Type-Checking Modules and Imports using Scope Graphs

A Case Study on a Language with Relative, Unordered and Glob Import Semantics

Paul Hübner[1]

Supervisor(s): Dr. 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: Paul Hübner
Final project course: CSE3000 Research Project
Thesis committee: Dr. Casper Bach Poulsen, Aron Zwaan, Dr. Thomas Durieux

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

Abstract

Scope graphs provide a way to type-check real-world programming languages and their constructs. A previous implementation that type-checks the proof-of-concept language LM, a language with relative, unordered, and glob imports, does not halt. This thesis discusses a five-step approach for constructing and type-checking a scope graph of an LM program. Using manually scheduled queries and auxiliary algorithms, type-checking the majority of examples failing in previous literature succeeds. The introduction of breadth-first-traversal and multi-origin querying is discussed as new scope graph primitives to aid in the reusability of this thesis for type-checkers that require stratified resolution.

1 Introduction

Presently, many representations of type systems omit an important functionality of real-world programming languages: module systems [14]. Consequently, a language-agnostic representation of a type system that easily enables the representation of modules and imports would provide great value. Scope graphs provide a formal representation of the naming structure [8], with applications for type systems. MiniStatix is a language that allows type-checking with scope graphs using predicates [9]. It aims to fulfill the characteristics of being principled, expressive, declarative, executable, reusable, and resilient [14]. MiniStatix implementations of Java and Scala subsets [9] show that not only is it possible to elegantly declare and type-check these languages with scope graphs, but also highlight the possibility of resolution of classes/objects (module-like structures) and their corresponding imports.

Nevertheless, there are still open questions regarding modules and their imports. While there exists a scope graph representation [2] of the LM(R) [2] proof-of-concept language (elaborated upon in Section 2.1), it is not executable in practice, as the type checker "gets stuck" [9, p.24] with regard to modules and imports [9]. What differentiates LM from Java and Scala is that its imports are relative and unordered, with every import also being a glob import [9]. An example of a real-world language that exhibits these properties is Rust [9].

This thesis answers the question of *if and how scope graphs constructed by a phased Haskell library can be used to type-check a language with relative, unordered, and glob imports*. Since these semantics are present in at least one real-world language, Rust, it is salient to investigate to what extent scope graphs can live up to their premise of supporting name-binding in the real-world [14]. Concretely, LM will be used to research this, though the approach is language agnostic and can be applied to for example Rust. In the MiniStatix case, stuckness arises in conjunction with query scheduling [14]. In contrast, this thesis will consider an alternative phased approach, where the scope graph will be built up incrementally over multiple passes of the AST[1], and queries are manually scheduled.

[1] Abstract Syntax Tree

(a) Glob imports.

```
module A {
  def x = 1
  def y = 2
}
module M {
  import A
  def z = x+y
}
```

(b) Rel. imports.

```
module A {
  module B {
    def x = 19
  }
}
module M {
  import A
  import B
  def y = x
}
```

(c) Unord. imports.

```
module A {
  module B {
    def x = 19
  }
}
module M {
  import B
  import A
  def y = x
}
```

Figure 1: Code snippets highlighting LM import semantics.

In summary, the contributions of this thesis are twofold. For one, we present a five-step stratified approach to type-checking LM, which is evaluated against test cases. On top of that, we introduce two new primitives: BF-traversal and multi-origin querying.

This thesis begins with a more in-depth insight into the problem and scope graph resolution presented in Section 2. This thesis' contribution, Phased type checking, is explicitly highlighted in Section 3. Analysis through the use of test cases is elaborated upon and discussed in Section 4. Section 5 provides further related literature. Finally, Section 6 concludes the research with further recommendations.

2 Problem Description

This section discusses the problems surrounding type-checking with imports. To do so, a background of LM is provided in Section 2.1, and scope graphs are explained with reference to LM in Section 2.2. The problem itself is elaborated upon in 2.3.

2.1 LM

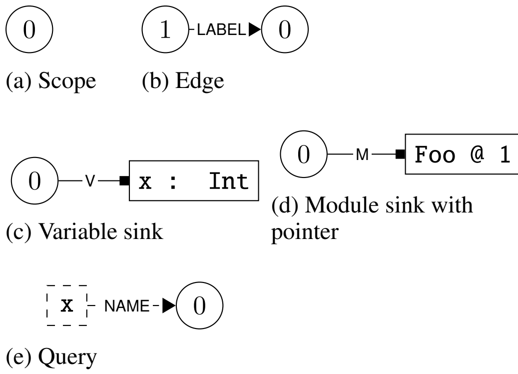
This thesis considers a variation of the proof-of-concept language LM (Language with Modules) by Neron et al. [8]. LM closely resembles LMR as described by van Antwerpen et al. [2], though it omits records. The focus of LM is such that the module/import system can be used and example programs can be run. The full grammar is shown in Appendix B, Figure 15.

LM's imports are all glob imports. This is shown in Figure 1a, whereby all declarations of module A are visible directly in M as A is imported. Note that this definition of LM does not support transitive imports; a module importing M would not be able to see x or y.

Furthermore, imports are unordered and relative, as shown by Figures 1b and 1c. Due to relative imports, B can be imported without requiring the reference $A.B$. Simultaneously, whether the parent or child module is imported first is irrelevant, since imports are unordered.

We will consider the flavor of LM that consists of the following (non-)features [8]:

1. Qualified names: declarations can be referenced through module names via dot notation.



(a) Scope (b) Edge (c) Variable sink (d) Module sink with pointer (e) Query

Figure 2: Scope graph notation for LM.

2. Glob imports: all declarations in an imported module are visible in the module that imports it.
3. Non-transitive imports: declarations visible in imports of imports are not visible.
4. Cyclic imports: both self and (indirect) cyclical imports are supported.
5. Arbitrary nesting: modules can recursively nest each other, which can be brought into scope by the above rules.

