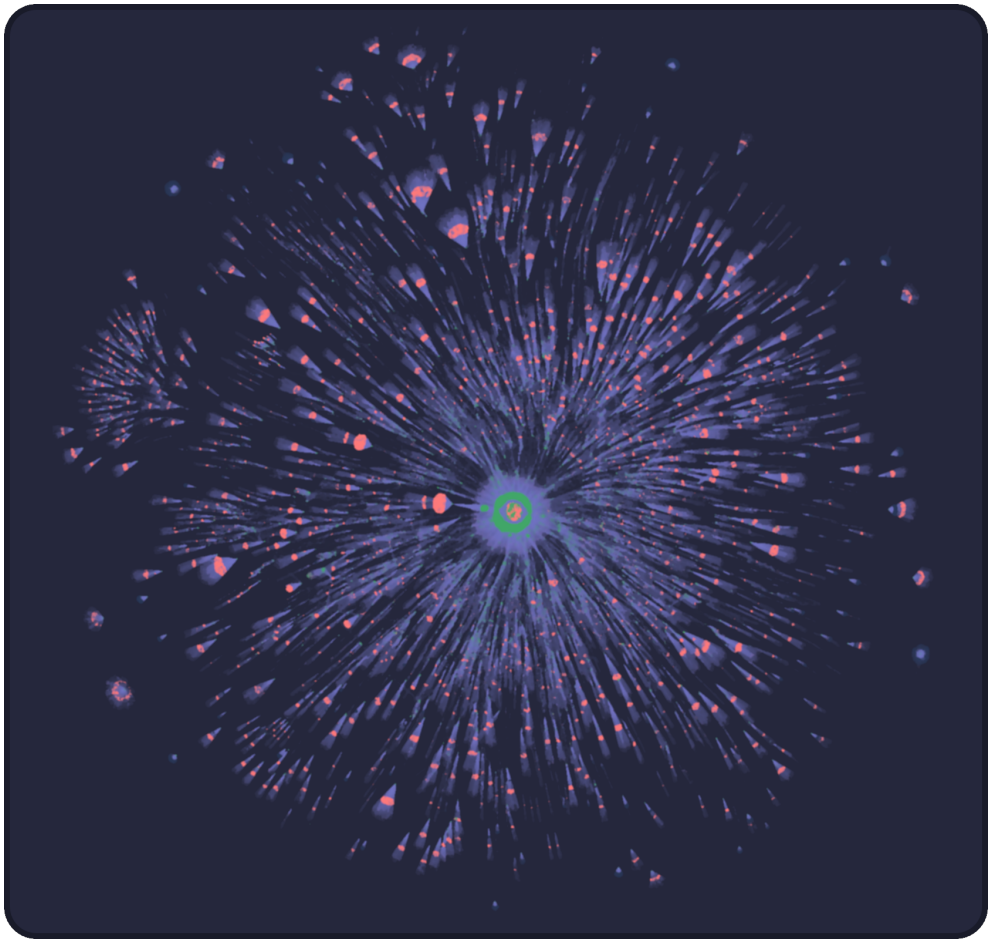
Memoising Scope Graph Query Resolution

October 28, 2025



Arthur de Groot

Memoising Scope Graph Query Resolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Arthur de Groot
born in Delft, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Scope graph for the Java 1.5standard library.

Memoising Scope Graph Query Resolution

Author: Arthur de Groot
Student id: 5068916

Abstract

The Statix meta-language is a domain-specific language that is used to describe specifications for type systems using high-level declarative inference rules. Type checkers can be automatically generated from these rules, saving one from the burden of writing it manually. One of the problems with these generated type checkers is performance; handwritten type checkers usually outperform the generated ones. Statix uses the formalism of scope graphs to represent name binding, and querying these scope graphs is known to be the main performance bottleneck. Improving the performance of queries means improving the performance of the type checkers generated by Statix.

