Building Type Checkers using Scope Graphs For a Language with Type Classes

Andreea Mocanu[1]

Supervisor(s): Casper Bach Poulsen[1]**, Aron Zwaan**[1]

[1]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: Andreea Mocanu
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Aron Zwaan, Thomas Durieux

Abstract

In this paper, we explore scope graphs as a formal model for constructing type checkers for programming languages that support type classes. Type classes provide a powerful mechanism for ad hoc-polymorphism and code reuse. Nevertheless, the incorporation of type classes into type checkers poses challenges, as it necessitates the resolution of instances and the assurance of coherence amidst overlapping instances. Our approach facilitates the separation of concerns between type class resolution and type checking, promoting extensibility and maintainability of the type checker. We contribute with a formal definition of scope graphs for languages with type classes, accompanied by algorithms for type class resolution and type checking. To assess the correctness of this approach, we implement a prototype type checker, and conduct experiments on a collection of representative programs. The results demonstrate the effectiveness of this baseline approach.

Keywords: scope graphs, type checkers, type classes, ad-hoc polymorphism.

1 Introduction

Type checking allows software engineers to catch bugs in their code early in the development process. One particularly challenging aspect of implementing type checkers is name binding (associating references with the appropriate declarations). Scope graphs can be particularly useful in this case, as these provide a model for declaratively specifying the semantics of name binding structures in programming languages. This data structure allows defining declarative typing rules, similar to the ones found in programming language literature. However, traditional typing rules rely on typing contexts defined on a language by language basis for certain scoping structures. In contrast, scope graphs offer a uniform representation of name binding in a language parametric manner. Therefore, by using scope graphs as a model, algorithms that resolve names during type checking in a consistent manner can be implemented, independent of language.

When it comes to languages with support for type classes, it is not yet well understood how to implement type checkers using scope graphs. A type class can be defined as a family of types that implement a common set of functions. This feature of programming languages enables ad hoc polymorphism. Type classes were introduced in [13], as a new approach to ad hoc polymorphism. The same paper explains ad hoc polymorphism, named by Strachey in [10], as "occuring when a function is defined over several different types, acting in a different way for each type." Work in the area of building type checkers for languages with type classes include a set of type inference rules for resolving overloading introduced by type classes [1], Mark Jones' type system of constructor classes [3] and the type inference system [7] proposed by Tobias Nipkow and Christian Prehofer.

Regarding the problem of building type checkers using scope graphs, this has been achieved for a subset of Java, Scala and a language with modules and records as a case study for [9]. Other relevant work is introducing scopes as types in [11], which uses scope graphs to model type systems with more sophisticated form of type representation. However, both these papers use Statix[1], which does the internal scheduling of queries to resolve name resolution. This offers less flexibility than the provided Haskell library used in this paper, where the task of scheduling queries is done explicitly.

The research question that this paper aims to answer is then the following: *"How can we implement a type checker for a language with support for type classes, using scope graphs?"*, combining aforementioned concepts. This paper aims to answer this question by providing the following list of contributions:

- We propose a novel approach for building type checkers for type classes using scope graphs.
- We provide a practical implementation of our scope graph-based type checker, demonstrating how the proposed approach can handle type class resolution and type inference accurately.
- We develop a test suite use for evaluation, consisting of diverse Haskell program that cover challenges of type checking type classes.
- We provide insights into the design and implementation choices for handling type classes. We analyze the benefits and limitations of our scope graph-based approach compared to existing ones, highlighting its potential for improving type checking for type classes.

In the subsequent sections of this research paper, a comprehensive exploration of the research questions at hand is provided. Section 2 expands on the problem and how scope graphs and type classes are integrated. In Section 3, we highlight the novel aspects of the proposed solution and demonstrate how the approach works for some example programs. Section 4 presents the methods used to assess the proposed implementation and presents the results. We compare the declarativity and extensibility of the type checker to existing approaches and cover the limitations, as well as potential areas for improvement, in Section 6, while Section 5 addresses ethical considerations. Related work is discussed more in-depth in Section 7. Finally, we summarize the key contributions and outline directions for further research in Section 8.

