling of the `this` keyword follows a similar approach. It resolves the `this` keyword by querying the scope for paths with the regular expression (LEX* THIS). This query allows Ministatix to identify the shortest path to the class scope, taking into account the possibility of nested classes which are supported in the Ministatix implementation. By giving priority to the shortest path, Ministatix can resolve the `this` keyword and retrieve the necessary type information.

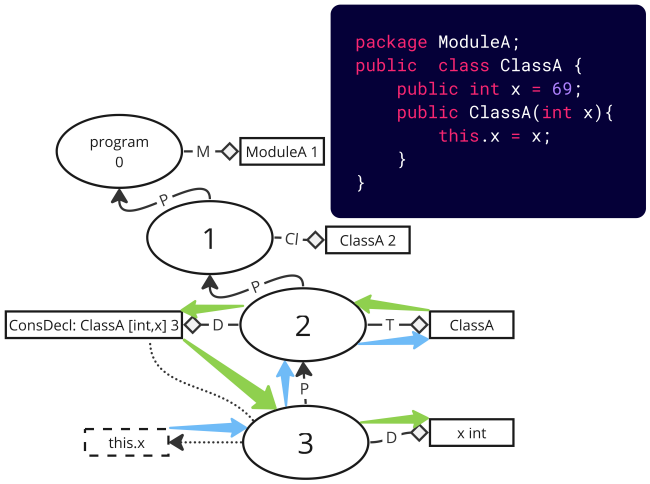


Figure 8: Example usage of the `this` keyword to refer to a field instead of a locally declared variable with the same name.

Field access expressions contain two components, an expression to select the object to be accessed followed by the name of the field to select. Therefore resolving the field access expression is done in three steps. First, the expression needs to be evaluated in order to find the name of the type that is being acceded. Once we have the name of the object, we need to query its scope for the field name, in order to get the object scope we need to query the scope containing the field access expression for the name of the class of the object, the query takes a path regular expression of the form `(P*Cl | P* I Cl)` as the object could be locally declared or imported. The query result will contain a reference to the object scope which is queried for the field name, the regular expression given to the query is `(D)` in order to find the field declaration. Ministatix differs in the field query regular expression as it supports selecting inherited fields therefore the path has the regular expression `(EXT* FIELD)`. Field access expression `this.x` in Figure 8 is represented by the blue arrows to resolve `this` and by the green arrows to resolve `x`.

Method invocations and expressions involving the `new` keyword follow a similar process to field access expressions. However, there is an additional step involved, which is type-checking the arguments of the method or constructor and ensuring that they match the declared parameters.

To find the method or constructor, a query is performed using a path with the regular expression `(D)`. It is worth noting that the query may return multiple results if the method is overloaded, meaning there are multiple methods with the same name but different parameter lists. In such cases, the type checker compares the given arguments to the method parameters in each query result to determine the correct method or constructor to be invoked.

Type Checking Statements

Statements represent the contents of method and constructor bodies. They can declare new variables in scope or alter the execution order of the program. For some statements, such as loops, separate scopes are required. During the type-checking process, the statements are evaluated in the order they appear within a code block.

When an if statement or a loop statement is encountered, a new scope is created specifically for the bodies of those statements. This new scope is connected to the parent scope using an edge labeled *P*. Additionally, the type checker ensures that the `continue` and `break` statements are appropriately used within loops only.

In Java, the `return` statement is used to exit a method and return a value to the caller. The type checking of the ”return” statement involves several steps. First, for methods with non-void return types, a return statement must be included to return a value of the specified type. The ”return” statement can also be used to exit a method prematurely based on certain conditions. However, any code after a ”return” statement is considered unreachable, leading to an error. Conditional returns are allowed, but the type checker ensures that all possible execution paths lead to a ”return” statement.

The processing of variable declaration statements is similar in both Ministatix and the phased type checker. A new scope is always created for the remaining statements in the block to avoid monotonicity errors caused by adding a new declaration to an already queried scope. This approach allows for a multi-level scope hierarchy without requiring an additional phase for type-checking method bodies. Additionally, it enforces the rule that variable names cannot be used before their declaration in Java [2].

After creating the new scope, it is connected to the parent scope using an edge labeled *P* in the phased type checker, or *LEX* in Ministatix. The variable is then declared in the new scope by adding a sink labeled *D* that contains the variable type and name. This declaration process corresponds to the edge labeled *VAR* in Ministatix.