2.2 Scope Graphs

Scope graphs are directed graphs in which every scope is represented as a node. Sinks attached to nodes provide name-binding information. In the case of LM, sinks are variable declarations denoting certain types and type variables or module declarations. When type-checking, a query for a specific name is run that originates at a specific scope and can take edges based on a provided regular expression. Across different papers, scope graph notation can differ [14]. This thesis will consider (numbered) scope graph concepts to be denoted as portrayed in Figure 2.

Two distinct sinks are required for variables and modules, since they hold different data. Variable sinks are declarations used for name binding; they specify a variable and its associated type. In Figure 2c, the variable x has the integer type. Module sinks are declarations that contain module metadata. In Figure 2d, the module *Foo* has its scope body in scope 1. There are four labels considered:

1. **P** denotes lexical parent
2. **V** is used exclusively for variable sink edges
3. **M** is used exclusively for module sink edges
4. **I** denotes an import edge

The scope graphs for the program in Figure 1a can be found in Figure 3. Note that due to the unordered imports, the scope graphs are identical for programs 1b and 1c. Such a scope graph can be found in Figure 4. Although queries are also visualized in these examples, they will be elaborated upon further in Section 2.3.

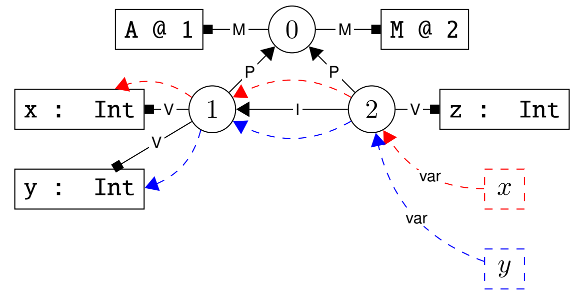


Figure 3: Scope graph of the glob import example in Figure 1a.

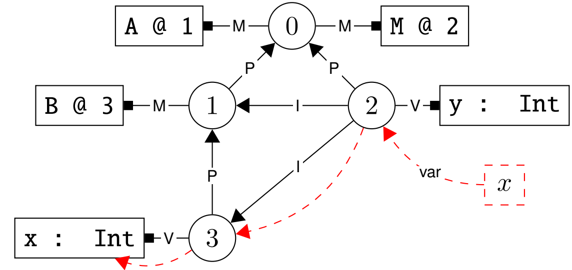


Figure 4: Scope graph of the relative/unordered import example in Figures 1b and 1c.

2.3 Resolution and Monotonicity

MiniStatix cannot perform import resolution for LM programs with imports due to the way it delays queries [9]. At its core, querying for imports poses *monotonicity issues*. This is explained in more depth in the below subsections.

Queries

Queries query paths that conform to particular edges and order these paths based on some priority (for example, shortest path). To specify which edges can be taken, regular expressions are used. In Figures 3 and 4, queries are visualized in red and blue. Taking as an example the resolution of x in Figure 4, a query is executed from scope 2 for all sinks whose path labels match the regular expression **P*I?V**. Some transitive-inclusive implementations in literature consider **P*I*V** [14], though transitive imports were omitted for simplicity as highlighted in 2.1. This yields a singular path, that is shown in red. If there had been multiple results, the highest priority edge [2] (elaborated upon in Section 3.3) would have been selected. If establishing a priority is not possible, then the program is ambiguous.

Monotonicity

Query results must remain valid throughout the entire lifecycle of type-checking. Scope graph queries must be *monotone* [9]: if a scope has been queried for a path containing label X, an outgoing edge of label X may not be placed. Concerning this, the semantics of LM's modules and imports pose a challenge.

An example that highlights the usefulness of monotonicity can be seen in Figure 5a (and its corresponding scope graph in Figure 5b). A query Q_1 made for x *before* the import edge (in red) is placed would result in the x of scope 1. However,

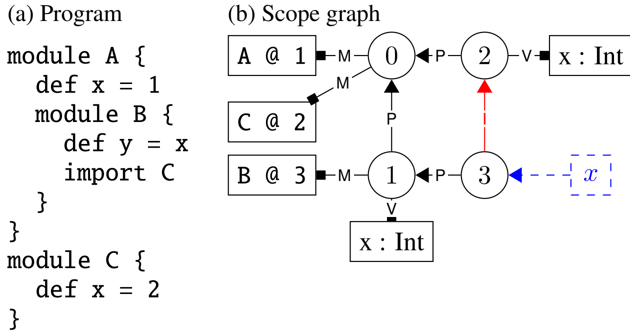


Figure 5: Example program and its scope graph that could generate monotonicity errors.

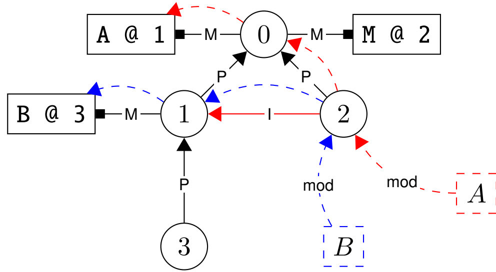


Figure 6: Partial scope graph of the relative/unordered import example in Figures 1b and 1c.

according to LM edge priorities [2], when the scope graph is fully constructed, it should prefer **I** over **P** and resolve to the x of scope 3. Thereby, Q_1 would no longer be valid. Placing this import edge after Q_1 is therefore forbidden to ensure monotonicity of the scope graph.

Relating this concept to LM: consider the partial scope graph in Figure 6. Here, module M in scope 2 would like to import modules A and B by placing an import edge to both. For a **P*I?M** query (the query to find module sinks) to find B, we need to place an import edge to the scope of A. This is shown as a red query, placing the red edge in Figure 6. Now the previously executed query of resolving A via the red path is (potentially) no longer valid, since the introduction of the **I** edge may (but not in this example) yield a different most-suitable path. More precisely, the aforementioned principle of monotonicity is violated.