2 Understanding Type Classes

This section presents a comprehensive overview of the problem addressed in this research, focusing on the task of building type checkers using scope graphs for languages with support for type classes. We adopt a problem-oriented perspective, aiming to outline the relevant background concepts, theory, and models that form the foundation of this research. The goal is to establish the necessary context and formalize the problem under investigation.

[1] Constraint-based declarative language for the specification of type systems that combines type constraints with name resolution constraints based on scope graphs.

To begin, an introduction is provided to the concept of type checkers and their role in programming language analysis. Next, the concept of type classes and their relevance in programming languages is explored, highlighting their role in enabling polymorphism and code reuse. Furthermore, we explain the significance of scope graphs in facilitating precise type checking and name resolution, as well as the representation of scope graphs used in this paper.

Type checkers are fundamental tools in programming language analysis. They play a crucial role in ensuring the correctness and reliability of programs by verifying the compatibility of data types used within a program. Type checkers examine the expressions, variables, and functions in a program and ensure that the operations performed on them are consistent and well-defined. By catching type-related errors early, type checkers help prevent runtime crashes, improve program reliability, and facilitate efficient debugging. It is considered that a program that type checks correctly "cannot go wrong" [6].

Type classes are a concept in programming languages that enable ad hoc polymorphism and code reuse. These provide a mechanism for defining behavior that can be shared across different data types. By defining type class instances, programmers can specify how different types implement the functions defined by the type class. This promotes generic programming and allows code to be written in a more abstract and reusable manner. Thus, type classes offer a powerful way to abstract over different data types, while providing a unified interface for interacting with them.

One particularly challenging aspect for type checkers is name binding. This is due to the complex nature of scoping rules and the resolution of names within a program. Type checkers need to accurately determine the associations between names and their corresponding units, such as variables, functions, or types. This involves traversing the program's scope hierarchy, considering factors such as lexical scoping and nested scopes. However, the presence of shadowing, aliasing, and dynamic scoping can introduce ambiguity and make name binding more intricate. Additionally, type checkers often need to handle language-specific features such as modules, object-oriented constructs, or type classes, further complicating the task of accurately binding names to entities. Achieving robust and efficient name binding in the context of type checking requires careful consideration of language semantics and the development of sophisticated algorithms and data structures to navigate the complexities of scoping and name resolution.

2.1 Scope Graphs

Scope graphs, first defined in [8], are data structures used in programming language analysis, specifically in the context of building type checkers. These can be used as a model to conduct and represent the results of name resolution. They capture the structural and lexical information of programs, representing the scope hierarchy, variable bindings, and name resolution relationships. In the context of scope graphs, name resolution is solved by merely finding a valid path from references to the corresponding declarations.

Generally, the following types of nodes are defined in scope graphs:

- scope nodes, which represent a scope;
- declaration nodes;
- references.

There can be labeled edges between them, such as those defined in Fig. 1. The label *D* is used for edges that go from scope nodes to declarations, whereas *P* is used to denote a *parent* relationship between two nodes. An edge from a reference node to a scope node does not have a label, as it is only an edge used for searching through the scope graph to find a matching declaration. The process of searching through a scope graph for a declaration is called *querying*.

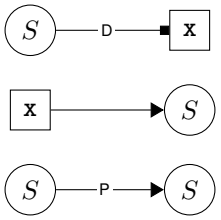

Figure 1: Legend for scope graphs.

To illustrate how scope graphs are created, we take a look at the simple example program and its scope graph depicted in Figure 2. In this case, there are two references that need to be resolved to declarations: `a` and `f`.

```
1 f a = a
2   where
3     a = 4
4
5 f 5
```

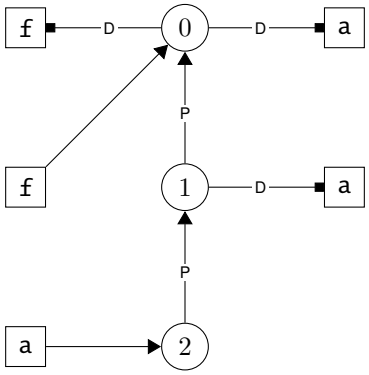

Figure 2: Example program and scope graph.