In this thesis, we propose a memoised variant of the current state-of-the-art query resolution algorithm that memoises data encountered during graph traversal, reducing future queries to a cache lookup. We also identify common patterns in real-world scope graphs and link those to the name binding structure that created them. We construct a synthetic dataset with these patterns that is used to evaluate query resolution algorithms with microbenchmarks. This gives us more granular information on what name binding structures benefit most, if at all, from memoisation.

The results of these benchmarks are the performance differences between the memoised algorithm and the current state-of-the-art algorithm per identified pattern. They show that our proposed algorithm breaks even in terms of performance after only two queries for most patterns. Furthermore, we demonstrate that our proposed algorithm and the current state-of-the-art provide identical efficacy. The tradeoffs are twofold: the cache increases the memory usage of query resolution significantly and the query resolution parameters were tweaked to make caching possible, but less versatile. The changed query parameters' limitations should only be theoretical however, the Statix specification for Java 1.5 has 23 out of 25 fully compatible queries.

Thesis Committee
Dr. J. Cockx, EWI-ST-PL, TU Delft
Dr. A. Costea, EWI-ST-PL, TU Delft
Dr. K. Langedoen, EWI-ST-PL, TU Delft
A. Zwaan, EWI-ST-PL, TU Delft

Supervisors
Dr. J. Cockx
Dr. A. Costea
A. Zwaan
J.B. Dönszelmann

Preface

The road to getting this thesis done was very rocky to say the least and I want to thank all of those who have helped me throughout this journey. I would like to thank Jana Dönszelmann for getting me acquainted with the PL group and giving me the opportunity to start this thesis. Without her, this thesis would never have even happened. I also want to thank Aron Zwaan for his help and readiness to answer any and all questions I had. Even after he had left the TU Delft, he still made time for me, which I am very grateful for. I am also very thankful for Andreea Costea for her supervision of this project. She has put a lot of effort into helping me with the thesis, even though scope graphs were a novel concept to her. Finally, I want to thank Isabelle Beekmans for helping me find all the big and small spelling and grammar errors. Finally, I want to thank you, the reader, for taking the time to read through this thesis.

Arthur de Groot
Delft, the Netherlands
October 28, 2025

Contents

Chapter 1

Introduction

Implementing type checkers is difficult. On the one hand, they must properly represent the specification of a given type system, while on the other hand the implementation of type checkers is plagued with real-world concerns. Type checking is closely interwoven with name binding and requires careful traversal of the abstract syntax tree. Furthermore, while many type system specifications for different languages share commonalities, their implementations obscure this fact, making it difficult to adapt advances in one type system to another language.

To help with this difficult task, the Statix meta-language provides language implementors with a Domain-Specific Language (DSL) for defining type systems with high-level declarative inference rules [1]. The goal is to automatically generate a type checker from these rules, saving language designers from implementing one manually. This allows language designers to focus more on the semantics of their type system, rather than the implementation details of a type checker.

While Statix makes it easier to create type checkers, there is one major downside: its performance. Zwaan [2] reports that type checking the Apache Commons IO library is at best 3 times slower with Statix compared to *javac* (the Java compiler), even with their contributed optimisations. The Statix compiler has been bootstrapped, meaning it has been written as a Statix specification [3]. While the compiler was functionally correct, poor performance was observed here as well; in fact, the poor performance has hindered its adoption.

Statix uses scope graphs to generate its type checkers. Scope graphs were introduced by Neron et al. [4] to model the name binding structure of a program, suitable for both formalisation and implementation. Scope graphs have also been shown to be able to model type systems of differing complexity, from simply typed lambda calculus to Featherweight Generic Java [1].