MiniStatix Scheduling

MiniStatix attempts to avoid monotonicity errors by deferring queries to later points [9]. However, in the case of imports, this yields a contradiction. To find module B, we must query for module A. On the contrary, the query of module A is delayed as the import edge of B would need to be placed first. Consequently, MiniStatix enters a deadlock it cannot recover from. So, while the mathematical model for LM's scope graph exists, it currently cannot be constructed for LM, thereby violating the MiniStatix goal of being *executable*.

3 Phased Type-Checking

In order to avoid monotonicity errors, queries need to be executed only when strictly necessary and scheduled in a way such that new edges do not invalidate existing queries. This work presents a five-step procedure from creating the scope graph to type-checking its members. An import resolution algorithm of LM that does not violate monotonicity is derived. These five steps are as follows:

1. Constructing a module hierarchy.
2. Constructing a scope graph consisting of scopes and module sinks.
3. Iteratively resolving imports and placing import edges in the scope graph.
4. Adding all declarations of all modules to the scope graph.
5. Type-checking the bodies of all declarations with respect to the scope graph.

3.1 Phase 1: Creating a Module Hierarchy

Modules in LM can be considered a hierarchy: arbitrary nesting of modules is possible. The initial representation of the AST of LM programs can be found in Appendix B. However, it is not a convenient module-centric representation of the program.

Instead, an alternate representation is created, where modules have a list of imports (`[LModule]`), a list of recursive child modules, and a list of declarations (`LDecl`). The specific definitions of these constructor parameters can be found in Appendix B.

```
data ModTree
  = Anon [LModule] [ModTree] [LDecl]
  | Named String [LModule] [ModTree] [LDecl]
```

In this data structure, the `Anon` constructor is the top-level anonymous module. Similarly, the `Named` constructor is a module with a provided name.

3.2 Phase 2: Constructing Module Scope Graphs

The module tree obtained in phase needs to be turned into a scope graph. In order to accomplish this, the above module hierarchy is traversed, and a scope is created for every module. Child module scopes are connected to parent module scopes via a lexical parent **P** edge – this is such that declarations made in the parent are reachable by query in the child. For example, the scope graph from Figure 6 at this point can be shown in Figure 7.

Furthermore, we create an extension of the module hierarchy. This extension considers the scope in which each module exists, but is otherwise identical to the previous representation of the tree (see Section 3.1). Such functionality is convenient in the next three phases. It is represented as follows:

```
data AMT
  = AAnon Sc [LModule] [AMT] [LDecl]
  | ANamed Sc String [LModule] [AMT] [LDecl]
```

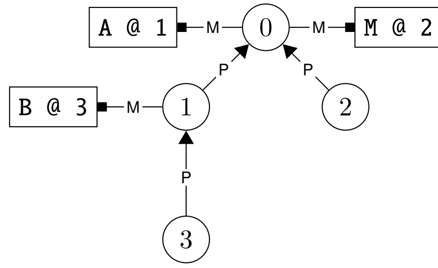


Figure 7: Scope graph containing just modules after phase 2.

Algorithm 1 Import resolution algorithm.

Require: I, F, m
  while $I \neq \emptyset$ **do**
    $(U, A) \leftarrow$ partition I into unique and shadowing names
    $f \leftarrow$ poll F
    if f is null **then**
      error *"not all imports could be resolved"*
    end if
    $S \leftarrow \emptyset$
    for $u \in U$ **do**
      $r \leftarrow$ query for u from f
      $S \leftarrow S \cup r$
    end for
    $S' \leftarrow \emptyset$
    for $a \in A$ **do**
      $r \leftarrow$ multiple queries for a from all in $\{f\} \cup S$
      $r' \leftarrow$ lowest cost path via label from r or $\emptyset$
      $S' \leftarrow S' \cup r'$
    end for
    $R \leftarrow S \cup S'$
    for $r \in R$ **do**
      place import edge from m to r
    end for
    $F \leftarrow F \cup R$
    $I \leftarrow$ remove modules in R from I
  end while

3.3 Phase 3: Resolving Imports

The module hierarchy (the AMT of Section 3.2) is first flattened into a list using breadth-first traversal. This allows for the unordered imports to be given a concrete ordering: if *A* occurs before *B* in the list, the resolution of *A* is attempted before that of *B*. As a result, parents' modules are always resolved before child modules when both of these are imported, as can be with LM.

The resolution algorithm (Algorithm 1) resolves imports for a single module m, given I imports and a queue F that initially consists of the module itself. One caveat is that, as aforementioned, the order of I must correspond to the order of modules in the flattened hierarchy. All queries in this algorithm use the regular expression **P*I?M**.

The core intuition behind the algorithm is to exploit the module hierarchy. If module *B* comes after *A* in the hierarchy, then the resolution of *A* is attempted before that of *B*. Therefore, if the resolution of *B* fails, this import is invalid, as resolution of all other imports of modules that could possibly be parents or siblings of *B* has already occurred. Furthermore, partitioning is also required, as resolution for shadowing module names is non-trivial.

(a) Unordered example.

```
module A {
  module B {
    def x = 19
  }
}
module M {
  import B
  import A
  def y = x
}
```

(b) Shadowing example.

```
module A {
  module B {}
}
module B {}
module M {
  import B
  import C
}
module C {
    module B {}
}
```

Figure 8: Example programs to highlight import resolution.