We can observe that `f` will be resolved in the global scope 0, but for `a` there are two matching declarations: one in the global scope, 0, and one in the scope introduced by the `where` block, scope 1. This program should resolve to 4, as the `a` introduced in the `where`-block will shadow the one given as argument. To resolve this correctly, the shortest path matching the regular expression `P*D` is chosen. This means it can

traverse zero or more *P* edges and one *D* edge. The scope introduced by the `where`-block is represented by 1 in Figure 2, and is closest to scope 2, which is where the reference for `a` will be resolved. This simple example illustrates that scope graphs are an intuitive model when it comes to name binding and name resolution.

2.2 Phased Haskell Library

For building type checkers using scope graphs, there are two viable options: implementing the type checker using Statix, or using the provided Haskell library [2]. After careful consideration, it was determined that the Haskell library offers more flexibility, as it allows for manual scheduling of queries. In the case of Statix, the internal scheduling posed a risk at actually implementing the feature. Therefore, our approach uses the scope graph implementation provided by the Haskell library.

2.3 Syntax

To provide a concrete implementation for the type checker, we first have to define a concrete language that the type checker will run on. For this, the syntax for a mini-language with support for type classes was defined. This definition specified the necessary constructs for type class declarations, instance declarations, and function calls with type class constraints. The syntax for this language can be found in Figure 3.

Type classes	C	
Type variables	α	
Types	$ty = Num$	numbers
	$\mid Bool$	booleans
	$\mid ty \rightarrow ty$	function type
	$\mid C \Rightarrow \alpha$	type variable with constraint
Expressions	$e = x$	identifier
	$\mid num$	numbers
	$\mid bool$	booleans
	$\mid \oplus\ e\ e$	addition
	$\mid (e\ e)$	function application
	$\mid \lambda x.e$	function abstraction
	$\mid$ let $x = e$ in e	let binding
Declarations	d = class $C\ \alpha$ where d	type class
	$\mid$ instance $C\ ty$ where d	instance declaration
	$\mid$ x : $((x : ty) \rightarrow ty)\ e$	function declaration
Programs	$p = [d]$	list of declarations

Figure 3: Syntax of the mini-language.

3 Type Checking Type Classes

This section lays forth the contribution of this research in the field of computer science. More specifically, we present the implementation of the type checking algorithm for a language with type classes, using scope graphs.

Scope graphs can easily be constructed to support type classes. A simple example of a program with type classes and its corresponding scope graph can be found in Figure 4. We will first highlight the new things that are added on top of a scope graph for a simple program, such as that in Figure 2, to support type classes, and then we will illustrate, through examples, how this construction of the graph is useful and why the changes were necessary.

Our algorithm for type checking type classes runs in two phases: creating the scope graph with the necessary declarations, as in Figure 2, and then type checking said declarations on the complete graph. During the first phase, we add new labels for the edges, *TC* and *I*, which introduce new scopes in programs. The reason for new labels is to allow following more specific paths when querying. Another addition to the general scope graph constructed previously (Figure 2) is that once we create a declaration for a type class, we also add an instance for its type variable, `a` in line 1, which is useful later on to resolve type constraints.

During the second phase of the type checker, we start by verifying that the instance declarations are compatible with the corresponding classes. Here we check that the type signatures match, and if they do, we move on to resolving the references.

```
1 class A a where
2     f :: a -> Bool
3
4 instance A Int where
5     f :: Int -> Bool
6     f x = True
7
8 foo :: Int -> Bool
9 foo x = f x
```

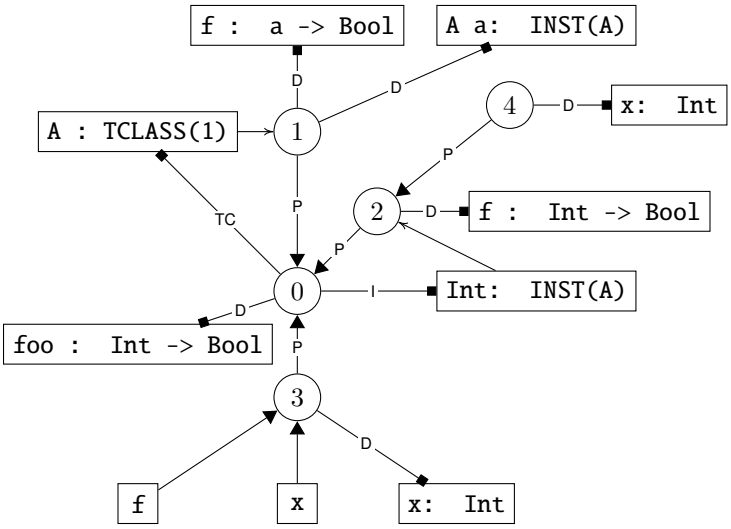

Figure 4: Example program with type classes and its scope graph.