Name resolution is done by traversing this graph, which is called querying, and resolving these queries is done using a query resolution algorithm. We denote the current state-of-the-art query resolution algorithm [5] with A_{cur}. This algorithm takes some parameters, such as which labels are allowed, to determine what scopes, and thus data, are reachable and visible. Queries are versatile and are the main mechanism for working with scope graphs. A query refers to a set of query parameters and a query instance refers to the execution of a query, starting in a certain scope. Queries result in an environment, which is a list of data, and the paths taken through the graph to reach that data.

According to Zwaan [2], the slowest part of scope graphs – and thus Statix – is the aforementioned query resolution. Multiple steps in the algorithm scale poorly with the graph size. They identified that traversal of the graph during query resolution was a significant bottleneck for performance and optimised this. However, even with these optimisations, poor performance was still observed for the bootstrapped Statix compiler.

To understand the causes of this inefficiency, we start from a simple observation: in most

programs, there are more references to some variable than there are declarations of that variable. After all, programmers declare variables with the intent to use them and, usually, unused variables are treated as warnings [6, 7, 8]. Finding a name in a scope graph involves performing one or more queries, meaning that name resolution in scope graphs scales with the number of references. This is not desirable, as this results in duplicate work, since individual query instances do not share any results. Oftentimes, multiple query instances are executed in scopes close to one another and all these instances would trace very similar paths through the scope graph.

This thesis proposes to improve the current query resolution algorithm in scope graphs by adding support for memoisation. This proposed memoised query resolution algorithm is referred to as A_{mem}, The core problem is that multiple query instances do not share any results, leading to duplicate work. Memoisation, or caching, could potentially alleviate these concerns, by allowing query instance to reuse environments from previously executed instances. Adding memoisation to the query resolution algorithm is not trivial and this thesis discusses the challenges surrounding and the implementation of A_{mem}. It is not enough to simply store a lookup table of previously executed queries, as the parameters for each query are different. For example, queries for name `"x"` have incompatible parameters with queries for name `"y"`, even if they traverse the same parts of the graph. Such a lookup table would thus have a low hit-rate.

The evaluation of A_{cur} and A_{mem} is done in a controlled setup, as a controlled environment gives much more insight into how and when the algorithms outperform each other. The environments are created based on commonly occurring scope graph patterns. Second, the current state-of-the-art dataset for scope graph evaluation includes only Java projects, which limits performance conclusions to that language. In contrast, a controlled setup is language-agnostics and provides insights into how query resolution algorithm performance is influenced by patterns occurring in scope graphs, rather than language-specific constructs.

While scope graphs abstract over name binding structures in languages, these features are still visible in the graphs. A name binding structure is usually modelled with the exact same small scope graph; creating a new variable or new structures will always create the same kind of scope graph. This means that there exists a rough mapping of language features to certain patterns in the scope graph. A dataset containing real-world scope graphs is used to create such a mapping; we extract patterns from the scope graphs and find out which name binding structure created them. From this, we create a synthetic dataset.

In this thesis, we create a synthetic dataset using these patterns, which is then used in the evaluation of both algorithms. This means that the synthetic scope graphs do not represent any specific language, but rather a set of commonly found name binding mechanisms, such as variable declaration, function declaration and struct/class declaration. This allows us to reason about the performance of A_{cur} and A_{mem} in relation to these language features.

In summary, this thesis seeks to answer the following questions:

1. $Q_{pattern}$: What kind of patterns occur in real-world scope graphs?
2. $Q_{efficacy}$: Does A_{mem} produce similar results to A_{cur}?
3. $Q_{performance}$: How does the performance of A_{mem} compare to A_{cur} for the found patterns?

1.1 Contributions

This thesis makes the following contributions:

- We describe and implement a memoised query resolution algorithm (A_{mem}). This new algorithm should solve the core problem of slow query resolution performance by reducing duplicate work.

- We create a synthetic dataset of common patterns in scope graphs, based on real-world data.
- We evaluate the state-of-the-art query resolution algorithm (A_{cur}) and our proposed memoised version (A_{mem}) using the synthetic dataset.

