Type checker for a language with a substructural type system using scope graphs

Jan Knapen[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: Jan Knapen
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Aron Zwaan, Thomas Durieux

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Substructural typing imposes additional constraints on variable usage during type checking and requires specialized approaches to ensure type soundness. In this study, we investigate the implementation of a type checker using scope graphs for languages with substructural type systems. Scope graphs, a data structure representing scoping, provide a foundation for defining type checking algorithms. Our research project extends an existing Haskell library, incorporating typing rules for non-substructural, linear, and affine type systems. Through careful examination and comparison of scope graph and calculus implementations, we evaluate their expressiveness, extensibility, and readability. While the scope graph implementation demonstrates promising results, passing all test cases, the calculus implementation encounters unification errors in a subset of the tests. We conclude that the scope graph implementation offers a solid foundation for substructural typing, with potential for easy extension and integration with other language features. However, further work is needed to develop a comprehensive test suite and address the challenges faced by the calculus implementation. By advancing these areas, we can enhance the effectiveness of substructural type checking and enable more reliable and secure programming practices in languages with substructural type systems.

1 Introduction

1.1 Type Checking

Type checking is a process used in computer programming to verify the types of variables, expressions, or functions in a program. In simple terms, it involves ensuring that the data types used in a program are consistent with the intended usage of those data types.

The importance of type checking lies in its ability to catch errors that may otherwise go unnoticed, resulting in incorrect program behavior or crashes. For example, if a program tries to perform arithmetic operations on a string, a type checking system would flag this as an error and prevent the program from executing.

1.2 The Problem: Substructural Programming Languages

Substructural programming languages are a type of programming language that impose additional constraints on the use of variables and other resources. These constraints are designed to prevent certain kinds of errors and make programs more reliable and secure.

A good example of this is file handling. With linear typing, which enforces that variables and resources are used exactly once, we can ensure that a file is released properly safeguarding against resource leaks.

Some common types of constraints used in substructural programming languages include linear types, affine types and relevant types [6]:

- Linear types are types that can only be used once, meaning that a variable of a linear type can be used exactly once in a program.
- Affine types are similar to linear types, but allow that variables are not used, meaning that a varibale of a affine type can be used at most once in a program.
- Relevant types are types that are used at least once. This ensures that variables are necessary for the program's execution.

1.3 Scope Graphs

Scope graphs are a data structure that can represent the lexical scoping of names in a program. They are a directed graph that captures the nesting of scopes and the relationships between them. Each node in a scope graph represents a scope, which is a region of a program where a name is defined and can be accessed. Connected with edges to the nodes there are leaves which represent variable declarations, module imports, etc. Scope graphs give us the possibility to delegate the challenge of resolving names during type checking to type checking algorithms we define once and for all [5].

However they can represent non-lexical scoping as well. While they are commonly used to capture the lexical scoping of names in a program, scope graphs were designed to handle various scoping mechanisms, including non-lexical scoping. The key concept behind scope graphs is to represent the relationships between scopes and their nesting hierarchy, regardless of the specific scoping rules involved. This allows scope graphs to accomodate different scoping mechanisms, such as dynamic scoping or other forms of non-lexical scoping.

1.4 Monotonicity

Monotonicity, a fundamental property of scope graphs, plays a crucial role in ensuring the integrity of edge queries within a given scope. This property guarantees that once edges have been queried within a scope, they cannot subsequently be utilized to establish sinks or dependencies within that same scope. Monotonicity serves as a vital mechanism for preserving the consistency and reliability of scope graphs, preventing the introduction of unexpected sinks or dependencies that may compromise the integrity of the graph structure.

For example, let's consider a scope graph representing a programming language. Each node in the graph represents a scope, such as a function or a block of code, and edges represent the relationships between scopes, such as variable references or function calls.

Now, suppose we have a scope A that contains a variable declaration and a scope B that references that variable. The graph would have an edge from B to A, indicating the dependency.