In the example program, we first try to resolve the reference for `f`, which we then apply to `x`. We will look through the paths that match `P*(TC)?D`, as function declarations can be found within type classes as well. This means it can traverse zero or more parent edges, followed by zero or one type class edge and, finally, it must match on a declaration. Func-

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

tion `f` gets resolved to declaration `f : a → Bool`, through the path $3 \rightarrow P \rightarrow 0 \rightarrow TC \rightarrow D$. We encounter a type variable in the declaration, namely `a`. Moreover, since we matched to a declaration within a type class, `a` needs to be an instance of class `A`. We model this as a type constraint on variable `a`. This constraint needs to be resolved, so we will look for evidence that the argument type `f` is applied on satisfies it. We infer from the scope graph that the type of `x` is `Int`, thus what is left is to check that `Int` is an instance of `A`.

To resolve an instance to the correct type class, we will query paths with `P*I?D`. Furthermore, we check whether one of the instances matches the type class constraint the argument should adhere to, as well as the argument type. Through the path $3 \rightarrow 0 \rightarrow I \rightarrow D$, we find the declaration we needed, namely that `Int`, our argument type, is an instance of class `A`. Thus, the type constraint is resolved, and the program type checks correctly.

Now we can consider how we type check a program that has a type constraint explicitly in a function declaration, i.e. `foo: A a => a → Bool`. In this case, the type checker considers the parameter to be of type *type variable A*, keeping track of which constraint it can resolve through it. If we consider the same example illustrated above, with the method signature for `foo` changed, our algorithm will find the same declaration for `f`, inside type class `A`. This is where the algorithm tries to resolve the type constraint of matching to a declaration within a type class. For consistency with the approach when `x` is a concrete type and an instance provides evidence to resolve the type constraint, the scope graphs has an instance declaration for type variable with constraint A. This directly solves the type constraint on `x`.

With the addition of type variables as instances of type classes, one might wonder whether this leads to overlapping instances. Namely, if we define an instance of class `A` for a type variable, we have two matching declarations for `A a: INST(A)`: one comes from the class declarations, and the other from the instance declaration. To mitigate this, the algorithm first tries to resolve the type constraint through I edges, and if that is not successful, will look through TC edges. In the case of non-overlapping instances, the type constraint will be resolved through class `A`, but for this case, the instance declaration is more specific and takes precedence. An example where ambiguous overloading arises, with proper overlapping instances, is discussed in Section 4.2.

4 Evaluation of Prototype Implementation

For evaluating the proposed implementation, the expected behaviour of a type checker for languages that support type classes was considered to be that of Haskell's type checker. Thus, a set of 35 representative programs was meticulously devised and executed on both the proposed implementation and Haskell's built-in type checker. This section presents a justification for the suitability of the test suite as a comprehensive and representative set of tests. Subsequently, the behavior of this implementation on the provided programs is thoroughly examined.

4.1 Test Suite

The test suite encompasses six distinct categories, each targeting specific aspects of the type checker's functionality. The aim is to provide a thorough assessment of the correctness and robustness of our implementation by incorporating these diverse test cases.

The first category focuses on basic type checking, ensuring that the type checker can correctly infer types and detect type errors in simple programs that utilize type classes. This set of tests covers basic language constructs and operations, serving as a foundation for subsequent evaluations.

The second category examines the instance resolution mechanism within the type checker. These tests assess the type checker's ability to resolve instances correctly, especially in scenarios where multiple instances are defined for a particular type class. Through the evaluation of the instance resolution functionality, the aim is to validate the proper selection and utilization of instances based on the context.

Challenges arise when dealing with overlapping instances, which necessitates dedicated tests in the third category. These tests focus on scenarios where instances overlap, thereby evaluating the type checker's capability to handle ambiguity and detect overlapping errors accurately. Verifying the correct resolution of overlapping instances is crucial for ensuring reliable type checking outcomes.

