Scope Graph-Based Type Checking for a Scala Subset

Building Type Checkers Using Scope Graphs

Radu Mihălăchiuță[1]

Supervisors: Casper Bach Poulsen[1]**, Aron Zwaan**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

This paper investigates the viability of using scope graphs to implement type checkers for programming languages, specifically for a Scala subset. The primary objective is to determine if scope graphs can offer a declarative and extensible approach to type checking. To achieve this, we used a phased Haskell library to implement such a type checker. The declarativity and feature extensibility of the approach were evaluated by means of comparation with Rouvoet et al.'s approach in mini-Statix. The results demonstrate that using scope graphs as a basis for type checking yields a modular and extensible solution compared to traditional methods. However, it is noted that this approach may sacrifice a certain degree of declarativity. These findings suggest that scope graphs are a promising tool for type checking, particularly in the context of name binding. Further research is recommended to explore the possibility of implementing similar type checkers for other programming languages. Additionally, the paper suggests incorporating additional features into the targeted Scala subset, thereby enhancing its extensibility. The code is available on GitHub[1].

Key terms: Type Checking, Scala Type Checker, Scope Graphs, mini-Statix, Phased Type Checking, Monotonicity.

1 Introduction

The field of programming languages has seen significant advancements in recent years, with the focus shifting towards developing efficient type checkers that ensure the correctness of programs. In this context, scope graphs have emerged as a promising solution for handling scoping and binding in programming languages [1].

Previous research has explored the use of scope graphs for building type checkers. For example, van Anterwepen et al. [2] demonstrated the effectiveness of their approach in handling various language features, such as nested functions, imports and composite types. While alternative approaches, such as Hindley-Milner type inference [3] or type classes inference [4], exist for generating type checkers, challenges remain in declaratively implementing type checkers using scope graphs for certain language features.

Furthermore, the development of this research has been inspired by the work presented in [5], which introduces mini-Statix[2], a Haskell implementation of Statix-core [6] tailored for a Scala subset. It provides support for handling Scala-specific language features such as imports and objects, enabling the type checking of Scala programs against their declarative specifications. Declarative specifications offer concise and intuitive descriptions of desired behavior or properties, without specifying implementation details. The paper demonstrates the effective utilization of scope graphs in addressing some challenges of name resolution.

Despite recent progress in generating type checkers from declarative specifications using scope graphs, a knowledge gap remains regarding how these approaches represent name binding and scoping rules in a clear and understandable manner [7]. In this paper, we address the question ***"Can we implement a type checker for a targeted Scala subset, using scope graphs and a Haskell library for phased scope graph construction?"***. This research question pertains to unanswered questions surrounding the declarative nature and extensibility of such type checker:

i. *" How effectively does the scope graph-based phased approach capture language declarativity and represent name binding and scoping rules?"*

ii. *" Can the scope graph-based phased approach be extended to support new language features in a modular and efficient manner, and does this require additional phases?"*

To answer these research questions, we implemented our approach based on Rouvoet et al.'s mini-Statix Scala project, using a phased Haskell library[3]. Then, we performed a qualitative comparison to the mini-Statix approach, utilizing their test suite for name resolution challenges. We also evaluated the declarativity and feature extensibility of our type checker compared to theirs, highlighting the strengths and weaknesses to guide future research in this area.

In summary, the contributions of this paper are:

1. We provide a Haskell implementation of a phased type checker for a Scala subset (presented in Section 3). Through this, we address challenges related to precedence in Scala, particularly with specific and absolute imports, by carefully phasing the type checking process (presented in Section 3.4).
2. We present a comprehensive comparison with the mini-Statix implementation, utilizing their existing test suite. This comparison serves to validate and evaluate the proposed approach, as discussed in Section 4.
3. We identify opportunities for further research, such as improving rule readability, addressing precedence in name resolution, resolving ambiguity between definitions across scope levels, and exploring alternative strategies for handling name conflicts (presented in Section 8).

The rest of the paper is organized as follows. Section 2 provides a thorough description of the addressed problem, while Section 3 highlights our contributions. Section 4 presents a qualitative analysis of our type checker, and Section 5 discusses the Responsible Research behind our implementation. We further compare our approach with other related work in Section 7, while Sections 6 & 8 summarize the findings and discuss possible directions for future work.