1.2 Outline

This thesis is structured as follows: Chapter 2 and Chapter 3 introduce relevant background information on scope graphs and query resolution. Next, Chapter 4 introduces A_{mem} and explains the reasoning behind it and Chapter 6 focuses on the evaluation and compares results of both A_{cur} and A_{mem}. Finally, in Chapter 8 concludes the thesis and discusses possible future directions.

Chapter 2

Scope Graphs

Modelling name binding structures generically in language is not an easy task and different approaches exist [4, 9, 10, 11]. In this thesis, the focus is mainly on scope graphs and this chapter discusses the necessary background information that is required to understand how they are useful for name resolution. A small example language is used and we will show how scope graphs are used by example.

2.1 Simply Typed Lambda Calculus

A Simply-Typed Lambda Calculus (STLC) program is used as a running example throughout this document and its specification is shown in Figure 2.1. This STLC is a simple language that contains name declaration and binding, (anonymous) function declaration and application, and simple arithmetic such as addition. The language only uses natural numbers.

```
n: natural numbers := {0, 1, 2, 3, ...}
x: identifier := string
t: types := num | bool | t->t
e: expr := n | b | x | e + e | fun(x: t) { e } | e(e) | let x = e; e
```

Figure 2.1: STLC Specification

Variables are declared using `let` expressions, which declare their name and type. For example, `let a = 3` declares a name `a` that implicitly has type `num`, since $3 \in \mathcal{N}$. Functions are declared using `fun(x: t) { e }`, however a `let` binding is still required to bind this declaration to a name. Functions can only have a single argument and a single return type, giving the previous expression the type `t -> t`. Addition can only be performed on two numbers and the return type of an addition is always another number.

With this language we can create an example program that declares a number and a function that multiplies a number by two. This would look as follows:

```
1 let x = 3;
2 let times_two = fun(x: num) { x + x };
3 times_two(x)
```

Figure 2.2: Example STLC program declaring a variable `x` and a function `times_two`

In the example in Figure 2.2 on line 1, a variable `x` is declared that is inferred to be of type `num`, since `3` is a num. On the next line, `times_two` is declared, which takes a `num` as an argument and returns `x + x := num`. Its type is thus `num -> num`. This program also shows

two different kinds of scopes being created. We define a scope as a region of a program where name resolution is uniform [1]. With this definition, a scope is created with a `let` binding and when creating a function. This also intuitively makes sense, as the name `times_two` is undefined on line 1 and inside the function body on line 2 `x` refers to the name declared inside `fun(...)`, rather than the `x` on line 1.

What was just explained are in fact scoping rules, and name resolution combined with type checking. This can be modelled using scope graphs, and the next section will show how that is done for Figure 2.2.

2.2 Modelling an STLC

```
1 let x1 = 3;
2 let times_two1 = fun(x2: num) { x3 + x4 };
3 times_two2(x5)
```

Figure 2.3: Example STLC program. The subscripts indicate the ith occurrence of a name.

We revisit an annotated version of the example from the previous section in Figure 2.3, where the subscripts indicate the ith occurrence of a name. Recall from the introduction that a scope is defined as a region of a program that is uniform with respect to name resolution. That means that every `let` binding and function declaration creates a new scope. In this program, three scopes are created because there are two `let` expressions and one function declaration. The first scope (s_1) is created on line 1, which declares $\texttt{x}_1$. On line 2, two distinct scopes are created, s_2 that declares $\texttt{times\_two}_1$, and s_3, which represents the function body, that declares $\texttt{x}_2$. Both s_2 and s_3 must be able to reach s_1, but not each other. This makes sense, as inside the function body, `times_two` should be undefined[1]. Line 3 does not create a new scope since no names are declared, instead it is in s_2.

As we have seen, this program creates some scopes with a relation to each other. Not all scopes are reachable from a given scope, for example when declaring a function, the scope in which the function is declared and the scope representing function body cannot reach each other. We can construct a graph representation of these scopes, where nodes represent the scopes and the (directed) edges the relation between scopes.