The fourth category explores the type checker's behavior with polymorphic functions. These tests examine how well the type checker infers and handles type variables and constraints within polymorphic functions that involve type classes. By evaluating this aspect, the type checker's ability to handle the complexity introduced by polymorphism and type classes concurrently can be assessed.

The fifth category focuses on type class constraints in method signatures of functions. These tests investigate how the type checker handles these constraints. By evaluating this aspect, the proper integration of type class constraints into the type checking process is ensured.

Lastly, the test suite includes tests that assess error handling. This category includes programs that contain type errors. Thus, this set of tests evaluates the type checker's error reporting capabilities, ensuring clear and accurate error messages for ease of debugging.

By covering these six categories of tests, the comprehensive test suite aims to provide a holistic evaluation of the correctness and reliability of the prototype implementation. These tests also serve as a means to identify potential bugs and areas of improvement for the type checker. However, we note that this set of tests only covers the baseline of what a language widely used in practice covers.

4.2 Results

This subsection presents the results of evaluating the implementation against the aforementioned test suite. The results are summarized in Table 1, categorizing the tests based on whether they should type check without errors (*Type checks*) or should result in type check errors (*Type check error*). The table provides the percentage of tests exhibiting correct and incorrect behavior, with correctness defined as emulating the behavior of Haskell's type checker.

Table 1: Results of prototype implementation.

Haskell \ Prototype	Type checks	Type check error
Type checks	82.61% (19/23)	17.39% (4/23)
Type check error	8.33% (1/12)	91.67% (11/12)

Out of the total number of tests conducted, we observed that 30 out of 35 tests passed. The test suite has 12 tests that should raise an error, and 23 that should type check. The results suggest that the implementation works better when identifying type check errors and has difficulty in type checking correctly when no errors should arise. We look at two examples more in-depth, one that should throw an error and correctly does so, and one that fails to type check.

Furthermore, in the process of evaluating our approach, we encountered a specific case related to overlapping instances. Consider the code snippet shown in Figure 5, where two instances, `A a: INST(A)` and `Int: INST(A)`, could potentially match the same path in our implementation, `P*I?D`. Additionally, the type of the argument could be considered to resolve either constraint. In line with Haskell's behavior, our type checker does not make a choice between `Num` or `A a` as the argument type for `42`, resulting in an error being raised. This test program was inspired by [5], highlighting the handling of ambiguous types and overlapping instances. Our implementation aligns with the expected behavior in Haskell, which detects overlapping instances for `f` and raises an error.

```
class A a where
    f :: a -> Bool

instance A a where
    f :: a -> Bool
    f x = False

instance A Int where
    f :: Int -> Bool
    f x = True

foo :: A a => a -> Bool
foo x = f 42
```

Figure 5: Example program illustrating overlapping instances in Haskell.

For the failing tests, it has been observed that these all have something in common: all of the programs have references to parameters inside instances or type classes. An example of such a program can be found in Figure 6. The current implementation raises an error, stating that no matching declaration was found for `x`. Since function `f` introduces a new scope, say 2, the declaration for its parameter, `x: Int`, is found inside 2. If we consider the instance's scope to be 1, there is a P-labeled edge from 2 to 1. The current implementation seems to add the reference to `x` in scope 1 instead of the scope of the function, namely 2, and no declaration for `x` can be found through `P*(TC)?D` paths. Concretely, this means that when we type check the program within the instance declaration, we use the instance scope and not the one introduced by `f`, and there is no edge we can take to get inside the function scope.

```
class A a where
    f :: a -> Int

instance A Int where
    f :: Int -> Int
    f x = x + 2

foo :: Int -> Int
foo y = f y
```

Figure 6: Example program illustrating failing test case.

To resolve the failing test cases, the implementation needs to be updated to either find the correct function scope even within instances and classes. Regarding the one test that does not raise an error when it should, this is because, currently, we allow for functions with the same name to be declared, as long as their signatures are not identical. Haskell does not allow for this, thus this is a difference in language semantics that should be resolved if Haskell's semantics are needed.

It is important to acknowledge that our test suite, while comprehensive, is not exhaustive and represents a subset of the language with support for type classes commonly used in practice. Although we have made efforts to include diverse and challenging cases, it is possible that certain edge cases and intricate interactions within a complex type system have not been fully captured. Therefore, it is essential to interpret the results with caution and recognize that the evaluation's scope is limited by the coverage of the test suite.