If the third phase of type-checking is completed without any errors, it indicates that the program has successfully passed the type-checking process, and the scope graph is considered complete. An example of a program and its corresponding scope graph representation can be seen in Figure 9.

7 Evaluation

In this section, we demonstrate how the phased type checker was evaluated including the testing methodology and how the scope graphs were constructed.

7.1 Test Cases

Similar to Ministatix, The evaluation of the phased type checker involved two types of test cases: those targeting correctly written code and those examining code that was expected to fail the type-checking process. These test cases were designed to provide a comprehensive assessment of the type checker’s capabilities and to identify areas for improvement.

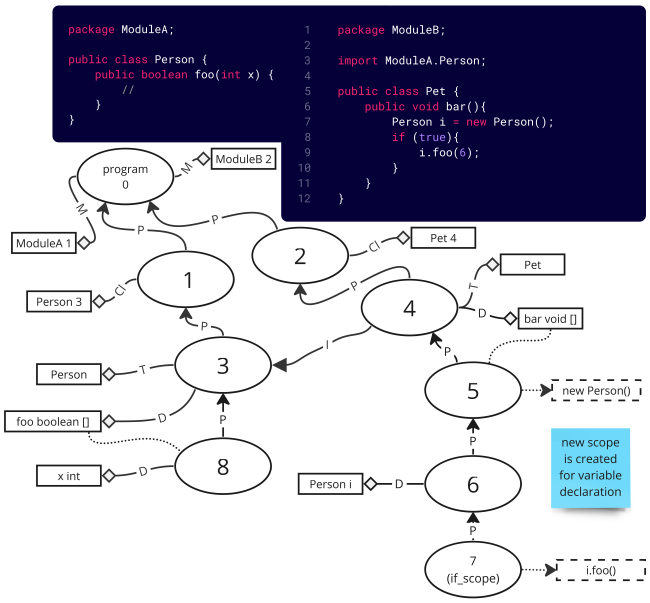


Figure 9: An example program with its complete scope graph.

The test suite consisted of 50 individual test cases, including all the compatible test cases from the Ministatix implementation. The goal was not to achieve 100% code coverage, but rather to focus on various edge cases and diverse combinations of statements and expressions as well as class structures with varying levels of complexity. Figure 10 provides an estimation of the lines of code covered by the test suite. The graph was generated based on the lines of code reached within the different functions that perform type-checking on packages, imports, classes, class members, statements, and expressions.

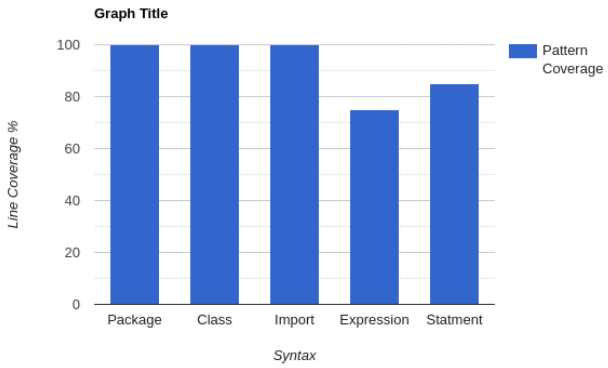


Figure 10: Line coverage estimation

Figure 11 illustrates the distribution of the test cases across the different components of the type checker. The majority of the tests focused on statements and expressions, as they include a wider range of Java features compared to Ministatix. Additionally, a significant proportion of testing was dedicated to compilation units (classes and imports), as these are crucial aspects of the type-checking process and are gradually type-checked across the three phases.

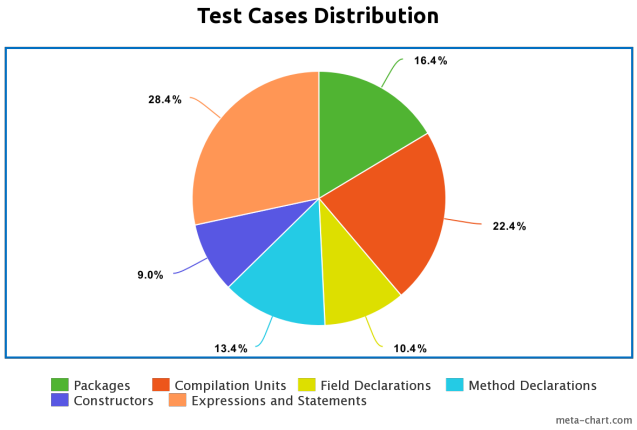


Figure 11: A pie-chart showing the distribution of the test cases over the diffrent componenets of the type checker