[1] Code: https://github.com/rgmihalachiuta/ScalaScopeGraphs.git

[2] Scala mstx: https://github.com/MetaBorgCube/scala.mstx

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

2 Problem Description

In this chapter, a comprehensive description of the addressed problem is provided. Firstly, the chapter explains the background of the main concepts in 2.1, which are part of the stated objective of the research in 2.2. Then, the focus is put on a targeted Scala subset and the challenges posed by objects and imports, justifying their relevance in Section 2.3. Lastly, the concept of *monotonicity errors* is presented in 2.4, underscoring their significance in maintaining type correctness. The solution to these challenges is further explained in Section 3.

2.1 Preliminaries

Type checking

Type checkers are programs that ensure the compatibility and correctness of types in a program. They help to catch errors early in the development process and ensure program reliability through valuable feedback [8]. In the research context, a robust and accurate type checker is essential to enforce type safety, maintain the integrity of object hierarchies, and ensure the proper resolution of imported symbols.

Scope Graphs

Scope graphs are a promising approach in constructing type checkers for handling intricate name resolution rules [2]. They offer a unified view of scoping, name resolution, and visibility, accurately tracking program entity accessibility within scopes.

Scope graphs provide declarative rules for scoping structures, eliminating the need for ad hoc typing contexts (i.e. language-specific scoping mechanisms). They offer a parametric approach to name resolution, enabling consistent and reusable type checking algorithms [7].

The adoption of scope graphs in this research addresses complexities in name resolution and scoping mechanisms. It ensures correct enforcement of scoping rules, unambiguous name resolution, and resolved import conflicts.

Scope Graph Representation

Scope graphs are directed graphs where scopes are represented as nodes with attached sinks for name binding information. Type-checking involves running queries for specific names starting from a given scope and traversing edges based on regular expressions. We used the notation depicted in Figure 1, where the scope graph showcases variable and object sinks linked to their respective scopes through labeled edges. These edges represent various relationships such as lexical parents (**P**), object sink edges (**O**), definition scopes (**D**), variable sink edges (**V**), type aliases sink edges (**TY**), explicitly imported variable sink edges (**EI**), and wildcard import edges (**WI**).

```
Sink.ValDecl  ::= [([String] [Type])]
Sink.ObjDecl  ::= [([String] [Scope])]

Edge.Label    ::= P | O | D | V | TY | EI | WI
```

Figure 1: Scope Graph Parameters.

Van Antwerpen et al. [2] propose representing scopes as types to address challenges in name resolution, including imports and composite types. In contrast, the Haskell library incorporates scopes as arguments of declarations, as shown in Figure 1. The sink object declaration takes arguments of the object name and associated scope, aligning with the concept of treating scopes as types. Thus, it provides explicit scoping and improves understanding of interactions with the environment. This approach offers a declarative representation of name resolution rules, ensuring consistency across scopes and declarations.

2.2 Research Objective

The current mini-Statix implementation handles stable querying, addressing scoping and name resolution complexities, including objects and differences between explicit and wildcard imports. Ergo, the objective of this research is to develop an alternative scope graph-based type checker, combining the convenience of automated scheduling with the flexibility and control offered by the phased Haskell library.

Additionally, the research aims to explore the number of phases required for this type checker and investigate the impact of explicit phasing on the implementation. This exploration is particularly intriguing as previous work, such as mini-Statix, has handled the phasing automatically, and the effects of explicit phasing remain unknown. By addressing these aspects, the study seeks to expand the current understanding in programming languages and type systems, specifically within the context of Scala.

2.3 Targeted Scala Subset

Due to time constraints and the specific nature of the research question, only the mini-Statix Scala subset was considered for development and evaluation. A part of the selected subset can be visualised in Figure 2, and it includes essential language features related to definitions, objects and import mechanisms.

```
ScProg.Prog        ::= [([ScDecl*])]

ScDecl.Val         ::= [([ScPr] [ScExp])]
ScDecl.TypeAlias   ::= [([String] [Type])]
ScDecl.Def         ::= [([String] [ScPr*]
                          [Type] [Body])]
ScDecl.Obj         ::= [([String] [ScDecl*])]
ScDecl.Imp         ::= [([ScImp])]

ScImp.ExplicitImp  ::= [([String*] [String])]
ScImp.WildcardImp  ::= [([String*]]

ScPr.Parameter     ::= [([String] [Type])]
Body.DefBody       ::= [([ScDecl*] [ScExpr])]

ScExpr.Id          ::= ID
ScExpr.Num         ::= INT
ScExpr.Boolean     ::= BOOL
```

Figure 2: Context-free Syntax for the Scala Subset.