1. $I = \{A, B\},\ F = \{M\}$
2. $I \neq \emptyset$, pop M from the frontier.
   (a) Try to resolve A, query yields $\{A\}$.
   (b) Try to resolve B, query yields $\emptyset$.
3. $S = \{A\},\ R = S$ (no shadowing imports, these are not considered).
4. Place edges from M to each in R, an edge is placed to A.
5. $F = F \cup R = \{A\}$
6. $I = I \setminus R = \{B\}$
7. $I \neq \emptyset$, pop A from the frontier.
8. Try to resolve B, query yields $\{A.B\}$.
9. $S = \{A.B\},\ R = S$
10. Place edges from M to each in R, an edge is placed to $A.B$.
11. $F = F \cup R = \{A.B\}$
12. $I = I \setminus R = \emptyset$
13. Iteration stops due to $I = \emptyset$.

Figure 9: Walk-through of program 8a.

Example Resolution

To illustrate this algorithm, two programs are shown in Figure 8. Both of these programs only have a single module with imports for simplicity.

The walk-through of the algorithms can be found in Figures 9 and 10. Note that to enhance readability, modules are referred by their absolute paths with respect to the global scope.

Modifications

For the algorithm to work, it is assumed to be possible to query from multiple origins. This is simple to implement programatically, but not part of the scope graph primitives. Combined with breadth-first traversal (queue frontier) of the scope graph, this enables the implementation of the algorithm.

1. $I = \{C, B\}$, $F = \{M\}$ (Note: this could also be $\{B, C\}$).)
2. Split I into $U = \{C\}$ and $A = \{B\}$.
3. $I \neq \emptyset$, pop M from the frontier.
4. Try to resolve C, query yields $\{C\}$.
5. $S = \{C\}$
6. Try to resolve B via C or M, query yields $B, C.B$.
7. Select the correct B via priority path, $S' = \{C.B\}$.
8. $R = S \cup S'$
9. Place edges from M to reach in R, an edge is placed to C and $C.B$.
10. $F = F \cup R = \{C, C.B\}$
11. $I = I \setminus R = \emptyset$
12. Iteration stops due to $I = \emptyset$.

Figure 10: Walk-through of program 8b.

$$\mathbf{M} < \mathbf{P},\ \mathbf{M} < \mathbf{I},\ \mathbf{V} < \mathbf{P},\ \mathbf{V} < \mathbf{I},\ \mathbf{I} < \mathbf{P}$$

Figure 11: Edge priorities.

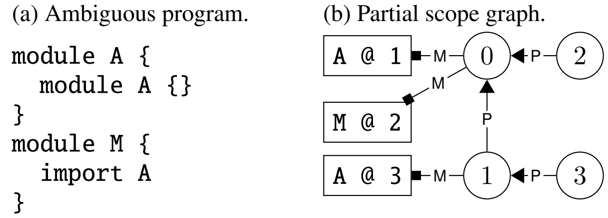


Figure 12: Example ambiguous program and partial scope graph.

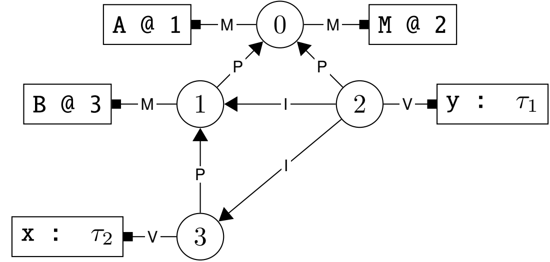


Figure 13: Scope graph containing declarations after phase 4.

Unique Module Resolution

To reiterate, it is essential to attempt the resolution of imports in the correct hierarchical order: a parent needs to be imported before its child can be resolved. The frontier extends with every resolved import. Hence, the discovery of modules via querying from the frontier results in the frontier being populated with modules in the same order as they appear in the flattened module list. Since imports are unique, query results may return zero or one module. There is no need to select a module according to priorities since queries will never yield duplicate results here.

Shadowing Module Resolution

An issue with duplicate module names is to not resolve the shallowest-possible module in the module hierarchy, as is the case if the algorithm is not adapted to explicitly handle shadowed names. To circumvent this problem, the resolution is performed at every iteration via multiple queries from the frontier and non-ambiguous imports. This will yield the shallowest-possible module, as well as other reachable modules.

When multi-querying, multiple results over multiple queries may be yielded. Prioritization of all aggregated results must be performed, which is why the shallowest-possible module is not selected. This prioritization is done via minimization of edge labels using ordering rules [2]. If no comparison can be made for the first edge, the second edge is considered. If no more edges are available for comparison for one path, the shorter path takes priority. If two paths are identical, the first one is chosen as a tiebreaker. These priorities are as shown in Figure 11 (n.b.: the smallest value has the highest priority).

Ambiguity

The program in Figure 12 is ambiguous. Without the import edge between scopes 2 and 1, a query for module *B* resolves in 2. However, when the import edge is placed, it will resolve in 3, as resolution prefers imports over lexical parents. This is a contradiction, and no scope graph exists for such a program.

The algorithm cannot detect ambiguity during resolution. Instead, when an import edge from m to m' for the name *X* is placed during resolution, this is memorized. Then, after all imports have been resolved, every module is queried for the names of all its imports again (via **P*I?M**). If the query result of this query differs from the edge that was placed during resolution, the program is rejected for ambiguity.

Termination

Invalid imports should be rejected – like MiniStatix, this algorithm must be resilient [14]. Part of this is termination, this is achieved implicitly in Algorithm 1 by exhausting the frontier. The frontier is only ever populated with valid query results, so eventually once all valid imports are resolved, it will become empty, and an error is raised. Since the frontier is used for both unique and shadowing names, the algorithm will terminate for both.

3.4 Phase 4: Adding Declarations

Before type-checking of declaration bodies is possible, all declarations need to be added with a type variable. Constructing the declarations first allows for forward referencing, similar to how all modules must exist before import edges may be drawn. In this phase, the annotated module tree is traversed, and for every module's scope, its `defs` are added to the scope graph with a type variable. The partial scope graph at this point for Figures 1b and 4 is shown in Figure 13.