If the monotonicity property is violated, it would mean that after querying the edges in scope B, additional edges are introduced that establish sinks in scope B, i.e., references to variables that are not actually present in scope B. This would lead to an inconsistent graph structure where the dependencies do not accurately reflect the actual scope and visibility of variables.

1.5 Linear Typing in Rust

"Rust uses a strong type system based on the ideas of ownership and borrowing, which statically prohibits the mutation of shared state" as stated by Jung et al (2021) [3]. This ownership system and borrowing rules can be seen as inspired by substructural types. Rust's ownership system ensures that each value has a single owner at any given time, and it enforces strict borrowing rules to prevent data races and memory unsafety.

The concept of ownership in Rust is similar to the idea of linear types, where a resource can only have one owner, and ownership can be transferred or borrowed temporarily. The borrow checker in Rust enforces rules such as exclusive borrowing (mutable references) and shared borrowing (immutable references) to prevent multiple mutable references or dangling references.

While Rust's ownership and borrowing rules provide some substructural-like guarantees, it does not fully implement all the features of a pure substructural type system. For example, Rust does not directly support features like unrestricted duplication of values or explicit linear types annotations. However this makes substructural typing a key element to be implemented using scope graphs, because we want to be able to define type checking algorithms for any language including programming languages inspired by substructural type systems.

1.6 Research Question

In this research project, we *explore the implementation of a type checker using scope graphs for languages with a substructural type system*, specifically focusing on linear and affine type disciplines, using the Haskell programming language and the phased Haskell library [2]. Rather than focusing on a specific programming language, an abstract syntax is defined to represent a parsed programming language.

First, we will provide a comprehensive problem description. Next we present the contribution of the research done highlighting the solutions proposed to tackle the identified problem. Following the contribution, we proceed to the evaluation stage, where we present the results of our research using designed test cases. Then we dedicate a section to discussing the ethical considerations of our work. The subsequent section is dedicated to the discussion of our findings and their interpretation. In the last section we present the conclusion of our paper. Furthemore we outline potential avenues for future work in this section, identifying unresolved questions, new directions, or extensions to our research that could be explored to advance the field. At last, related work is mentioned.

2 Problem Description

2.1 Haskell Library

The use of scope graphs for type checking is a topic of ongoing research in the field of programming languages. There is a Haskell library still in development for writing type checkers that construct scope graphs [2]. This library has not been used yet to write a type checker for substructural programming languages.

In this research project we aim to extend the phased Haskell library to be able to construct scope graphs for languages with a substructural type system. The library can be split into the following phases:

1. **ATerms**, resulting from parsing the initial programming language string.
2. **Abstract Syntax**, resulting from parsing the ATerms.
3. **Typechecking program**, input conform the Abstract Syntax and output is the typechecking result.

The first phase is skipped in this research project because there are programming languages with a pure substructural type system. The Abstract Syntax used in the second phase consists of Expressions and Types. It should be extended and adjusted to facilitate the substructural type systems. The behavior of the third phase should also be adjusted accordingly. A fourth phase needs to be introduced to enable checking for substructural types, more about this later.

2.2 Methodology

The methodology employed in this research project aimed to investigate the implementation of typing rules using scope graphs. The overall approach involed several steps. First we started by defining the typing rules that would serve as the foundation for our investigation. As starting point we used the typing rules as defined by David Walker (2002) [6]. These rules were carefully (re)formatted to capture the desired behavior and constraints of the system under study.

The typing rules were implemented without using scope graphs, the purpose of this implementation is to evaluate the later implementation using scope graphs. Both use a similar Abstract Syntax so that test cases are clearly the same and can be evaluated easily.

Next, we embarked on a thorough exploration of different ideas and approaches to incorporate these typing rules into scope graphs. This step allowed us to gain a comprehensive understanding of the challenges and opportunities associated with leveraging scope graphs for enforcing type contraints.