Name Resolution in Scala Objects
Objects in Scala provide a powerful abstraction mechanism for encapsulating state and behavior. However, their interaction with scoping rules or member access introduces complexities in the type checking process. In Scala, scoping rules vary for names defined in the lexical scope (which can be forward referenced) and imported names (which cannot), resulting in more complex resolution and disambiguation rules [5]. Figure 3a (with its scope graph representation in Figure 3b) exemplifies two such possible complexities. It comprises of a forward-reference case, where `x` depends on the type of `y`, which is declared later in the program, and a case where the inner block of an object method shadows the outer scope. By targeting this subset, the goal is to handle object declarations and ensure proper scoping and name resolution within the object context.

(a) Program

```
object A {
  val x: Int = y
  val y: Int = 42

  def m : Boolean = {
    val x: Boolean = true
    x
  }
}
```

(b) Scope graph

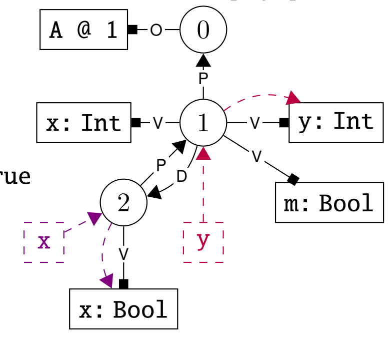

Figure 3: Visualisation of challenges imposed by Scala Objects. A case of forward-reference and block-shadowing.

The process of querying in scope graphs involves examining paths that adhere to specific edges and arranging them based on their shortest path. Regular expressions are used to define the permissible edges that can be traversed. In Figure 3b, queries are represented by purple and violet lines. For instance, when resolving the reference to `x`, a query is initiated from scope 2, searching for all sinks whose path labels match the regular expression **P*D*WI?(EI|V)**. This query yields a single path, depicted in violet. In cases where multiple paths exist, the shortest path is chosen. If it is not feasible to establish a path, the program becomes ambiguous.

Imports Precedence
In Scala, explicit and wildcard imports have different precedence rules based on whether names are explicitly listed or caught by a wildcard, allowing developers to selectively import specific members or import all members from a module [5]. The difference in precedence between these import mechanisms creates an interesting challenge in name resolution. When a name is referenced, conflicts, called name clashes, may arise between explicitly imported members and wildcard-imported members with the same name. This can be observed in Figure 4, where both *A & B* define a variable *x*, imported by *C*. Resolving these conflicts requires defining rules that prioritize one type of import over the other. These rules will be further explained in Section 3.2.

```
object A {
    val x : Int = 21
}

object B {
    val x : Boolean = true
}

object C {
    import A.x
    import B._

    val y : Int = x /** queries to A == 21
}
```

Figure 4: Name-clash between specific and absolute imports.

2.4 The Challenge of Monotonicity

Monotonicity ensures that query results remain valid throughout the entire type-checking process (i.e. query stability). In the context of scope graph queries, being monotone means that once a scope has been queried for a path with a specific label, an outgoing edge with the same label cannot be added later. These edges are called *critical edges* in literature [5] and they can invalidate the stability of queries. Hence, these errors occur when the addition of critical edges influences previous queries, leading to inconsistent results upon re-evaluation.

In Figure 5a (scope graph representation in Figure 5), querying for `A` before the second explicit import would initially resolve `A` from scope 1. However, when the scope graph is fully constructed, querying for label **EI** in the same scope would violate monotonicity, introducing a crticial edge. This ambiguity highlights that an explicit import does not effectively shadow the earlier one. To maintain monotonicity, only one explicit import should be used. We describe how our approach handles these challenges imposed by monotonicity in Section 3.2.

(a) Program

```
object O {
  type A = Boolean
  object M {
    type A = Int
  }

  object I {
    import O.A
    val x : A = 3
    import M.A
  }
}
```

(b) Scope graph

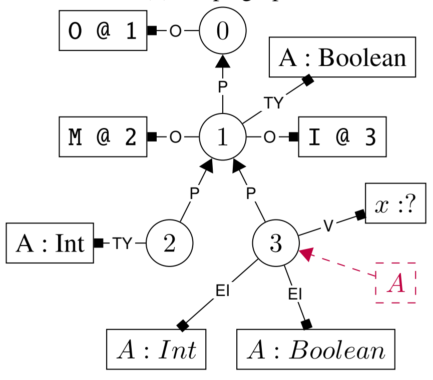

Figure 5: A program and its associated scope graph that lead to monotonicity errors.