3.5 Phase 5: Type-Checking

As the last phase, the body of declarations (i.e. expressions) is type-checked using inference. Specifically, equality constraints are generated (with respect to the type variables laid out in the previous phase) that are unified. The rules by which

$[\![i]\!]^{expr}_{s,t} := t \equiv Int$

$[\![\textbf{true}]\!]^{expr}_{s,t} := t \equiv Bool$

$[\![\textbf{false}]\!]^{expr}_{s,t} := t \equiv Bool$

$[\![x_i]\!]^{expr}_{s,t} := x_i^{\mathbf{R}} \rightarrow s \wedge x_i^{\mathbf{R}} \mapsto \delta \wedge \delta : t$

$$[\![e_1 + e_2]\!]^{expr}_{s,t} := t \equiv Int \wedge \tau_1 \equiv Int \wedge \tau_2 \equiv Int \\ \wedge [\![e_1]\!]^{expr}_{s,\tau_1} \wedge [\![e_2]\!]^{expr}_{s,\tau_2}$$

$$[\![e_1 - e_2]\!]^{expr}_{s,t} := t \equiv Int \wedge \tau_1 \equiv Int \wedge \tau_2 \equiv Int \\ \wedge [\![e_1]\!]^{expr}_{s,\tau_1} \wedge [\![e_2]\!]^{expr}_{s,\tau_2}$$

$$[\![e_1 * e_2]\!]^{expr}_{s,t} := t \equiv Int \wedge \tau_1 \equiv Int \wedge \tau_2 \equiv Int \\ \wedge [\![e_1]\!]^{expr}_{s,\tau_1} \wedge [\![e_2]\!]^{expr}_{s,\tau_2}$$

$$[\![e_1 == e_2]\!]^{expr}_{s,t} := t \equiv Bool \wedge \tau_1 \equiv Int \wedge \tau_2 \equiv Int \\ \wedge [\![e_1]\!]^{expr}_{s,\tau_1} \wedge [\![e_2]\!]^{expr}_{s,\tau_2}$$

$$[\![\textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_f]\!]^{expr}_{s,t} := t \equiv \tau_t \wedge \tau_c \equiv Bool \wedge \tau_t \equiv \tau_f \\ \wedge [\![e_c]\!]^{expr}_{s,\tau_c} \wedge [\![e_t]\!]^{expr}_{s,\tau_t} \wedge [\![e_f]\!]^{expr}_{s,\tau_f}$$

$$[\![\textbf{fun } (x_i : \tau_1) \{ \mathit{body} \}]\!]^{expr}_{s,t} := t \equiv \tau_1 \rightarrow \tau_2 \wedge s' \xrightarrow{\mathbf{P}} s \\ \wedge s' \xrightarrow{\mathbf{V}} x_i^{\mathbf{V}} \wedge x_i^{\mathbf{V}} : \tau_1 \wedge [\![\mathit{body}]\!]^{expr}_{s',\tau_2}$$

$[\![e_f\ e_a]\!]^{expr}_{s,t} := \tau_f = \tau_a \rightarrow t \wedge [\![e_f]\!]^{expr}_{s,\tau_f} \wedge [\![e_a]\!]^{expr}_{s,\tau_a}$

$$[\![\textbf{letrec } (x_i = e) \textbf{ in } \mathit{body}]\!]^{expr}_{s,t} := t \equiv \tau_b \wedge s' \xrightarrow{\mathbf{P}} s \\ \wedge s' \xrightarrow{V} x_i^{\mathbf{V}} \wedge x_i^{\mathbf{V}} : \tau_a \wedge [\![\mathit{body}]\!]^{e}_{s',\tau_a} \wedge [\![\mathit{body}]\!]^{expr}_{s',\tau_b}$$

Figure 14: Typing constraint generation and scope graph modifications, based on that in [2].

one can type-check LM are laid out in Figure 14, which are largely based on the rules presented by van Antwerpen, et al. [2]. Every expression gets a type variable t and is then type-checked with respect to the current scope s. Note that parentheses have been omitted, as these solely influence parser rules. The notation $[\![x[\![^{expr}_{s,t}$ denotes the behavior an expression x has with respect to the scope s it is type-checked in, and the type variable t it should equate to. $x_i^{\mathbf{R}} \rightarrow s \wedge x_i^{\mathbf{R}} \mapsto \delta$ is a query using **P*I?V** in scope s that resolves to a δ. This uses the prioritization rules of Figure 11 in Section 3.3.

4 Evaluation and Discussion

To evaluate the performance of phased type-checking, test cases are used, a standard practice in programming language research. Few test cases can already highlight and test for the interesting semantics of LM. All the code and test cases mentioned in this section are available[2] and distributed alongside this thesis.

We also discuss this thesis' contribution with respect to performance, the new primitive extensions, and declarativity/feature extensibility. Reusability and limitations are further emphasised.

[2]https://doi.org/10.4121/288296b9-fcdf-4960-bab4-8aee3a46927c

Table 1: Properties of the test cases (overlap is possible).

Property	**Amount**
General Behaviour	8
Qualitative References	3
Qualitative Imports	1
Relative Imports	3
Unordered Imports	2
Duplicate Module Names	2
Ambiguity	1
Self Imports	2
Cyclical Imports	4

4.1 Test Cases

The test cases shown by Rouvoet et al. [9] compose the core of this evaluation. These are augmented by further tests derived specifically for this thesis, forming a total of **26** cases. Essentially, they aim to cover a breadth of LMR semantics and corner cases such that in the absence of formal verification, some conclusions can be drawn. This can be demonstrated in Table 1, which highlights how many test cases test certain LM properties. In particular, specific import types (both in isolation and combination) are focused on. Since an LM parser is out of the scope of this thesis, the test cases were converted to annotated terms [13].