After analyzing the possible solutions, we proceeded to implement the chosen solution into the phased Haskell library. To assess the proposed approach, we designed test cases for evaluation. These test cases encompassed various scenarios and scenarios that would exercise the implemented typing rules both with and without the integration of scope graphs. By comparing the results of these experiments, we were able to qualitatively evaluate the implemantation.

3 Contribution

3.1 Typing Rules

In this subsection we will discuss the defined typing rules that are the base of this research project. We started from the typing rules defined by David Walker (2002) [6] and made adjustments for our specific use. First we define the simply-typed lambda calculus rules. Then we define the typing rules for linear types, which are the initial focus of this research project. Later we define the typing rules for affine types which are very similar. The typing rules are later used for the implementation and also combined.

For each type system we define typing rules for each Expression of the Abstract Syntax. The are respectively $Ident\ x$ (Variable Identifier), $Plus\ e_1\ e_2$ (Addition), $App\ e_1\ e_2$ (Application), $Abs\ x\ t\ e$ (Function Abstraction) and $Let\ x\ t\ e1\ e_2$ (non-recursive Let-Binding).

Simply-Typed Lambda Calculus Rules

The following equations are straightforward for each Expression. Since the typechecker is adjusted to also return the Context along the resulting Type, because the Context may be changed since we are removing variables in case of linear/affine, the following typing rules are adjusted to show how the returned context is used.

We can take typing rule 2 as an example of how to read the typing rules:

- We first read the upper part of the fraction: given a context Γ in which we can successfully type check expression e_1 to a Num, resulting in an updated context Γ', and further, in the updated context Γ', we can successfully type check expression e_2 to a Num, resulting in a final context Γ''.
- We then read the lower part: the typing rule concludes that we can type check the Plus expression $Plus\ e_1\ e_2$ to type Num in the original context Γ, resulting in the final context Γ''.

  Other symbols that we use:

  - $t_1 \multimap t_2$: Function type from type t_1 to t_2.
  - $\Gamma \equiv \Gamma', \Gamma''$: Context Γ splitten into the two context Γ' & Γ''.
  - Γ_x: Partial context of context Γ that only binds variable x.
  - $\Gamma_{\overline{x}}$: Partial context of context Γ that binds everything except variable x.
  - $x : t \notin \Gamma$: Variable x is not binded with type t in context Γ.
  - $\Gamma\backslash_1\{x : t\}$: Resulting context Γ without the binding of variable x with type t.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \quad (1)$$

$$\frac{\Gamma \vdash e_1 : Num; \Gamma' \quad \Gamma \vdash e_2 : Num; \Gamma''}{\Gamma \vdash Plus\ e_1\ e_2 : Num; \Gamma''} \quad (2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \multimap t_2; \Gamma' \quad \Gamma \vdash e_2 : t_1; \Gamma''}{\Gamma \vdash App\ e_1\ e_2 : t_2; \Gamma''} \quad (3)$$

$$\frac{x : t_1, \Gamma \vdash e : t_2; \Gamma'}{\Gamma \vdash \lambda(x : t)\ .\ e : t_1 \multimap t_2; \Gamma'} \quad (4)$$

$$\frac{\Gamma \vdash e_1 : t_1; \Gamma' \quad x : t_1, \Gamma' \vdash e_2 : t_2; \Gamma''}{\Gamma \vdash let\ x : t = e_1\ in\ e_2 : t_2; \Gamma''} \quad (5)$$

Linear Typing Rules

Variables are removed from the context when they are used.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t; \Gamma\backslash_1\{x : t\}} \quad (6)$$

Both the Addition and Applicaiton typing rules stay the same.