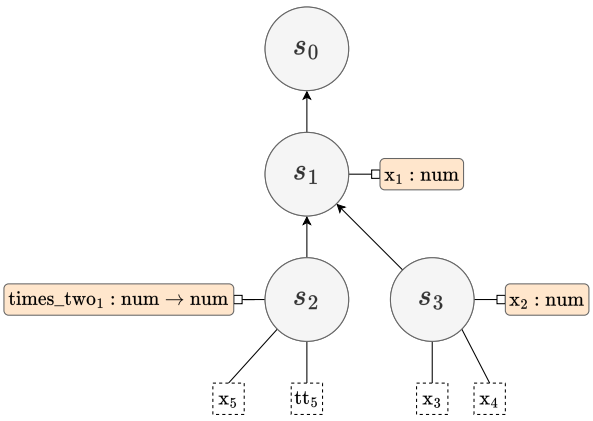


Figure 2.4: Graph model of the scopes seen in Figure 2.3.

Figure 2.4 shows a graph of the scopes of the example program. The nodes indicate scopes, with the number being the (unique) ID of that scope. The blocks indicate variable

[1] Functions are anonymous in the STLC

declarations, which are connected to a single scope using an edge with a blocky arrowhead. The squares indicate that references to a certain name are made in a scope. These are not actually part of the graph, but shown here for clarity. Note how these references do not hold type information, as they are only a name.

The graph shows the same scopes that were explained previously, with the exception of s_0. This scope represents the root of a program, and always exists. This can be thought of as a "global" scope for languages that have such a concept.

To reiterate, s_1 is created after the declaration of `x`$_1$ on line 1. s_2 is created because of the declaration of `times_two`$_1$ and s_3 represents the function body. `x`$_2$ is declared inside this scope, as that is where the name exists.

The edges between the scopes are new, and the edges with a pointy arrowhead represent that another scope is a lexical parent. The blocky arrowheads represent a declaration. The graph can only be traversed in the direction of the edges, meaning we successfully represent that s_2 cannot reach s_3 and vice versa.

Name resolution can be performed by traversing the graph. A name can be resolved by starting in a scope and following the edges of the graph until a declaration is found with a matching name. Edges with a declaration (a blocky arrowhead) are followed over edges with a normal arrowhead. Following these rules, `x`$_3$ and `x`$_4$ in s_3 both resolve to `x`$_2$`: num` and `x`$_5$ resolves to `x`$_1$`: num`. Notice that this is the exact same result as the name resolution that was previously done by hand in Section 2.1.

The graph we just constructed and used to perform name resolution is in fact a simplified scope graph. In the next section, scope graphs are formalised.

2.3 Scope Graphs

Scope graphs are a framework that can be used to model the name binding structure of a program [4]. Scope graphs abstract away the syntax of any specific language, and are applicable to many real languages [1]. Scope graphs are defined as follows:

Definition 1 (Scope graphs)**.** A scope graph is a graph $\mathcal{G}$ that contains scopes (s), labelled edges $(s_1 \xrightarrow{l} s_2)$ and data (d). A scope graph has labels $\mathcal{L}$ and data terms $\mathcal{D}$. Each scope represents a block in the modelled language in which name resolution is uniform. Scopes are connected by labelled edges $s_1 \xrightarrow{l} s_2$, $l \in \mathcal{L}$ where the label is used to indicate the relation between the two scopes. An edge from s_1 to s_2 indicates that the declarations in s_2 are reachable from s_1. d is a model for the name-type declarations made in a program. They are connected to scopes by a datum $s \xrightarrow{r} d$, which is a labelled edge with some relation r.