We conclude that our approach can be used as a baseline for type checking type classes using scope graphs, whilst missing more interesting features. Although the prototype implementation resulted in incorrect behavior, thus revision of the code is necessary, the proposed model using scope graphs still proves to be reliable.

5 Ethical Considerations

We undertook this study with a strong emphasis on conducting research in a manner that aligns with responsible practices and principles. Throughout the research process, particular attention was given to ensuring transparency and reproducibility.

To promote transparency, a clear documentation of the tools used in building the type checker is provided in Subsection 2.2. This documentation allows other researchers to understand the approach taken and replicate it if desired. Additionally, to further facilitate reproducibility, we made the implementation of the type checker openly accessible on Github[3]. By sharing the implementation, along with the test suite, it enables others to examine and validate the findings presented in this research.

[3]https://github.com/andreealmocanu/scope-graph-type-class

In the interest of full transparency, all results obtained during the evaluation process have been reported, including both successful and failing test cases that are part of the test suite used for evaluation. No results have been omitted or exaggerated, ensuring the accuracy and integrity of the findings. In Subsection 4.2, we include False Positives (test cases that are expected to not type check, but do so in our implementation) and False Negatives (test cases that are expected to type check, but do not in our implementation) and explicitly report them as what these are. In the context of type checking, presenting false positives as positives can have significant implications in real-life applications. These can lead to incorrect program behavior or runtime errors, causing system failures, data corruption, or security vulnerabilities. The consequences can be severe, particularly in safety-critical systems, where incorrect type checking can compromise the integrity and reliability of the entire system.

We strive to bridge the gap between technical expertise and societal impact. We recognize that as engineers, our responsibilities extend beyond the mere technical creation of products, contrary to what separatism promotes. Separatism is defined in [12] as "the notion that scientists and engineers should apply the technical inputs, but appropriate management and political organs should make the value decisions." In our efforts to recognize engineers' responsibility, we recognize the impact that faking results could have and that

Furthermore, it is important to acknowledge that the correctness of the implementation is determined against a restricted set of tests. Given the complexity of building a type checker with support for type classes, there is a possibility that bugs have been introduced in the implementation and were not caught by the test suite, or may arise in future implementations. To minimize the probability of such occurrences, rigorous testing is essential. The current test suite presented in Subsection 4.1 consists of 35 test cases and does not include programs that are not supported by the mini-language we use throughout this paper, whose limitations are discussed more in Section 6. Thus, this suite only covers a subset of Haskell's features that involve type classes.

By adhering to responsible research practices, including transparency, reproducibility, and not promoting separatism, this study strives to contribute to the collective advancement of the field. These considerations not only enhance the integrity of the research, but also encourage further exploration and refinement of our proposed approach.

6 Discussion

In this section, we compare the declarativity and feature extensibility of our implementation for type checking type classes with the prototype implementation given in [3]. We are particularly interested to make a comparison with [3], as it introduces a type system similar to Haskell's current approach. This comparison aims to evaluate the strengths and limitations of our approach in relation to a widely adopted language, such as Haskell.

Declarativity refers to the ability of a type system to provide concise and expressive constructs for specifying types and their relationships. Haskell's type checker is renowned for its declarative nature, offering a rich set of type inference features. Haskell's extended Hindley-Milner type system can infer the principal type of most expressions, making explicit typing usually optional. For more information on Hindley-Milner type system, we guide the reader towards the papers that laid the foundation for these systems, namely [2] and [6].

In contrast, our prototype type checker does not possess the same level of declarativity as Haskell. Due to the limited language features of the mini-language employed in our prototype, explicit type annotations are required to guide the type checker in the absence of comprehensive type inference capabilities. This lack of automatic type inference diminishes the declarative nature of our approach, as it necessitates explicit type declarations.

A simple example in which explicit typing is necessary even for Haskell's type inference system is given in Figure 7. In this example, the parameter of `foo` has a type constraint, but in the declaration `f 42`, multiple types can be inferred for the argument `42`: Int, Float, Double etc. Depending on which one it infers, the result of Haskell's type checker would be different, as only `Int` is an instance of `A`. This example highlights that omitting explicit typing cannot be done for all cases, even for a complex type checker such as Haskell's, and how type annotations guide the type checker.

```
1 class A a where
2     f :: a -> Bool
3
4 instance A Int where
5     f :: Int -> Bool
6     f x = True
7
8 foo :: A a => a -> Bool
9 foo x = f (42::Int)
```

Figure 7: Example program with explicit typing in Haskell.