3 Phased Type Checking

In this chapter, we present the main contribution of this thesis: the phased type checker. We start by discussing the technical implementation, utilizing the Haskell library, in Section 3.1. Next, in Section 3.2, we outline the phases of the type checker. To provide even more depth, we present another comprehensive example in Section 3.3. Finally, we reflect upon the encountered challenges in Section 3.4. This foundation sets the stage for the subsequent analysis and evaluation of the type checking approach, which we present in Section 4.

3.1 Technical Implementation

The type checker is implemented using Haskell, leveraging a range of tools and libraries to facilitate its development. Additionally, we have used a phased Haskell library which offers a functional approach to build and inquire about scope graphs using the principles of effect handler theory introduced by Bach Poulsen and van der Rest in their work [9].

The fundamental resources were represented by the main primitives of the Haskell library: **`new`**, **`edge`**, **`sink`**, and **`query`** for creating efficient and correct scope graphs. These primitives enable the creation of scopes, establishment of scope relationships, storage of data, and efficient querying within the type checking process. The specific implementation of these primitives can be found on GitHub[4].

3.2 Phased Algorithm

To address the challenges outlined in Section 2, we developed a phased type checker. While mini-Statix automatically handles query scheduling, tracking critical edges and delaying queries until all edges are constructed [5], the Haskell library takes a different approach. It explicitly manages the phasing, providing fine-grained control over the order of operations to prevent monotonicity errors. This enables the type checker to operate in distinct phases, each targeting a specific aspect of the program. Hence, the core algorithm of the type checker consists of the following main phases, which will be exemplified through the scope graph in Figure 6.

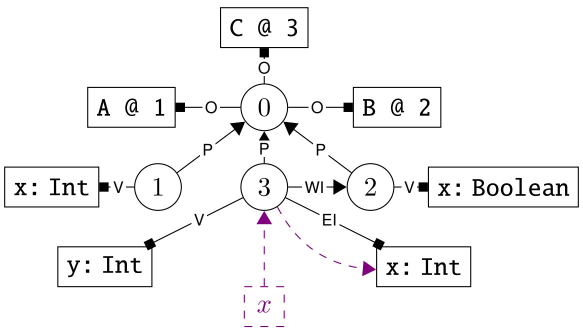

Figure 6: Scope graph representation of the name-clash in Figure 4.

[4]Primitives Implementation: https://github.com/heft-lang/hmg/blob/master/src/Free/Scope.hs

1. Establishing the parent object sinks along with their corresponding scopes.
2. Incorporating variable declarations into the graph.
3. Handling imports by including the relevant sinks and edges in the graph.
4. Type-checking of declaration bodies in relation to the scope graph.

Phase 1: Allocating Object Scopes
The initial step involves traversing the program, encompassing parent objects with the potential for nested objects, declarations, and imports. By commencing with a global import denoted as *0*, we capture all parent objects such as *A, B, C*. Subsequently, a dedicated scope is allocated for each parent object (*1, 2, 3*), establishing a coherent connection between the global scope and the corresponding object sinks. This intricate process draws parallels to the workings of mini-Statix in terms of object declaration methodology.

Phase 2: Declaring Variables
Moving on, we proceed to declare variables within each object, ensuring their encapsulation within the corresponding scopes. In our example, we declare the two instances of variable *x*, as well the one of *y*. By prioritizing the construction of declarations, we facilitate the possibility of forward referencing and import accessibility. This ordering is crucial, as exemplified by the inability to copy the explicitly imported name without the prior declaration of variable *x*.

Declarations encompass a range of elements, including definitions, type aliases, and child objects. This implementation closely aligns with the specifications of mini-Statix, where the declaration of variables follows a similar approach. For values and types, we introduce a *V* or *TY* sink (labels *VAL & TYPE* in mini-Statix) within the associated scope. In the case of definitions, we add a *V* sink to the corresponding scope and establish a new scope for the parameters and body of the definition. To establish the connection between the parent scope and the definition scope, we introduce a *D* edge, a step not performed in mini-Statix. The treatment of child objects mirrors that of parent objects, following a similar procedure.

Phase 3: Import Resolution
One notable distinction between our approach and mini-Statix lies in the handling of imports. In mini-Statix, imports are defined as a sequence of scopes connected by the label *B*, which is passed alongside the current lexical scope. Consequently, name resolution occurs within the appropriate scope. Conversely, in our implementation, we handle specific imports by copying the explicitly imported names within the scope of the importing object. This deviation from mini-Statix simplifies the process of specific imports but eliminates the concept of sequenced imports, which carries certain drawbacks discussed in detail in Section 4. Additionally, when dealing with wildcard imports, we establish *WI* edges between the scopes of the importing and imported objects, which reflects the behavior in mini-Statix.