The distinction between labels (l) and relations (r)[2] is, for the purposes of this thesis, not important and instead we use a slightly different model. Instead of having a distinction between scopes and data, scopes can hold a datum instead. A declaration is thus modelled as a scope containing a datum, instead of using a separately defined datum. This model has been used before [12] and is used in implementations of scope graphs [13, 14]. It simplifies query resolution slightly, because the number of parameters is lower; we only have to consider one set of labels.

In Figure 2.5 a scope graph is shown of the example program from the previous section. The most notable difference with Figure 2.4 is that references are not part of the graph anymore. The coming section will explain name resolution, showing why this is not necessary. Furthermore, the edges are now labelled, which give the same meaning the pointy and blocky arrowheads did before. Blocky arrowheads are still used for declarations by convention, but they have no meaning. These edges represent declarations because of the D label,

[2]The distinction between labels and relations is made for typing the Statix surface language.

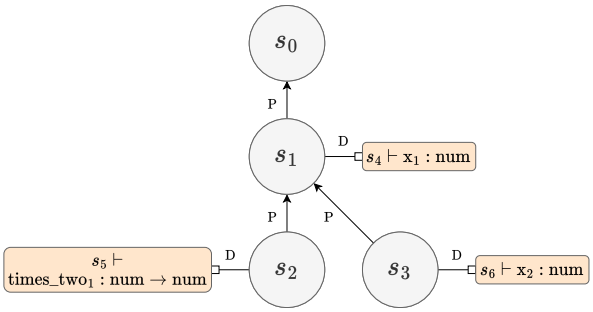


Figure 2.5: Scope graph for the program in Figure 2.3

not arrowheads. The blocks containing the declared data (yellow background) are scopes that contain data. Again, they are drawn as blocks to conform to convention.

Similarly to before, name resolution can be performed by traversing the graph. The next section will go into detail on how that is done.

2.4 Name resolution - Queries

To link a reference from a scope to data elsewhere in the graph, the graph must be traversed. This is referred to as querying and a query is essentially a parameterised graph traversal from some starting scope, to one or more data in the graph. The result is a list of the matched data, and a path:

Definition 2 (Path)**.** A path (p) is a sequence of scopes and edges through the scope graphs from some starting scope to another scope. A path from scope s_1 to scope s_2 is noted as $s_1 \rightarrow s_2$.

This list of matched data and a path is an environment, defined as:

Definition 3 (Environment)**.** An environment (E) is a set of pairs of (p, d), which is a tuple of a path and data. Each tuple represents a datum in the graph, and the path taken to find it.

This section will focus on a high level explanation of querying, while Chapter 3 discusses executing queries.

2.4.1 Reachability & Visibility

When querying the graph, we want control over what environments we find. In the name resolution example in Section 2.2, we wanted to find `x` and thus deliberately ignored `times_two`. Implicitly, the concepts of reachability and visibility were used. In scope graph queries, these two concepts are controlled with four query parameters that control both the reachability and visibility of paths and data. This section will explain those parameters and how they are usually represented.

Scope graphs contain labels that indicate the relationship between scopes. While constructing paths during a query, only the labels between scopes are considered, not the scopes themselves. The behaviour of query resolution only depends on the labels between scopes. This is emphasised, since the query parameters operate on either the labels or data in the graph.

Definition 4 (Reachability)**.** Reachability states that a certain scope, and by extension its datum, is reachable from the current scope. In other words, a well-formed path p $s_1 \rightarrow s_2$ exists to some well-formed datum d.

To control what is considered well-formed, two query parameters are used: label well-formedness (R_l) and data well-formedness (D_{WFD}). The R_l defines what labels a certain path is allowed to contain. This is defined as a regular expression (hence the R), with $\mathcal{L}$ as the alphabet. The labels of the traversed edges are used as the input string to this regular expression. During query resolution, the path is only partially known, which means the regular expression must be partially evaluated. Chapter 3 goes more in depth on how this is done. Since this is a regular expression that determines possible paths, it is also referred to as the path regex.