$$\frac{\Gamma \vdash e_1 : Num; \Gamma' \quad \Gamma' \vdash e_2 : Num; \Gamma''}{\Gamma \vdash Plus\ e_1\ e_2 : Num; \Gamma''} \quad (7)$$

$$\frac{\Gamma \vdash e_1 : t_1 \multimap t_2; \Gamma' \quad \Gamma \vdash e_2 : t_1; \Gamma''}{\Gamma \vdash App\ e_1\ e_2 : t_2; \Gamma''} \quad (8)$$

When type checking a Function Abstraction we split the context to not contain x-bindings when type checking the body with the parameter x. We then add the paramter variable and type check the body. At the end after typechecking the body there should be no x-bindings because linear variables should be used exactly once. After type checking the Function Abstraction we add back the x-bindings of the original context.

$$\frac{\Gamma \equiv \Gamma_{\overline{x}}, \Gamma_x \quad x : t_1, \Gamma_{\overline{x}} \vdash e : t_2; \Gamma'_{\overline{x}} \quad x : t_1 \notin \Gamma'_{\overline{x}}}{\Gamma \vdash \lambda(x : t)\ .\ e : t_1 \multimap t_2; \Gamma'_{\overline{x}}, \Gamma_x} \quad (9)$$

When type checking a Let-Binding, first we typecheck the first expression. Then we split the resulting context to not contain x-bindings. We then add the let variable and type check the second expression. At the end after typechecking the second expression there should be no x-bindings. After type checking the Let-Binding we add back the x-bindings of the original context.

$$\frac{\Gamma \vdash e_1 : t_1; \Gamma' \quad \Gamma' \equiv \Gamma'_{\overline{x}}, \Gamma'_x \quad x : t_1, \Gamma'_{\overline{x}} \vdash e_2 : t_2; \Gamma''_{\overline{x}} \quad x : t_1 \notin \Gamma''_{\overline{x}}}{\Gamma \vdash let\ x : t = e_1\ in\ e_2 : t_2; \Gamma''_{\overline{x}}, \Gamma'_x} \quad (10)$$

The last two typing rules ensure that there is no variable shadowing by removing the x-bindings before type checking the body. This allows us to remove the variable binding in the first typing rule, else in the second it would use the shadowing variable.

Affine Typing Rules

Variable Identifier, Addition and Application are the same as for Linear.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t; \Gamma\backslash_1\{x : t\}} \quad (11)$$

$$\frac{\Gamma \vdash e_1 : Num; \Gamma' \quad \Gamma' \vdash e_2 : Num; \Gamma''}{\Gamma \vdash Plus\ e_1\ e_2 : Num; \Gamma''} \quad (12)$$

$$\frac{\Gamma \vdash e_1 : t_1 \multimap t_2; \Gamma' \quad \Gamma \vdash e_2 : t_1; \Gamma''}{\Gamma \vdash App\ e_1\ e_2 : t_2; \Gamma''} \quad (13)$$

For type checking Function Abstraction the difference with linear is that it is possible that the variable x is not used. So we remove the possible x-binding, which was added due to the lambda parameter, after type checking the body.

$$\frac{\Gamma \equiv \Gamma_{\overline{x}}, \Gamma_x \quad \Gamma' \equiv x : t_1, \Gamma_{\overline{x}} \quad \Gamma' \vdash e : t_2; \Gamma'' \quad \Gamma'' \equiv \Gamma''_{\overline{x}}, \Gamma''_x}{\Gamma \vdash \lambda(x : t) \,.\, e : t_1 \multimap t_2; \Gamma''_{\overline{x}}, \Gamma_x} \tag{14}$$

This is also the difference for Let-Bindings, we simply remove the x-binding that might be left after type checking the second expression and add back the x-bindings of the context that resulted after typechecking the first expression.