Test Cases for Correctly Written Code

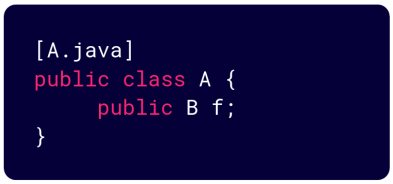
Approximately half of the test cases were designed to assess the ability of the type checker to correctly identify properly typed Java code. Each of these test cases was first checked using the standard Java type checker to ensure there were no false negative outcomes by the phased type checker.

During the initial stages of development, small-scale test cases were employed, with each test targeting a specific feature or stage of the type checker. Figure 12 provides an illustrative example of such a simple test case. These initial tests played a crucial role in uncovering numerous bugs in the type checker such as the addition of sinks with invalid data.

During the final stages of development, the complexity of the test cases was escalated to encompass multiple packages and classes. This comprehensive approach aimed to verify not only the accurate functioning of individual components within the type checker but also the seamless integration and compatibility of these components. Special attention was given to potential issues such as name shadowing and method overloading, which are critical aspects of the Java language specifications [2]. The goal was to ensure that the type checker adhered strictly to these specifications and avoided any instabilities or errors that could arise from the interaction of various components.

```
package PackageB;

public class ClassB {

    public char b = 'a';

    public ClassB() {
    }

    public String sayHello(String name) {
        return "hello " + name;
    }
}
```

Figure 12: An example of a simple test case that validates the correct construction of a scope graph containing a single class with arbitrary members



a) Phase TC error: "Type B doesn't exist in scope"
b) Ministatix Error:"java.names.lexical-type-name.*Id\(\"B\"\)"
c) Java Compiler error: "cannot find symbol"

Figure 13: An example program with incorrect code resulting in an error

Test Cases for Error Detection

The remaining test cases were specifically designed to evaluate the error detection capabilities of the type checker. These test cases consisted of code snippets that intentionally violated type rules or contained other type-related errors. In order to ensure accurate error detection, the error outputs of the phased type checker were compared with those of the standard Java type checker. This comparison allowed us to identify any potential false positives and ensure the reliability of the type checker's error reporting.

One crucial aspect of these error detection tests was not only to identify the presence of typing errors but also to assess the type checker's ability to generate meaningful error messages and accurately pinpoint the location of the error within the code. This information was essential for developers to understand and correct the identified issues efficiently.

Additionally, for compatible tests, the error outputs of Ministatix and the phased type checker were compared. This comparison served as a validation step to ensure that the phased type checker was able to detect the same errors as Ministatix, further confirming its accuracy and effectiveness. An example of an incorrect program and the corresponding error messages from both type checkers as well as the Java compiler is illustrated in Figure 13.

The phased type checker exhibits a weakness in correctly identifying an error caused by importing a class with the same name from a separate package, instead, it generates a circular import message intended for cases when a class imports itself. Although this does not result in a false negative output, the incorrect error message indicates a flaw in the phased approach. This issue stems from the limitation of traversing the scope graph in a single direction, making it challenging to rectify this bug without a considerable redesign of the package declaration handling in the first phase.

In contrast, the Ministatix implementation accurately detects these errors due to its different approach to handling packages and imports. This disparity highlights the need for further improvement in the phased type checker to address such scenarios effectively.

7.2 Analyzing Results

Testing alone cannot guarantee the absence of bugs, which is why it is important to analyze the steps taken by the phased type checker to ensure the correct construction of the scope graph. To facilitate this analysis, the type checker logs its steps using the `Debug.Trace` library. Analyzing these steps helps uncover design issues that may not be detectable through tests alone but could cause problems when extending the type checker with additional Java features. Most notably, the incorrect error message bug explained in section 7.1 was discovered after analyzing the logs.

7.3 Limitations of Testing

While the evaluation process provided valuable insights into the functionality and effectiveness of the type checker, it is important to acknowledge its limitations. Analytical proofs of the type checker's correctness were beyond the scope of this research, and the evaluation focused primarily on empirical testing.