4.2 Implementation

The implementation of the type checker (complete in accordance with the algorithm of Section 3) uses the *Phased*[3] Haskell template. This library provides a monadic way to construct and query scope graphs based on the effect handler theory laid out by Bach Poulsen and van der Rest [3]. Furthermore, it provides automatic unification, which is required, as the type-checker generates constraints.

This implementation is in theory cross-platform compatible, though not all shells may support the characters used in the visualization of scope graphs. We used *GHC 8.10.7/Cabal 3.6.2.0* in a Unix development environment.

4.3 Results

A confusion matrix of test results is shown in Table 2. The Phased implementation correctly type-checks 25 test cases,

[3]https://github.com/MetaBorgCube/scope-graph-scheduling-bsc-template

Table 2: Confusion matrix of test results.

		True behaviour		
		Accept	Reject	Total
Impl.	Accept	20	0	20
	Reject	1	5	6
	Total	21	5	26

with their scope graphs being verified empirically. A singular false negative arises from the implementation as per Section 4.2. The test case that fails (see Appendix C) has to do with shadowing/duplicate module names and is rejected by the ambiguity detection.

We learn that it is possible to type-check LM, to some extent, using a stratified approach. Many edge cases and import semantics are covered by tests, which pass. Ergo, the implementation can be considered partially correct.

While implementation specific, it is worth elaborating upon the Phased Haskell library. It was possible to integrate auxiliary algorithms and strategies for LM into the library. Additionally, it was also possible to extend the library primitives themselves to provide a new query operation that returns a full path as a result, not just a sink. This flexibility and feature extensibility was found to be a highlight of the library.

4.4 Performance

The Phased approach provides a huge benefit over that of MiniStatix: it halts on all the test cases and provides correct judgment for the vast majority of them. Therefore, it is now possible to use scope graphs to check at least parts of the module system of LM. Concretely concerning LM, this now means that scope graphs can be considered (partially) executable – one of its target characteristics. Nonetheless, it should be noted that with failing test cases, this implementation cannot possibly be fully correct.

Furthermore, computational performance should be touched upon. Type-checker information is crucial to IDE integration [15], and must often be delivered swiftly for a good editor experience [15]. The Phased implementation does not consider efficient algorithms or code as it intends to be a proof of concept.

4.5 Primitive Extensions

The import resolution algorithm described in Section 3 relies on two new concepts to function: breadth-first traversal and queries of multiple origins. These concepts could be converted to primitives, such that the reusability of this research increases. Reusability is further elaborated on in Section 4.7.

Breadth-First Traversal

Let $\mathcal{G}$ denote the scope graph and $\mathcal{S}(\mathcal{G})$ the scopes within it. Then, traversal can be defined by the triplet $\langle S_0, c, v \rangle$, whereby:

- $S_0 \subseteq \mathcal{S}(\mathcal{G})$ denotes the initial scopes in the frontier.
- $c : F \mapsto \{\mathbb{T}, \mathbb{F}\}$ is a function that takes the current frontier and decides whether or not the next iteration will occur. An iteration consists of polling the frontier and performing an action.
- $v : s \mapsto S$ is a function, given the polled head of the frontier, that will perform an action/side effect, and provide the next scopes to add to the frontier.

Since side effects can occur, monotonicity must be explicitly checked for and may not be violated. It should be noted that v may require additional information – in the context of Phased type-checking, this would be the set of imports that still need to be resolved. If this traversal is implemented in a pure language, it may be required to introduce a form of state (in the Phased example, encoding the state within the `Free` monad) for c and/or v.

Multi-Origin Querying

Presently, querying requires an origin, a regular expression, a function to check if a sink matches a condition, and a function to select the most appropriate path and its result. Let $\mathcal{Q}(x)$ denote a function to query a scope x via regular expression re with matcher $m : s \mapsto \{\mathbb{T}, \mathbb{F}\}$. It returns a set of tuples $\langle p, r \rangle$ where p is the resolved path and r the resolved sink. Then, the multi-origin query result given a set of n origins as O may be defined as:

$$R = \bigcup_{i=1}^{n} \mathcal{Q}(O_i) \tag{1}$$

Consequently, the final query result gives the sink for which the path is minimized:

$$\pi_2(\operatorname*{argmin}_{r \in R} \pi_1(r)) \tag{2}$$

Which assumes $|R| > 0$, the existence of order given a path, and denotes π_i as a projection of the i-th tuple element.

4.6 Declarativity and Feature Extensibility

The current Phased implementation is not declarative, as a manual implementation of the resolution algorithm is required. In that sense, MiniStatix is much more declarable (and concise, with only 263 lines [9] compared to > 500), although it is unable to resolve imports. Similarly, although subjective, MiniStatix is much less verbose and provides a concise and elegant way to define type checkers. However, if multi-origin querying and in particular breadth-first traversal were to become scope graph primitives, the Phased approach would be much more declarative.

In terms of feature extensibility, the ability to manually schedule queries provides a powerful tool to Phased that MiniStatix does not have. Indeed, this is one of the concepts required to allow import resolution in the first place. Manual control provides superior extensibility than the convenience of automatic scheduling in MiniStatix. Furthermore, the extensibility of the Phased library itself makes this solution very extensible.

4.7 Reusability

It is important to discuss the reasons for and against the reusability of the work laid out in this thesis. While relative, unordered and glob imports as in LM are not commonly used together [9], there are still applications of the Phased approach. Particularly, relative and glob imports are prone to causing monotonicity issues.