Illustrating this with our example, upon encountering the specific import of object *A*, we query for object *A* and check if the variable x exists within its scope. Upon confirming its presence, we proceed to copy x using an *EI* edge within the imported scope of object C. Similarly, when importing object B, which is a wildcard import, we simply establish a *WI* edge from scope *3* to scope *2*.

Phase 4: Type-checking the Program

Finally, we proceed to type check the bodies of all declarations. During the type checking process, when encountering an identifier, we perform a query that encompasses both the current object and the imported objects, utilizing the resolution regex pattern: **P*D*WI?(EI|V)**. It is worth noting that in mini-Statix, the regex pattern **B*(PB*)*(I|W)?VAL** is employed for variable resolution. The main distinctions lie in the block atoms, as our approach does not support them, and the representation of explicit imports as sinks rather than edges. Consequently, our regex pattern provides the flexibility to select paths between values and explicitly imported names.

In our example, by selecting the shortest path, the correct resolution of the x is ensured. The type checker prioritizes the explicit import A over the wildcard import B (i.e. violet path), even though it is farther away, adhering to Scala's preference for explicit imports. As a result, the value of 21 from 'A.x' is appropriately chosen, mitigating any potential ambiguity or erroneous outcomes that could arise from name clashes.

3.3 Handling Name Resolution

Our techniques and design choices ensure proper scoping, member access, and resolution rules within objects, while innovative strategies address conflicts and maintain desired scoping semantics for different import mechanisms. In order to illustrate the effectiveness of the phased algorithm in handling these complex name resolution scenarios, this section provides one more comprehensive example. This example highlights the advantages of the phased approach in ensuring proper name resolution and avoiding conflicts, ultimately contributing to the robustness and reliability of the type checking process.

(a) Program

```
object O {
  def f : Int = g
  import N._
  def g : Int = h
}

object N {
  import O.f
  def h : Int = f
}
```

(b) Scope graph

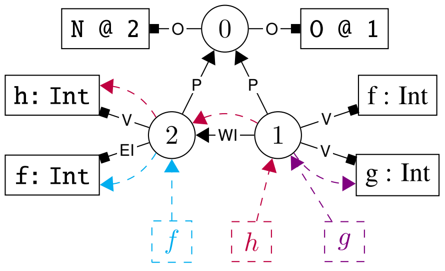

Figure 7: A program and its associated scope graph to exemplify the phased type checker.

Figure 7a and its corresponding scope graph in 7b exemplify the challenge of handling mutually recursive definitions with imports, specifically in the context of circular dependencies between objects O and N. Such circular dependencies can introduce monotonicity errors in name resolution and type checking. In Scala, the semantics allow for mutually recursive definitions within an object, enabling forward references, while imports follow sequential rules, restricting references to imported names until after the import statements [5].

The phased type checking process addresses this challenge by sequentially creating the scope graph and declaring objects O and N in the first phase, without performing name resolution. Subsequently, the definitions are declared, followed by the processing of imports within each object. In the scope graph, the absolute import in object O is represented by a *WI* edge, establishing a link between the object scopes, allowing complete access to object N's scope from object O. Conversely, the explicit import $O.f$ is reflected by copying the named import f into the scope of object N. Finally, the definitions are type-checked, leveraging the resolved imports and mitigating the risk of monotonicity errors, as depicted by the colored paths in Figure 7.

3.4 Reflection upon Encountered Challenges

The phased nature of the type checker poses challenges in coordinating and sequencing the different phases, especially in complex programs with interdependent components. Coordinating the order of execution and ensuring the correct propagation of information across phases can become intricate. Additionally, changes in one phase may necessitate corresponding adjustments in subsequent phases, adding to the complexity of maintaining the overall integrity of the algorithm. However, these challenges are addressed through a rigorous implementation, which carefully handles the dependency between each phase.

Moreover, the phased approach effectively addresses the challenge of monotonicity errors. By carefully ordering the operations, critical edges, which can disrupt query stability, are introduced only after processing all relevant declarations and imports. This preserves the formal property of query stability and ensures consistent and reliable type checking results.

Finally, the mini-Statix approach encounters challenges related to the complexity of scope graph-based type checking rules, precedence conflicts, and ambiguity caused by shadowing. These challenges require careful consideration and effective handling. Our phased approach offers a potential solution by systematically addressing these complexities, ensuring clearer and more robust resolution of name bindings and scoping rules. Coordinating the sequencing and interactions between the phases mitigates rule complexity, precedence conflicts, and shadowing ambiguity, resulting in more accurate and effective type checking.