$$\frac{\Gamma \vdash e_1 : t_1; \Gamma' \quad \Gamma' \equiv \Gamma'_{\overline{x}}, \Gamma'_x \quad \Gamma'' \equiv x : t_1, \Gamma'_{\overline{x}} \quad \Gamma'' \vdash e_2 : t_2; \Gamma''' \quad \Gamma''' \equiv \Gamma'''_{\overline{x}}, \Gamma'''_x}{\Gamma \vdash let\; x : t = e_1 \; in\; e_2 : t_2; \Gamma'''_{\overline{x}}, \Gamma'_x} \tag{15}$$

3.2 Calculus Implementation

The calculus implementation involved utilizing the existing code in the `lang-hm` subfolder as a boilerplate and can be found in the new subfolder `./lang-linear-calculus` in the project's GitHub repository [4]. This provided a foundation for incorporating the newly defined typing rules into the system. In the case of the non-substructural typing rules they closely resembled those already implemented. To accommodate substructural types, the syntax was extended by introducing $linearT$ and $affineT$ as additional layers over the actual types. These layers were used to indicate the substructural nature of the actual types behind them. Furthermore, the implementation added support for let-bindings, which were previously absent. To enforce substructural typing, the $AlgJ$ typechecking function was adjusted to include the return of the adjusted context after checking each expression. This modification was necessary to remove used substructural variables from the context, thereby ensuring compliance with substructural typing rules. For instance, the $Ident$ typing rule from the Linear Typing rules, as defined in equation 6, demonstrates the removal of a variable from the context after its usage. Moreover, the $AlgJ$ typechecking function was enhanced to enforce linear variable usage in let-bindings and lambdas, mandating that they are utilized at least once. This is not necessary for the affine variables as they can also be left unused.

3.3 Researched Solutions using Scope Graphs

The next step in this research project was translating the typing rules into scope graphs. The main challenge was keeping track of total usages of a variable without giving up monotonicity. Some of the considered but not chosen solutions are:

- Delete declaration edge to variable when used. Problem: this gives up monotonicity since any queries after that are the same return something different
- Add meta-deta to variable declaration keeping count of total usages. Problem: Haskell does not have mutable data, so it's not possible to edit the variable declaration to update the count.

3.4 Chosen Solution

To solve this problem without giving up the monotonicity property, we introduce a new phase after the type checking phase. In the type checking phase the scope graph is being constructed and we keep count of the total usages of a linear/affine variable by adding $UsageDecl$ after each query. In

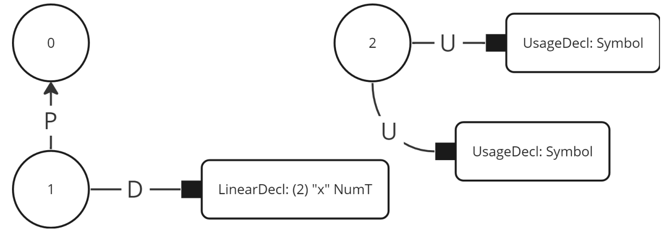

Figure 1: Example Code corresponding Scope Graph

the next new phase, we check the whole scope graph for linear/affine variables and check if they have a valid total count of usages.

We will use listing 1 as an example where we will asume the let is linear, this code snippet results into the scope graph shown in figure 1. Given a code snippet we start with scope 0, then the first line declares linear variable x. We thus create a $LinearDecl$ where (2) points to a new empty scope which is used to keep count of total usages. $"x"$ is the variable name and $NumT$ is the actual underlying type of the variable. This way we create a layer over variable x, whereas a normal variable x would simply be a $Decl\ "x"\ NumT$. In line two of the example code we use the variable x two times, so two queries are executed in scope 1. This leads to the $LinearDecl$ we just declared. For each query we declare a new $UsageDecl$ in scope 2, using them we can count the total usages in the later phase where we check all linear/affine types. The $Symbol$ used by $UsageDecl$ is used to make each $UsageDecl$ unique, this is necessary because else we will get an error that the $UsageDecl$ is already declared.