Feature extensibility refers to the ability of a type system to accommodate and handle a wide range of language features. Among the features that Haskell's type checker, we find higher-order functions, type classes, subclasses of type classes, type constructors, and polymorphic types. This extensibility empowers programmers to write expressive and flexible code, enabling sophisticated abstractions and reusable components.

In comparison, our prototype type checker is limited in feature extensibility. The mini-language designed for this paper intentionally omits advanced language features, which are also discussed in Section 6. Consequently, the scope of our prototype is more restricted, and it may not be directly applicable to scenarios that necessitate these features. The absence of these language constructs limits the flexibility and expressiveness of our type checker and restricts its ability to handle complex programming paradigms.

In [3], Mark Jones introduces *constructor classes*, which were an extension to what type classes in Haskell were defined to be at that time. Before [3], type classes in Haskell provided a mechanism for ad-hoc polymorphism, but were defined on types alone, based on [13]. Constructor classes

generalize type classes to operate on data constructors defined in the program, supporting the use of monads. Nowadays, there is no distinction in Haskell between constructor classes and type classes [5].

In terms of feature extensibility and expressiveness, our prototype is limited and does not support data constructors and type classes defined for those. Thus, our implementation lacks the novelty introduced by [3], but the approach could be extended once data constructors are introduced in the language.

Another feature that is mentioned in [3] is qualied types. Qualified types are defined by Mark Jones in [4] as types for which some predicates *P* hold: $P \Rightarrow ty$ in our syntax. P is defined to be "an expression of the form $P\ C_1...C_n$, representing the assertion that constructors $C_1...C_n$ are related by P" [3]. As we do not support constructors, we refer to these as type variables with class constraints instead of constructors. In our implementation, we encode qualified types as type variables with zero or one class constraint. Currently, we resolve at most one class constraint for a type variable, while in [3], *P* is a set of predicates.

For dealing with qualified types, Mark Jones uses the typing rules that can be found in Figure 8. In these rules, ρ is defined as a qualified type, $\rho = P \Rightarrow \tau$, and τ represents types. Rule ($\Rightarrow E$) states that, given an expression E with a predicate π on $\Rightarrow \rho$, and that predicate π can be inferred from the predicates in P, E has type ρ. The predicate is eliminated and E has qualified type ρ. Our scope graph encoding works similarly, if we consider P to be the set of type classes available within the program, and π to be a class constraint.

Qualified types:

$$(\Rightarrow E)\quad \frac{P \mid A \vdash E : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid A \vdash E : \rho}$$

$$(\Rightarrow I)\quad \frac{P, \pi \mid A \vdash E : \rho}{P \mid A \vdash E : \pi \Rightarrow \rho}$$

Figure 8: Typing rules for qualified types, defined by Mark Jones.

The second rule for dealing with qualified types, rule ($\Rightarrow I$), states that if expression E requires predicate π, and π is available within the context, predicate π can be put into the qualified type.

In the same paper, Mark Jones also proposed typing rules for dealing with (parametric) polymorphism. Parametric polymorphism can be explained as when a function is defined over a range of types, acting essentially in the same way for each type. This can easily be illustrated through an example: take the function that returns the length of a list. This `length` function acts in the same way for a list of integers, characters, or floats. Because our language did not include type schemes and data constructors, the functionality for this type of polymorphism has not been implemented. Nonetheless, using scope graphs as a model is still a viable option for dealing with polymorphism, as it does not impose this limitation.

The language designed for this research paper takes inspiration from Haskell's syntax. However, it was intentionally kept simple to focus on a subset for which it is feasible to provide an implementation in the given timeframe. As a result, this simplicity introduces limitations to the expressiveness of the language. Notably, the mini-language does not support language features such as subclasses and type or data constructors. These limitations restrict the applicability of the type checker to more complex scenarios where these features are used.