4 Experimental Setup and Results

In this section, the evaluation setup (4.1) and procedure (4.2) used are described to assess the effectiveness and performance of the phased type checker. This is based on a compare and contrast between the phased approach and the existing type checker in the Statix library, whose results are analysed and reflected upon in 4.3.

4.1 Evaluation Setup

To comprehensively evaluate the type checker, we constructed a diverse test suite based on the existing Scala programs from the mini-Statix implementation's test suite[5]. The mini-Statix test suite consists of 108 well-crafted test cases designed to specifically address and assess the challenges of name resolution, object structures, and various import scenarios. Additionally, we expanded the test suite by incorporating new tests that cover edge cases and more intricate name resolution scenarios, including the program examples highlighted in this paper. As a result, the test suite grew to a total of 113 tests, as depicted in Figure 8. This augmented test suite formed the foundation for comparing the performance and correctness of our type checker against mini-Statix's type checker.

Due to time constraints, our approach prioritized core functionalities and omitted support for import renaming, hiding, and nested lexical scopes in definition bodies, resulting in 16 unsupported test cases. While these cases are excluded from evaluation, they serve as potential areas for future optimization. The subsequent subsection focuses on the remaining five unsupported test cases, enabling us to concentrate on evaluating supported features and identifying opportunities for enhancement.

Figure 8: Test suite overview.

Property	# Test Cases	# Unsupported Tests
Comprehensive	4	2
Definitions	18	-
Expressions	11	2
Imports	25	9
Precedence	35	7
References	7	1
Statements	8	-
New tests	5	-
Total	**113**	**21**

4.2 Evaluation Procedure

The test suite (comprising of the remaining 92 tests with supported features) was systematically executed to evaluate the performance of our phased type checker. The analysis of the results involved assessing the number of true positives, true negatives, false positives, and false negatives. True positives are tests that pass in both mini-Statix and our approach, indicating consistent and correct behavior. True negatives are tests that fail in both implementations, indicating agreement on identifying errors. False positives are tests that fail in mini-Statix but pass in our approach, suggesting improved accuracy. False negatives are tests that pass in mini-Statix but fail in our approach, highlighting areas for further refinement. These findings are presented in Figure 9, providing a visual representation of the performance metrics.

[5]Test suite: https://github.com/MetaBorgCube/scala.mstx/tree/master/tests

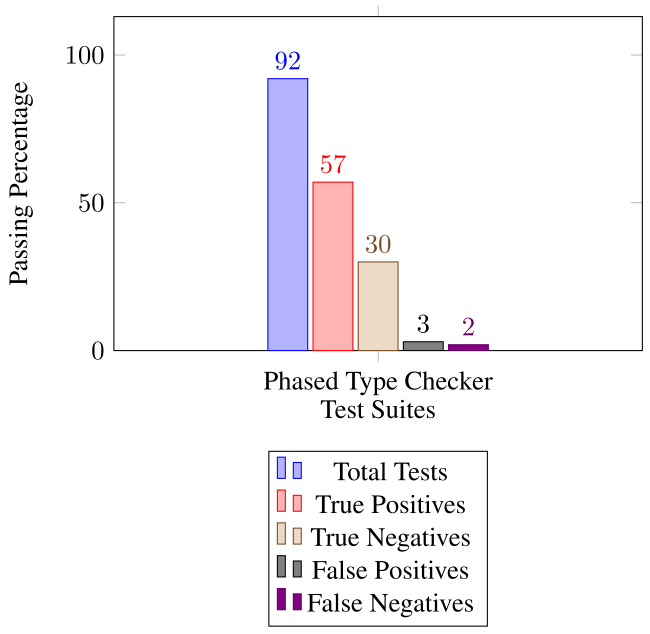

Figure 9: Bar graph highlighting results.

4.3 Results

The bar graph in Figure 9 highlights the great performance of our approach compared to mini-Statix across 92 tests. With an accuracy rate of 87 out of 92 tests, our approach demonstrates a high level of reliability and precision. This accuracy is crucial as it minimizes the chances of name conflicts, ambiguity, and incorrect type assignments, resulting in improved program quality, efficient debugging, and increased productivity.

It should be noted that the false positives are due to the lack of support for sequenced imports in our approach, as discussed in Section 3.2, resulting in two test failures. Additionally, there is one false positive and two false negatives observed in specific nested tests that examine wildcard shadowing. While time constraints prevented the resolution of these issues, addressing them would be a valuable consideration for future work.