In the last phase we go over all scopes in the constructed scope graph and check if they contain any Linear Declarations. We then use the scope given bij the $LinearDecl$ to count the total usages. If this total is not exactly one, then we throw a Linear error, since all linear variables should be used *exactly* once.

```
let x = 4
  in x + x
```

Listing 1: Example Code

3.5 Implementation of Chosen Solution

The implementation using scope graphs involved utilizing the existing code in the `lang-mylang` subfolder as a boilerplate and can be found in the new subfolder `./lang-linear-scope-graph` in the project's GitHub repository [4]. Similar to the calculus implementation, this process involved extending the syntax to incorporate let-bindings and the $LinearT$ and $AffineT$ types, which functioned similarly to their counterparts in the calculus implementation. Furthermore, a new label, namely U, was introduced to serve as an edge for the new $UsageDecl$ in order to distinguish it from variable declarations. The type checking function was modified to accommodate the usage of substructural types and declarations. Finally, the $runTC$ function, which acts as the central orchestrator, was expanded with an additional phase. This phase involved checking all substructural types, as explained in the previous section, thereby com-

pleting the integration with scope graphs and the implementation of the typing rules.

3.6 Affine Typing

To include affine types, we only need a few adjustments. A new declaration with the same parameters is introduced, namely $AffineDecl$. The behavior for this type is the same as linear in the penultimate phase. In the last phase we check the Affine types in a similar way, the only difference is that the total count of usages should not be one, but either zero or one since affine types should be used *at most* once.

4 Evaluation

In this section, we focus on evaluating the calculus and scope graph implementation by conducting tests and code analysis. Firstly, we compare the results obtained from testing both implementations with non-substructural type systems, linear type systems, and affine type systems. Through self-defined test cases, we aim to assess the effectiveness and accuracy of both implementations across these different types of systems. Secondly, we delve into an in-depth analysis of the code behind the scope graph implementation, specifically examining its extensibility, expressiveness, and readability. By scrutinizing these aspects, we aim to gain insights into the implementation's overall design and potential for future improvements.

4.1 Test Cases

The test cases were designed based on the typing rules and were categorized into three groups: non-substructural typing rules, linear typing rules, and affine typing rules. The test suite began with the non-substructural typing rules, and subsequently, the same tests were performed using only Linear or Affine types for the other typing rules. Each category of typing rules included specific test cases for Abs, App, Let, $Complex$ (combinations of earlier mentioned), and $Errors$ (tests that should always result in an error). It is important to note that the expected results for some test cases varied between the calculus and scope graph implementations. This discrepancy arises because the calculus implementation enforces substructural types during typechecking, whereas the scope graph enforces them after typechecking the entire expression.

The complete collection of test cases can be found on the project's GitHub repository [4]. For instance, let's consider a test case for the $Plus$ typing rule for non-substructural type systems:

```
Test Case: Plus Typing Rule
Input: (Plus (Num 0) (Num 1))
Expected Output: NumT
```

In this example, we test the typing rule for addition. The test case ensures that the expression is well-typed according to the $Plus$ rule which states that both expressions should be of type $NumT$. Further test cases for App, Let, $Complex$, and $Error$ scenarios can be explored in the project's GitHub repository [4]:

- Tests for the scope graph implementation can be found at: `./lang-linear-scope-graph/tests/Main.hs`.
- Tests for the calculus implementation can be found at: `./lang-linear-calculus/Test.hs`.

4.2 Results

The evaluation of the scope graph and calculus implementations yielded the following results:

Test Suite	Cases	Tried	Failures
Non-substructural tests	41	41	0
Linear tests	37	37	0
Affine tests	37	37	0

Table 1: Scope Graph Test Results

Test Suite	Cases	Tried	Failures
Non-substructural tests	41	41	0
Linear tests	37	37	6
Affine tests	37	37	6

Table 2: Calculus Test Results

It was expected that all test cases passing for the scope graph implementation would also pass for the calculus implementation, given that the calculus implementation served as the basis for this project. However, the failure of certain test cases in the calculus implementation is attributed to Unification Errors, likely caused by using an Abs (lambda function) as a parameter for another Abs. Notably, these failing test cases pass as expected in the scope graph implementation, which aligns with the project's ultimate goal.