We can see such properties in other languages, including but not limited to Ruby and C++. Both Ruby's `modules` and C++'s `namespaces` can be nested arbitrarily, and be "imported" (brought into scope) with `include` and `using namespace`, respectively (see Appendix D). For both of these languages and unlike LM, the imports need to be ordered and are transitive. To resolve child compilation units, the parent unit needs to be resolved first. Consequently, these languages would run into the same monotonicity error as LM as

described in Section 2.3. Thus, the import resolution portion and new scope graph primitives of the Phased approach are directly applicable to Ruby and C++.

On the contrary, the approach laid out in this thesis is neither simple to understand nor efficiently implemented. Taking Ruby as an example, the type-checker laid out by An et al. [1] is much more elegant, though it does not take advantage of scope graphs.

4.8 Limitations

An obvious limitation is that a test case fails (see Appendix C). This program is an example of behavior that should be supported. Unfortunately, this behavior cannot (easily) be fixed without it affecting other test cases. Disabling ambiguity detection could potentially pass this test case, but consequently, an ambiguous test would not be rejected. Instead of a false negative, the test suite would then report a false positive.

The ambiguity detection is very naive. It guarantees neither that ambiguous programs are caught, nor that all imports rejected are truly ambiguous (as can be seen by the failing test case). Improvement of the ambiguity checker poses a larger challenge and should be considered as a separate and future work.

Even considering a correct ambiguity detection, it remains an open question whether the Phased approach is truly correct. It could be that there is more deviating behaviour that was not unearthed by the test cases. Until a proof of correctness is offered, no assertions on correctness can be made.

5 Related Work

A comprehensive summary of scope graph research and its history has been curated in [14]. Both LM, scope graphs, and the combination of both have been discussed extensively in previous work. LM first appears by Neron et al. [8], and is extended with records (to form LMR) by van Antwerpen et al. [2]. Furthermore, it is included as a case study by Rouvoet et al. [9], the paper whose results inspired this thesis. Although these serve as an excellent theoretical basis on which we based some of my research, they are mathematical models. The creation and execution of these scope graphs based on input programs are delegated to meta-languages, where there are significant challenges.

The first meta-language to support scope graphs was NaBL2 [14], though it was unable to express scope graphs where the presence of edges depends on resolution [9]. Nonetheless, an implementation of LM with NaBL2 did not pose any issues and became complete, as highlighted in [2], and traces of this can be found online [11]. However, requiring the presence of edges is limited [7], and it further would not work with the Phased library's scope-graph representation. Consequently, Statix and MiniStatix were developed as a successor of NaBL2 [10]. Their flaw, scheduling of queries not halting with LM, left a knowledge gap and the opportunity to explore whether type-checking in phases resolves these issues.

A formalization using environments of a type-checker for a module system like that of Standard ML is provided by Leroy [6]. Similar to LM, the Standard ML module system supports arbitrarily nested modules. Furthermore, declarations within modules can be accessed using the `open` directive [5], effectively emulating LM's glob imports.

6 Conclusions and Future Work

To conclude, scope graphs allow for name-binding that can be used for type-checking applications [14]. Given the proof-of-concept language LM with import semantics similar to Rust, an existing type-checker written using MiniStatix does not halt when resolving imports [9]. This is problematic as scope graphs should be executable [14]. As such, the question was if scope graphs constructed by a phased Haskell library can be used to type-check a language with relative, unordered, and glob imports, given that this could not be achieved in a non-stratified approach.

LM can be represented in a scope graph using variable and module sinks, module/variable declaration, import, and parent edges. By using auxiliary data structures and manual query scheduling over many AST passes, it is possible to type-check LM examples with scope graphs, thereby making it possible to type-check relative, unordered, and glob imports in scope graphs. Precisely, a five-step Phased approach is discussed and implemented in Haskell. It consists of creating a module ordering, adding modules to the scope graph, import resolution, adding declarations to the scope graph, and type-checking declaration bodies. This passes all but one of the good-weather and bad-weather test cases and always halts whereas MiniStatix does not [9].

Since relative/unordered imports are found in other languages, and it is important to consider reusability, two improvements to scope graph primitives are suggested. Both of these stem from the Phased implementation and have been generalized for scope graph applicability. Firstly, a way to traverse a scope graph in a breadth-first fashion and perform side effects during the traversal, such as placing edges. Secondly, multi-origin querying, whereby a query can be executed from multiple origins and the results combined into one. Integrating such features would further improve the declarativity of the Phased type-checker, which is currently not declarative and much less concise compared to the MiniStatix one.

While this thesis presents an introduction to type-checking LM with the Phased approach, there are still limitations and various areas that can and further should be researched. First and foremost, this thesis presents a partially-correct implementation with no proof of correctness; a logical next step is to resolve failing test cases (and potentially derive a fully-correct ambiguity detection algorithm) and ultimately prove an implementation to be correct. In parallel, it could be investigated how the Phased approach performs when applied to Ruby or C++. Flavors of LM exist that contain transitive imports, such as the one laid out by van Antwerpen et al. in [2]. The applicability of the Phased approach to transitive imports and querying could be further researched. Lastly, the performance of this algorithm could be investigated on real-world programs with many imports due to this not being taken into consideration in this thesis.

Acknowledgements

First and foremost, I would like to express my gratitude towards my responsible professor Dr. Casper Bach Poulsen and Aron Zwaan, who patiently guided me and answered my many, many questions. Your feedback was not only invaluable in the development of this thesis but helped me develop academic soft skills in a broader context. I would like to also acknowledge the instructors of Academic Communication Skills and Responsible Research for conveying their knowledge.

I am deeply appreciative of my family and friends who have so kindly supported me throughout this thesis. I would like to thank Jan, Philip and Eloise for diligently highlighting my typos and Jeroen for helping me figure out C++. Lastly, I would like to acknowledge the late Ollie, who may never have grasped how to behave like a normal cat, let alone scope graph theory, but provided me with purrs and moral support.