To resolve the lack of support for sequenced imports, we can enhance the type checker by incorporating sequential import processing, like mini-Statix does. In addition, to address the false positives and false negatives related to wildcard shadowing, we can refine the scoping rules and name resolution algorithm. This may involve introducing advanced mechanisms to handle conflicts and ambiguity caused by wildcard imports, such as prioritizing local declarations or providing disambiguation options.

The comparison between the approach and Statix revealed a notable distinction in the phased nature of the type checker, offering explicit control and advantages in declarability and extensibility. The approach's modularization and separation of concerns allow for easier reasoning and extension. In contrast, mini-Statix's internal query scheduling limits control and presents implementation challenges.

5 Responsible Research

Responsible research is crucial and highly relevant in the field of type checking. As researchers and developers, it is our ethical obligation to ensure that our work not only produces valuable insights and advancements but also considers the broader impact on society. Responsible research encompasses various aspects such as validity, reliability, transparency, reproducibility, and ethical considerations. By adopting a responsible approach, we can address potential challenges, mitigate risks, and provide reliable and trustworthy outcomes that contribute to the well-being and progress of the software development community and its practical applications.

First and foremost, the validity and reliability of the type checking process have been thoroughly investigated to mitigate the occurrence of false negatives and false positives. These inaccuracies can have detrimental effects on program behavior and error detection. It is important to note that while our approach has demonstrated high accuracy, a few false positives and false negatives have been observed. As responsible researchers, we acknowledge these instances and recognize the need for further refinement. Future work will focus on addressing these false positives and negatives through continuous experimentation, validation, and algorithmic improvements to enhance the overall quality and precision of the type checking results.

Moreover, accurate and informative error messages play a crucial role in responsible research and software development. Clear and precise error messages aid developers in understanding and resolving issues in their code, leading to more efficient debugging and improved program quality. By providing detailed explanations of type errors, ambiguous references, or scoping conflicts, developers can quickly identify and address potential issues, resulting in more reliable and robust software. Moreover, user-friendly error messages enhance the usability and accessibility of the type checker, empowering developers of all levels of expertise to effectively utilize the tool and promote responsible programming practices.

In terms of ethical considerations, transparency has been a key focus throughout the research. Both the research methodology and the code implementation have been made transparent to promote openness and reproducibility. The availability of the code on GitHub facilitates the accessibility of the test suite, enabling others to verify and reproduce the experiments. This transparency helps establish a reliable foundation for further research and ensures the accountability of the findings.

Additionally, ethical considerations extend to the potential consequences of the research on practical life. Recognizing the trade-off between expressiveness and complexity in type checking with scope graphs, recommendations have been provided to mitigate the risks associated with the misuse or misinterpretation of the research outcomes. By emphasizing responsible usage and providing guidelines for proper application, the research aims to promote the responsible adoption and utilization of the developed code.

It is important to note that the theory of separatism [10], which suggests that researchers are only responsible for the development of the technology and not for how it is used by others, is inadequate. Instead, a comprehensive approach is advocated, in which researchers actively engage in both the development and the implications of how the technology is utilized. By acknowledging the ethical dimensions and taking proactive measures, researchers can contribute to the responsible and beneficial use of the developed tools.

6 Discussion

This discussion revolves around the two sub-research questions regarding the declarativity and feature extensibility of the phased approach. They aim to provide a comprehensive exploration of the strengths, limitations, and potential challenges associated with our approach. By delving into these discussions, we gain a deeper understanding of the trade-offs involved and the implications for practical implementation and future development.

Based on the findings presented in this paper, it can be concluded that our approach, while effective and promising, is not as declarative as mini-Statix. The mini-Statix implementation relies on a rule-based approach for type checking, providing a high level of declarativity. In contrast, our approach adopts a phased nature that emphasizes explicit steps and sequencing. While this may introduce a departure from strict declarative rules, our approach compensates for this by offering greater control, extensibility, and modularity. The trade-off between declarativity and flexibility is an important consideration when choosing a type checking approach, and our findings highlight the strengths and advantages of our phased approach.

The extensibility of scope graphs as a foundation for implementing a type checker in the Scala subset was also explored during the evaluation process. While scope graphs offer a solid basis for name resolution and scoping rules, certain language features or scenarios may require additional effort to handle effectively. However, the extensibility of the approach provides ample room for addressing these limitations through careful design choices and further research. By incorporating new language features, refining the resolution algorithm, and enhancing error reporting, the type checker can be extended to accommodate a wider range of programming constructs and provide more accurate and informative feedback to developers. This focus on extensibility ensures that the type checker remains adaptable to evolving language requirements and continues to provide valuable support for developers in their programming endeavors.