8 Responsible Research

In the development of the type checker, several considerations were taken into account to ensure responsible research practices and the overall quality of the type checker.

First and foremost, an important aspect was to ensure that the type checker provides meaningful error reports in cases where the program does not type check. The error reports were designed to be clear, informative, and easy to understand. This allows developers to quickly locate and address any type-related issues in their code.

Additionally, as this type of checker was developed as a proof of concept, it was crucial to document all aspects of its design, implementation, and functionality in order to help future researchers build on our work.

To ensure the correctness of the type checker, a comprehensive test suite was created. This test suite included various test cases, including edge cases, to detect both false positives and false negatives. By rigorously testing the type checker, its accuracy and reliability were validated, instilling confidence in its performance.

During the development process, the utilization of large language models played a crucial role in improving efficiency. These models facilitated the automatic generation of repetitive lines of code, saving valuable time during development. By automating repetitive code generation, the focus was redirected towards addressing more complex and critical aspects of the type checker. This approach resulted in increased productivity and accelerated the development process. A notable example of code generation was the creation of pattern-matching cases for all supported Java primitive types and literals, streamlining the implementation process.

9 Discussion

In this section, we will discuss the findings of our work and compare the phased type checker to the Ministatix type checker.

9.1 Feature Support

The Java subset supported in Ministatix and the phased type checker are different, the Ministatix implementation focuses on hierarchical package and class structures, interfaces, and inheritance while the phased type checker focus on type checking the inner working of classes as it supports a wider range of Expressions and Statements.

The table shown in figure 2 highlights the main differences in supported features.

9.2 Scope Graph Representation and Query Stability

The construction and querying of the scope graph differ significantly between the phased type checker and Ministatix. In the phased type checker, as discussed in the section 5.1, the scope graph consists of two distinct types of nodes: scope nodes and sink nodes. Scope nodes represent scopes and are allowed a single outgoing edge to another scope labeled as the lexical parent (labeled as P). Additionally, scope nodes can have multiple outgoing edges to other scopes labeled as imports (labeled as I). Sink nodes are used for declarations and have a single incoming edge from a scope.

The labels used for sinks and scopes are two disjoint sets, allowing for easy distinction and enabling the representation of paths in

Supported Java Feature	Ministatix	Phased TC
Arrays	FALSE	Creation and Indexing
Binary Operations	FALSE	Partial
Break Statement	FALSE	TRUE
Cast Expression	TRUE	FALSE
Class and Method Access Control	Public Only	Public Only
Compilation units	TRUE	Single class per CU
Conditional Operators	FALSE	TRUE
Constructor Declarations	Partial	TRUE
Continue Statement	FALSE	TRUE
Equality Operators	FALSE	TRUE
Fully Qualified Names	TRUE	FALSE
If Else Statements	FALSE	TRUE
Import on Demand	TRUE	FALSE
Inheritance	TRUE	FALSE
Initial Values of Variables	FALSE	TRUE
Inner Classes	TRUE	FALSE
Interface Types	TRUE	FALSE
Local Class Declarations	TRUE	FALSE
Method Body	Partial	Partial
Method Invocation Expressions	FALSE	TRUE
Nested Packages	TRUE	FALSE
Overloaded Constructors	FALSE	TRUE
Packages	TRUE	Partial
Primitive Types and Values	Partial	Partial
Return Statement	Partial	TRUE
Static Classes	FALSE	Partial
Subtyping	TRUE	FALSE
Type Inference	FALSE	FALSE
Unary Expression	FALSE	TRUE
Unreachable Statements	FALSE	TRUE
Variables of Primitive Type	FALSE	TRUE

Table 2: Comparison of Java Features supported in Ministatix and the Phased Type checker, partial means that the feature is present, however not all it's variations are supported