The D_{WFD} defines the data that a certain query can match. This is usually defined as a predicate that accepts some data term d and returns whether that datum matches what is being searched for. For example a D_{WFD} could look like D_{WFD}`: (d) => d.id == "x"`. Here D_{WFD} is defined as a function that takes in some datum `d`, and returns whether the id of `d` is equal to `"x"`. With this D_{WFD} we look for data that has the name `"x"` and reject any datum with a different name.

Together, the well-formedness parameters control what paths through the graph are possible and what data can be matched, which controls reachability of environments. In the next subsection, a R_l and D_{WFD} are formulated to perform name resolution on the scope graph in Figure 2.5.

In Figure 2.4, we found that x_4 and x_5 should resolve to x_2 and not x_1. Implicitly, we applied visibility to the graph and shadowed x_1 with x_2. When applying just the R_l and D_{WFD}, environments will be found for both x_1 and x_2. We have found the *reachable* environments, but have yet to filter for *visible* environments.

Definition 5 (Visibility)**.** Visibility states that a certain scope, and by extension also its datum, is both reachable and visible from the current scope.

Visibility entails the ordering of the reachable (sub)environments. To control this, two query parameters are used: the label ordering ($L_<$) and the data ordering or data equivalence ($D_\approx$).

The $L_<$ is a strict partial order between the labels that exist in a scope graph. Since paths consist of a sequence of labels, $L_<$ becomes a way to sort paths. Shadowing of paths can be controlled with this parameter, such as preferring declarations that are local to a scope. For example, in Figure 2.5, by using a label ordering $D < P$, a query starting in s_3 will prefer x_1 over x_1; this makes x_2 shadow x_1. Note that the "smaller" label is the label with a higher priority.

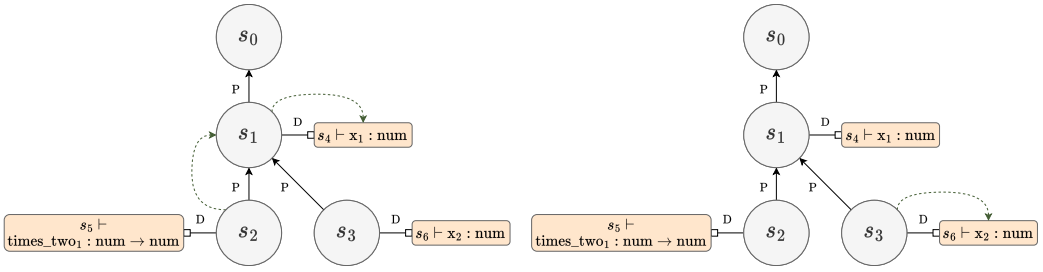
The $D_\approx$ is a partial ordering of the data in the scope graph. However, it is usually reasoned about as a data equivalence predicate instead [1, 5] and also implemented in this way [13, 14]. The data equivalence predicate takes in two data and returns whether they are equivalent. For example, $D_{\approx,name}$`: (d1, d2) => d1.id == d2.id` is a predicate that accepts two data d_1 and d_2 and returns whether they share the same name. The $D_\approx$ is used to determine what data in an environment should shadow each other. It can also be used to perform more advanced name resolution such as function overloading or subtyping. Of the four query parameters, the $D_\approx$ is used least. In the Statix specification for Java for example, either $D_{\approx,name}$ or a trivial function that always returns true is used [15]. It is rarely explicitly mentioned to be a predicate that returns whether two data share the same name [1]. In other work it is left out [5] or only briefly mentioned [2].

Summarised, there are four query parameters: the label well-formedness (R_l), data well-formedness (D_{WFD}), label ordering ($L_<$) and a data equivalence predicate ($D_\approx$). Together, these parameters control the reachable and visible scopes when querying.

Scope graphs can have more than one set of query parameters, as a single query is usually not enough to properly express the semantics of a given language. For example, a language with a class system like Java would use a separate query for resolving variable names and

class members [1]. The next section will discuss how the STLC can be queried using scope graphs.