The decision to introduce new phases in the type checking process for handling new features depends on the complexity and requirements of the features. While some features can be accommodated within existing phases, more complex ones may benefit from separate phases to ensure accurate type checking and maintain code organization. The choice should prioritize maintainability, clarity, and modularity, striking a balance between accommodating new features and preserving the simplicity of the overall type checking algorithm. By evaluating each feature's needs, a well-designed type checking process can effectively handle existing and future language constructs.

7 Related Work

The related work section plays a crucial role in contextualizing our approach within the existing literature and highlighting its unique contributions. It allows us to critically evaluate and compare our approach with relevant papers, gaining valuable insights into the strengths, limitations, and potential advancements in the field of type checking.

In the realm of name resolution, the use of environments as a means of expressing name resolution concepts, such as in Scala, has been common. However, Rouvoet et al. [5] argue that environments can be limiting in terms of high-level expressiveness. Scope graphs offer a novel and more flexible approach, capturing the complex interplay between nested scopes, imports, and lexical scoping rules. Compared to environments, scope graphs provide advantages in terms of precision, modularity, and control over name resolution. While previous works have predominantly focused on environments [11], the adoption of scope graphs opens up new possibilities for advanced and reliable type checking algorithms. Our phased approach leverages the benefits of scope graphs, offering a more declarative, modular, and extensible type checking process.

Mini-Statix [5] is a notable implementation in type checking with scope graphs. It provides a language for declarative typing rules and allows for the automatic derivation of executable type checkers from these rules. In the context of name resolution, the mini-Statix language employs scope graphs to handle the complexities of name binding. This approach offers advantages over manual type checkers by ensuring soundness and providing a declarative way to define typing rules. However, challenges remain in implementing certain language features declaratively using mini-Statix and scope graphs, necessitating further exploration and refinement.

In comparison, our phased approach emphasizes explicit control and sequencing in the type checking process, offering greater flexibility, modularity, and extensibility. While mini-Statix provides automated query scheduling and ease of use, our approach grants developers more control over execution order and avoids dependencies associated with specific language workbenches, reducing the learning curve. The trade-offs between automation and control, as well as the simplicity of dependencies, influence the suitability of each approach for different development scenarios.

8 Conclusion

To conclude, this paper successfully addresses the research questions of whether a type checker for a targeted Scala subset can be implemented using scope graphs and the Haskell library, and how effectively the approach captures language declarativity and handles name binding and scoping rules. The phased Haskell library enables the implementation of the type checker with four phases, striking a balance between declarativity and extensibility. While some declarative aspects are compromised, the approach compensates by offering enhanced modularity and the ability to support new language features efficiently through additional phases. This research contributes to the field of type checking by demonstrating the practicality and advantages of the scope graph-based phased approach, paving the way for further advancements in reliable and flexible software development processes.

The implementation of the type checker establishes a solid foundation for future extensions and improvements. Addressing the unsupported test cases to achieve full coverage and incorporating the missing functionality from mini-Statix are important avenues for enhancing the accuracy, capabilities, and usability of the type checker. These advancements will align the tool with existing declarative approaches, ensuring reliable and flexible support for developers working with the Scala subset. Furthermore, improving error reporting and integrating external tools for advanced functionality will enrich the capabilities, efficiency, and overall usability of the type checker, further empowering developers in their programming endeavors.

A Acknowledgements

I would like to express my gratitude to my supervisors and peers for their guidance and support throughout the research process. Additionally, I acknowledge the assistance of ChatGPT in ensuring grammatical accuracy and refining complex paragraphs, contributing to the clarity and coherence of this paper.

References

[1] Eelco Visser. Scope graphs: A fresh look at name binding in programming languages. In *Conference talk at Curry On 2017 (ECOOP 2017) in Barcelona*, June 2017.

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

[3] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[4] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.

[5] 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. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[6] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 49–60, 2016.

[7] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *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, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

[8] Luca Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1):263–264, 1996.

[9] Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proceedings of the ACM on Programming Languages*, 7:62, 2021. pp 1801–1831.

[10] I.R. Poel, van de and L.M.M. Royakkers. *Ethics, technology, and engineering : an introduction*. Wiley-Blackwell, United States, 2011.

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.