4.3 Code Analysis

The scope graph implementation exhibits various characteristics in terms of expressiveness, extensibility, readability, and documentation.

Regarding expressiveness, the implementation provides clear error messages when types do not match, as exemplified by the "$Let\ x\ t\ e1\ e2$" construct, which generates an informative error message if $e1$ is not of type t. However, the handling of substructural errors is not as expressive, as it only displays a generic error message stating "Substructural typing error" without indicating which type system (linear or affine) enforces the error.

Concerning extensibility, the scope graph library offers two main avenues for extension: the $Type$ and $Expr$ data types. While adding new substructural types is straightforward and the added Linear/Affine types are compatible with any new type, it necessitates modifications in the 'typecheck' function. Specifically, cases within Let, Abs, $Ident$, and App need to be adjusted to account for any new substructural types. Similarly, introducing a new $Expr$ requires considering the existing substructural types when implementing the 'typecheck' function for that expression, ensuring the proper use of different declarations for each substructural type.

Regarding readability, the implementation demonstrates consistency in indentation and adheres to standard Haskell formatting conventions. However, the code lacks comprehensive comments which could be beneficial, particularly in

complex sections or algorithms. Additionally, some variable and function names are concise and lack self-explanatory qualities, making it challenging to understand their purpose without additional context.

In terms of modularity, the code is organized into sections such as "Scope Graph parameters," "Type Checker," and "Tie it all together." This modular structure aids in comprehending the code's flow and purpose.

However, a notable aspect that requires improvement is the documentation. While the code follows some level of readability and organization, comprehensive documentation is lacking, further accentuating the need for additional comments to enhance clarity and understanding.

5 Responsible Research

In our research project, the implementation of the typechecker was conducted with a clear vision for extensibility. We recognized the importance of creating a framework that could accommodate the addition of other types and expressions in the future. To ensure research reproducibility, we have made the project code readily available and easily executable using the cabal package. However, we acknowledge that the evaluation conducted was limited in scope. This limitation raises ethical considerations regarding the validity and comprehensiveness of our findings. In Chapter 4.3 it was noted that documentation is currently very limited. We recognize that this lack of comprehensive documentation is ethically concerning, as it hampers users' ability to understand and utilize the typechecker effectively.

6 Discussion

The results indicate that the overall solution has been successfully implemented, offering potential for easy extension with other substructural typing systems. This finding suggests that the implementations provide a solid framework for handling substructural typing. Additionally, with limited knowledge, the implementations can be easily used when extending with other expressions, showcasing their usability and flexibility.

However, it is important to note that the evaluation was limited by the absence of a comprehensive test suite. While the provided test cases covered a range of scenarios, the lack of a comprehensive test suite hinders a thorough examination of the implementations' behavior and the identification of potential edge cases.

7 Conclusion & Future Work

In conclusion, this research project aimed to investigate the feasibility of implementing a type checker using scope graphs for languages with substructural type systems. The integration of substructural typing imposes additional constraints on variable usage during type checking. Scope graphs, as a data structure representing scoping, offer a foundation for defining type checking algorithms. Their fundamental property of monotonicity ensures the absence of unexpected sinks or dependencies. While the integration of substructural typing remains limited, languages like Rust draw inspiration from linearity with concepts such as ownership.