Future work for this research would include extending the approach to type constructors, and eventually subclasses. Type and data constructors would add a great amount of expressiveness to the language defined in this paper. After extending the syntax and semantics to support these features, the complexity of the type checker will increase, as it will have to deal with resolving constructor instances. Resolving instance resolution in that case may not be as simple as in our approach, as the possible types range over a much wider set; it would allow for defining new types within a program.

7 Related Work

The concept of type classes in programming languages, as introduced in [13] has been widely studied and extended in the literature. Several approaches have been proposed to build type checkers for type classes, each with their own advantages and limitations.

In [13], a practical solution for resolving ad-hoc polymorphism through type classes is proposed. Their work introduces the limitations of ad-hoc polymorphism and elaborates on the mechanisms for type class instance resolution. Although our research paper diverges from their specific approach, their insights into the practical aspects of type class implementation and the challenges associated with instance resolution have informed our considerations.

Another approach to type checking type classes is presented in [7]. This paper aims to give the simplest implementation to Haskell's type inference algorithm, using sorts (sets of type classes). A less general approach, that explicitly focuses on type classes in Haskell, can be found in [1]. This paper also provides an extensive problem overview, which helped with the understanding of type classes and their specifics in Haskell.

Current approaches seem to use mechanisms specific to type classes, which can introduce additional complexity and may require a deeper understanding of the exact features. In contrast, the use of scope graphs provides a unified and more general framework that is not tied to specific type class extensions or language constructs. Using scope graphs for type checking type classes seems to be an approach that is more intuitive than other existing models. Moreover, solutions with scope graphs are more general and can capture various scoping mechanisms, including non-lexical scoping.

Regarding type checking using scope graphs, one influential work in this area is [11], which presents a novel approach to resolving more complex type systems, including structural records and generic classes, using *scopes as types*. The concepts and ideas presented in this paper have influenced the design choices made in our type checker.

Our research paper builds upon these prior works by proposing a new approach for building type checkers for type classes using scope graphs. By leveraging the concept of

scopes and their relationships, we aim to address some of the challenges in type class resolution and provide a more principled and modular framework for type checking type classes.

8 Conclusion

In this paper, we addressed the question of how to build a type checker for a language with support for type classes, using scope graphs as a model. We presented an approach that leverages scope graphs to capture the relationships between types and type classes, allowing for effective type checking.

By designing and implementing a prototype type checker for type classes using scope graphs, we demonstrated the feasibility of our approach. Using a comprehensive test suite, we evaluated this approach. Our results indicate that type checking type classes using scope graphs is a promising approach.

However, our research also revealed several areas for improvement and future work. First, the limited language features of the mini-language used in this research paper restricted the expressiveness and applicability of our approach. Future work should explore extending the type checking system to support advanced language constructs such as subclasses and type constructors. By incorporating these features, the type checker can address a wider range of programming scenarios and offer more comprehensive support for real-world languages. Additionally, further research is needed to enhance expressiveness of the type checking system. While our prototype type checker relied often times on explicit type annotations due to the absence of extensive type inference capabilities, future work can focus on developing more advanced type inference algorithms.

In conclusion, we have shown that building a type checker for a language with support for type classes using scope graphs is a promising approach. While there are limitations and areas for improvement, our research provides a foundation for future work in this domain. By addressing the challenges of feature extensibility and declarativity, we can advance the state of type checking and contribute to the development of type checking algorithms that are independent of language.

Acknowledgements

We would like to thank the supervisor and responsible professor for providing guidance throughout this Research Project and valuable feedback on previous drafts of this paper.

References

[1] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, mar 1996. doi:10.1145/227699.227700.

[2] Roger Hindley et al. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[3] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 52–61, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/165180.165190.

[4] Mark P Jones. *Qualified types: theory and practice*. Number 9. Cambridge University Press, 2003.

[5] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Haskell 2010 language report. Technical report, 2010. URL: https://www.haskell.org/onlinereport/haskell2010/.

[6] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[7] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 409–418, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/158511.158698.

[8] 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. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_9, doi:10.1007/978-3-662-46669-8_9.

[9] 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.

[10] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13:11–49, 2000.

[11] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi:10.1145/3276484.

[12] Ibo Van de Poel and Lambèr Royakkers. *Ethics, technology, and engineering: An introduction*. John Wiley & Sons, 2023.

[13] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/75277.75283.