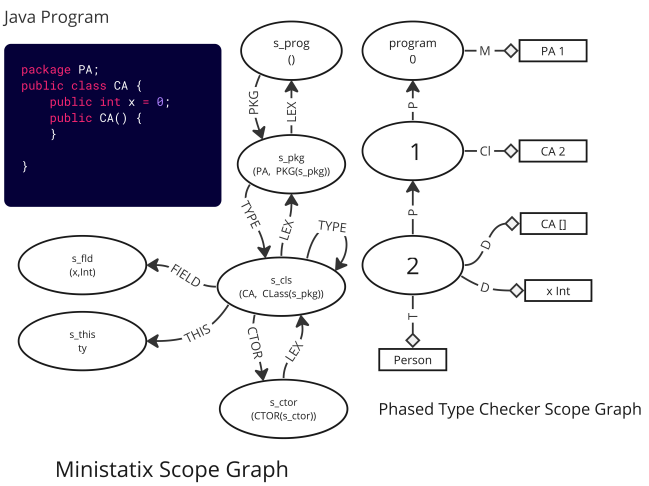


Figure 14: A side-by-side comparison between the scope graph of the Ministatix Type checker and the Phased Type Checker

the scope graph as easily distinguishable regular expressions. Additionally, the phased type checker traverses the scope graph in a single direction, from lower scopes to higher scopes, eliminating the need for backward edges.

On the other hand, Ministatix can use a single node to represent both scopes and declarations, reducing the overall number of nodes in the scope graph. However, this approach requires edges in both directions to capture scoping and name bindings. Due to the interleaved order of type checking in Ministatix, the scope graph can be traversed in both directions, Figure 14 shows a comparison between the scope graphs generated by the two type checkers for the same program.

Regarding label usage, Ministatix requires additional labels to differentiate between Field, Method, Constructor, and Variable declarations in the scope graph. In comparison, the phased type checker utilizes a single label for declarations because the necessary information is contained within the sink node's data.

Both type checkers ensure query monotonicity 5.3 but employ different approaches to achieve this goal, The phased approach as explained in Section 6 allows for inherited query stability as it's guaranteed that the query results will not be changed with subsequent additions to the scope graph [1], On the other hand, Ministatix achieves query stability by leveraging the concept of critical edges. Critical edges indicate that query results may depend on missing information. By delaying queries that rely on such information until all relevant binding structure has been collected, Ministatix guarantees stable query answers. In essence, the absence of critical edges implies the availability of a complete binding structure and stable query results [6].

9.3 Declarativity and Extendability

Both implementations of the type checker aim to bridge the gap between the language specification and the implementation of the type checker. In this section, we compare the declarativity and readability of the two type checker implementations.

Ministatix is built on Statix, which is a high-level static semantics definition language. It allows for the generation of a type checker derived from the language specifications. As a result, the code in Ministatix is more declarative, as the ordering of queries is abstracted using language-independent critical edges.

On the other hand, the phased type checker relies on a manually determined order of type checking. This reduces the declarativity of the code, as the construction of the scope graph across different phases becomes more imperative.

The phased type checker has a great advantage over the Ministatix in terms of ease of understanding the flow of the program, since the program works across separate and clearly defined steps, it's easier for newer developers to understand the steps taken by the program, unlike Ministatix where everything is interleaved and might be difficult to understand, especially when they have little experience with the Statix specifications.

In terms of code readability, it is crucial for code to be easily understandable by developers. Unreadable code can hinder program comprehension and lead to the introduction of bugs [8]. The phased type checker suffers from being entirely written in a single Haskell module, making it challenging to separate concerns and compromising good software architecture design. This compromise was made due to the limited time available for the project, prioritizing rapid development and testing. Figure 15 shows a comparison between code snippets from the two type checkers.

Furthermore, The Ministatix implementation lacks comprehensive code documentation, making it difficult to understand the exact functionality of each function. However, the declarative nature of the code in Ministatix helps mitigate this issue to some extent. In contrast, the phased type checker includes as much documentation as possible to enhance code clarity.

Moreover, compared to Ministatix, it's easier to understand the control flow of the program as each step is clearly defined affecting only a single aspect of type checking at a time. This results in a less declarative program as we need to imperatively program each phase.

The extendability of the phased type checker was a key consideration during its design. We believe that the phased type checker can be extended to support a larger subset of Java features without a major redesign of the program. Let's consider two examples to illustrate this.