In this project, we extended the existing Haskell library, starting with typing rules for non-substructural, linear, and affine type systems. These were implemented by extending the existing type checker that can be found in the folder `lang-hm` [2]. Then we extended the existing type checker using scope graphs by using the template folder `lang-mylang`. To support substructural types, we introduced the $LinearDecl$ and $AffineDecl$ that serve as a layer over the actual type. These declarations additionally point to a seperate scope that keeps count of the total usages of the actual variable behind it. An additional type checking phase was implemented to validate the counts of declared linear/affine variables by checking the scopes for each of them. The chosen solution prioritized extensibility of types and expressions. Around 37 test cases were executed to compare and validate both implementations. Encouragingly, all test cases passed for the scope graph implementation, while six of them did not pass for the calculus implementation. These findings highlight the potential of scope graphs as a promising approach for implementing type checkers in substructural type systems.

For future work, several avenues can be explored to further enhance the scope graph implementation and extend its capabilities:

Firstly, developing a more comprehensive test suite would be beneficial. This entails not only covering additional edge cases but also considering combinations of non-substructural, affine, and linear typing scenarios. This broader range of tests would provide a more thorough evaluation of the implementation's robustness and identify any potential areas for improvement.

Addressing the unification error that caused the failing test cases in the calculus implementation is another crucial aspect. Investigating the root cause of this error and devising a solution to resolve it would ensure that the calculus implementation aligns with the expected behavior and performs consistently with the scope graph implementation.

Furthermore, exploring the integration of the chosen solution with other language features and paradigms, such as type classes, modules, and objects, would be an interesting avenue for future research. Assessing the compatibility and determining the feasibility of combining the scope graph implementation with these constructs would contribute to a more comprehensive and versatile type checking framework.

By focusing on these future directions, researchers and developers can advance the scope graph implementation, strengthen its reliability and accuracy, and explore its potential integration with various language features to support a broader range of programming paradigms.

8 Related Work

Linear type systems have long been a subject of extensive research, holding great promise for programming languages. However, their integration into mainstream programming languages has been limited, as noted by Bernardy et al (2017) [1]. Interestingly, Rust, as discussed in the introduction and supported by Bernardy et al. (2017) [1], incorporates a unique ownership typing system that draws inspiration from linear-

ity. Additionally, Bernardy et al. (2017) [1] have explored the implementation of linearity in Haskell, with a specific focus on *linearity within the function arrow* rather than *linearity in the kinds*. In alignment with this research, our project has also implemented a linear typing rule for lambda functions, explicitly specifying that lambda arguments must be used exactly once in case of linearity and at most once in case of affinity. However whereas this research project only has the type $NumT$ which is a value of atomic base type and is "consumed" by simply evaluating it, the research of Bernardy et al. (2017) [1] extends this as following:

- Linear Function: apply it to one argument and consume its result exactly once.
- Pair: pattern match on it and consume each component exactly once.
- In general: pattern match on a datatype and consume its linear components exactly once.

9 Acknowledgements

I would like to express my gratitude to my supervisor, Aron Zwaan, for his guidance and support throughout this project. Always ready to help and think along the process.

Additionally, I would like to thank Professor Casper Bach Poulsen for his consistent presence in all the weekly meetings, going above and beyond the required obligations. I truly appreciate his dedication and availability for any additional necessary discussions/meetings.

References

[1] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[2] Aron Zwaan Casper Bach Poulsen. Haskell library for writing type checkers using scope graphs. https://github.com/MetaBorgCube/scope-graph-scheduling-bsc-template. Accessed: May 19, 2023.

[3] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Commun. ACM*, 64(4):144–152, mar 2021.

[4] Jan Knapen. Scope graph scheduling bsc substructural type systems. https://github.com/JanKnapen/scope-graph-scheduling-bsc-substructural. Accessed: June 23, 2023.

[5] Casper Bach Poulsen. Building type checkers using scope graphs. https://projectforum.tudelft.nl/course_editions/60/generic_projects/2533. Accessed: June 19, 2023.

[6] David Walker. Substructural type systems. In *https://mitpress-request.mit.edu/sites/default/files/titles /content/9780262 162289_sch_0001.pdf*, page 10, 2002.