References

[1] J. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. *SIGPLAN Not.*, 46(1):459–472, Jan 2011.

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

[3] C. Bach Poulsen and C. van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL), Jan. 2023.

[4] M. Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[5] D. Gratzer. A crash course on ML modules. https://web.archive.org/web/20230602041321/https://jozefg.bitbucket.io/posts/2015-01-08-modules.html, Jan 2015. Accessed 22.06.2023.

[6] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[7] P. D. Mosses. A component-based formal language workbench. *Electronic Proceedings in Theoretical Computer Science*, 310:29–34, Dec. 2019.

[8] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *Programming Languages and Systems*, pages 205–231, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] A. Rouvoet, H. van Antwerpen, C. Bach Poulsen, R. Krebbers, and E. Visser. Knowing when to ask: Sound scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[10] H. van Antwerpen, C. Bach Poulsen, A. Rouvoet, and E. Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018.

[11] Hendrik van Antwerpen. nabl. https://github.com/metaborg/nabl/blob/4d06d2cf2519d461288ad7e7fb31575da1ea24af/nabl2.test/testsuites/name-resolution/tests.spt#L141-L163, 2019. Accessed 25.06.2023.

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

[13] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

[14] A. Zwaan and H. van Antwerpen. Scope Graphs: The Story so Far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium (EVCS 2023)*, volume 109 of *Open Access Series in Informatics (OASIcs)*, pages 32:1–32:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[15] A. Zwaan, H. van Antwerpen, and E. Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), Oct. 2022.

A Responsible Research

When performing research, it is critical to consider the ethical implications. Separatism and the tripartite model dictate that the scientists/engineers are only responsible for the technical creations of products [12]. In contrast to such stigma, it is important to consider the big-picture implications of this research.

For one, it is necessary to consider validity. Specifically, the impact that false positive and false negative findings have, although neither was observed during research. It should be noted that the software created in this thesis is not directly applicable in the real world, rather it serves as a foundation that real-world relevant programming languages can build upon. False positives would reject the program of a user even though it is semantically correct. Such a situation is frustrating, but aside from decreased productivity has no larger-scale impact on society. However, false negatives delay the discovery of bugs. Although the direct real-world applicability of this research is limited, bugs can cause devastating damages (such as Ariane 5 [4]). Nonetheless, the broader analysis of the impact of bugs is outside the scope of this reflection, and something that should be researched further.

In general, the research performed in this thesis does not have a large-scale impact on individuals or society. Type-checkers are tools that aim to provide certain guarantees or help to the programmer. Consequently, research in this area contributes positively towards programmer productivity and it could be argued that this has a marginal impact on societal output.

On the contrary, programs can be used for malicious purposes, such as viruses or other forms of malware. While the abstract nature of this research means it will most likely not aid in the creation or augmentation of malicious software, the possibility is always there. Furthermore, job security could

be impacted by a sharp but incredibly unlikely increase in productivity.

Beyond ethics, important points of consideration are reproducibility (reliability) and transparency. In terms of reproducibility, the publication of source code, test cases, and technology baseline ensures that this thesis is reproducible. Furthermore, all assumptions and omissions have been explicitly highlighted and justified, ensuring that the research is transparent. Should this thesis contain flaws in the code, this can easily be fixed due to its distribution and permissive license. Lastly, citations are used to provide further insight into sources of information, such that this thesis' context and research can be reproduced by others.

B Representation

Figure 15 shows the grammar of LM. Similarly, Figure 16 shows the AST representation of the LM variant described by the grammar in Figure 15, directly after being extracted from ATerms.

prog ::= decl*

decl ::= **module** ident { decl* }
| **import** modref
| **def** ident = expr

expr ::= int
| **true**
| **false**
| varref
| expr + expr
| expr - expr
| expr * expr
| expr == expr
| **if** expr **then** expr **else** expr
| **fun** (ident : type) { expr }
| expr expr
| **letrec** ident = expr **in** expr
| (expr)

modref ::= modref . ident
| ident

varref ::= modref . ident
| ident

type ::= Int
| Bool
| type → type

Figure 15: BNF grammar for LM, a subset of the LMR grammar by Van Antwerpen et al. [2].

```
type LProg = [LDecl]
data LDecl
  = LMod String [LDecl]
  | LImport LModule
  | LDef String LExp
data LModule
  = LMLiteral String
  | LMNested LModule String
data LFDecl
  = LFDecl String LType
data LType
  = LInt
  | LBool
  | LFn LType LType
data LExp
  = Num Int
  | Tru
  | Fls
  | Id LIdent
  | Plus LExp LExp
  | Minus LExp LExp
  | Mult LExp LExp
  | Eql LExp LExp
  | If LExp LExp LExp
  | Fn (String, LType) LExp
  | App LExp LExp
  | LetRec (String, LExp) LExp
data LIdent
  = LILiteral String
  | LINested LModule String
```

Figure 16: AST representation of LM.

C Failing Test Case

The test case that fails is falsely rejected is shown in Figure 17. Again, this occurs due to ambiguity detection.

```
module B {}
module A {
  module B {}
}
module C {
  module A {}
}
module I {
  import B
  import A
}
```

Figure 17: The test case that fails.

D Other Languages

Ruby and C++ have similar relative and glob import semantics as LM, though imports are unordered. This can be seen in Figure 18. Note that in Ruby, *x* could not be referred via *B::x*. This is however possible in C++.

(a) Ruby module example.

```
module A
  module B
    def x; 19; end
  end
end
include A
include B
def y; x; end
```

(b) C++ namespace example.

```
namespace A {
  namespace B {
    int x = 19;
  };
};
using namespace A;
using namespace B;
int y = x;
int main() {};
```

Figure 18: Relative and glob imports of arbitrarily nested compilation units in other languages.