First, let's look at incorporating a new language construct, such as an enhanced for loop, into the phased type checker. This construct simplifies iteration over elements of an iterable collection. To han-

][h]

Phased Type-Checker

```
discoverPackages :: (Functor f, Error String < f, Scope Sc Label Decl < f)     =>
    JavaPackage  -> Sc -> Free f ()
discoverPackages (JavaPackage n cus) programScope = do
  packageScope <- new
  sink programScope M $ PackageDecl n packageScope
  edge packageScope P programScope
  mapM_ (`discoverPackageClasses` packageScope) cus
```

Ministatix

```
compilation-unit-ok(s, cu) :- cu match
  { CompilationUnit(pkgDecl, imports, typeDecls) -> {s_pkg, d_pkg}
      new s_pkg -> d_pkg
      package declaration
    , s_pkg -[ LEX ]-> s
    , package-declaration-ok(s, pkgDecl, s_pkg)
    , imports-ok(s, imports, s_pkg)
    , type-declarations-ok(s_pkg, typeDecls)
  }.
```

Figure 15: A code snipped taken from function responsible for package declarations from the two type checkers

dle this feature, the third phase responsible for statement analysis and expression type checking can be extended. During the statement analysis phase, the type checker can recognize the enhanced for loop syntax and validate its usage according to the Java language specification. It can ensure that the variable declaration, iterable expression, and loop body conform to the expected types and semantics. Similarly, during the expression type-checking phase, the type checker can verify that the elements accessed within the loop body have the appropriate types based on the declared type of the iterable collection

Secondly, there are scenarios where introducing certain features may require additional phases and further modifications. In the case of subtyping and inheritance, an additional step between the first and second phases would be needed. This new step would involve adding the inheritance hierarchy to the scope graph and connecting the scope of child classes to their superclasses using a new edge label, similar to the approach taken in the Ministatix implementation of subtyping. Additionally, the evaluation of certain expressions, such as field access expressions or method invocation expressions, would need to be modified to consider superclasses.

10 Conclusion and Future Work

In conclusion, we have explored two possible approaches to type-check Java using scope graph semantics. We have discussed the key features, design choices, and trade-offs of each approach, highlighting their strengths and weaknesses.

The phased type checker offers a modular and step-wise approach to type checking, dividing the process into distinct phases. This allows for a type checker that is easier to trace its steps and understand the flow of its execution. However, incorporating additional Java features will necessitate modifications to the current implementation, including the introduction of additional phases to facilitate correct scope graph construction and ensure stable query results.

On the other hand, the Ministatix implementation, built on the Statix framework, offers a more declarative and language-independent approach to type-checking. It provides query stability through the critical edge mechanism. Nonetheless, the code readability of the implementation could be improved with better code documentation, making it easier for developers to understand the purpose and functionality of each function.

In conclusion, both the phased and the Ministatix type checker implementations help narrow the gap between the language specification and their type checker and provide valuable insights into the design and implementation of type checkers using scope graphs laying the groundwork to further innovations to take further advantage of scope graphs.

Finally, The continuation of this project would involve extending the implementation of the phased type checker to support the same subset of Java features as Ministatix. This would include reviewing the necessity for additional phases and ensuring the correct construction of the scope graph as discussed in section 9.3. Additionally, exploring techniques to optimize the query function, such as early termination, could further enhance the performance of the type checker. Furthermore, the phased type checker can be extended to implement parallelization, similar to the approach proposed by Van Antwerpen and Visser (2021) [9]. Parallelization can significantly speed up the type-checking process by leveraging multiple threads or processors.

Dedication

I dedicate this thesis to my friends, family, and course supervisors, who have been unwavering in their support and encouragement throughout my academic journey. Their belief in me has been a constant source of motivation and inspiration.

I would also like to express my heartfelt gratitude to my colleagues at Vintus, who have played a significant role in my growth and development. Their guidance, knowledge, and camaraderie have been invaluable, and I am grateful for the opportunities to learn from them.

Finally, I would like to extend a special thank you to IQ, whose unwavering love and support have been my moon during my darkest nights and the most challenging times. Your belief in me has given me strength and determination to overcome obstacles and pursue my dreams.

References

[1] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, page 49–60, New York, NY, USA, 2016. Association for Computing Machinery.

[2] James Gosling. The java language specification: Java se 8 edition. *(No Title)*, 2015.

[3] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016.

[4] Graham Hutton. *Programming in haskell*. Cambridge University Press, 2016.

[5] Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*, pages 205–231. Springer, 2015.

[6] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: Sound scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[7] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[8] Hendrik Van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[9] Hendrik Van Antwerpen and Eelco Visser. Scope states: Guarding safety of name resolution in parallel type checkers. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[10] Aron Zwaan and Hendrik van Antwerpen. Scope graphs: The story so far. In *Eelco Visser Commemorative Symposium (EVCS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.