2.4.2 Querying the STLC



(**a**) Query for `"x"` starting in s_2 (**b**) Query for `"x"` starting in s_3

Figure 2.6: Scope graphs for the example program in Figure 2.3 where queries have been performed. On the left a query for `"x"` is started in s_2, while on the right the same query is started in s_3

In Section 2.1, name resolution was performed by hand on a small STLC program. In Section 2.2, name resolution was performed on that same program using a graph of scopes. Now we will perform name resolution on that program using scope graphs.

Since the STLC is quite simple and only contains one built-in name binding structure, only a single set of query parameters is necessary. To properly set up these query parameters, consider the example in Figure 2.3 and its scope graph in Figure 2.5. The scope graph contains two labels: P and D labels, which represent a lexical parent and a declaration respectively. Furthermore, there is only one type of datum: a variable declaration, which has a name and a type.

First, we must set up the rules for reachability. In the STLC, names are always reachable, no matter how many `let` statements exist in between. Valid paths must end at a scope containing data. Paths can thus contain zero or more P label, while needing to end with a D label, which translates to the regular expression $R_l = P^*D$. Data is only considered well-formed if it matches a name that is currently being queried for. Hence, we choose D_{WFD} `= (d) => d.id == WFD_ID`, where `WFD_ID` is the name that is being queried for.

Second is visibility. We prefer more local declarations in the STLC. For example, x_4 should resolve to x_2 and not x_1. We can represent this using $L_< = \{D < P\}$. Two data can only shadow each other if they share the same name, meaning that for the $D_{\approx}$ we choose $D_{\approx,name}$.

In Figure 2.6a and Figure 2.6b the scope graph for the STLC example program where a query has been performed is done. The dashed green line show the found environments.

The query shown in Figure 2.6a starts in scope s_2 and `WFD_ID = "x"`. Following the R_l, only scopes with a declaration are reachable, which in this case are scopes s_5 and s_4. `times_two` does not match the D_{WFD}, so that datum is disregarded. x_1 does match the D_{WFD}, which means an environment was found. Since there is only one environment, no shadowing has to be performed and we are done. The resulting environment is $\{(p : s_2 \rightarrow s_4, d : x_1 \texttt{: num})\}$.

In Figure 2.6b, the query instead starts in scope s_3 with `WFD_ID = "x"`. We follow the same steps as before, and are left with two environments for both x_1 and x_2. Now, the $L_<$ can be used to perform shadowing. The labels in the paths to x_1 and x_2 are PD and D respectively. With $D < P$ as the label ordering, this means that $D < PD$ and thus the path to x_2 shadows the one to x_1. Finally, we confirm x_1 and x_2 are equivalent by using the $D_{\approx}$. Since the two data share the same name, they are indeed equivalent and we can say that x_2 shadows x_1. The result of this query is the environment $\{(p : s_3 \rightarrow s_6, d : x_2 \texttt{: num})\}$.

2.5 Summary

In this chapter we have seen how scope graphs are defined and how they can be used to perform name resolution. Name resolution in scope graphs is done via querying and four query parameters were introduced to control both reachability and visibility of data. Below is a summary of these four parameters.

- Label well-formedness or Path regex (R_l): Regular expression of labels that can appear in path.
- Data well-formedness (D_{WFD}): Predicate for matching data.
- Label ordering ($L_<$): Strict partial order of labels.
- Data equivalence: ($D_\approx$): Data equivalence predicate.

The next chapter will focus on algorithmically performing these queries and the current state-of-the-art query resolution algorithm.

Chapter 3

Query Resolution Algorithm

In Chapter 2, it was shown how to model an STLC using scope graphs and how name resolution is performed on scope graphs using queries. We have seen the four query parameters and an example query. This chapter will focus on how a query resolution algorithm functions and relevant implementation details. The currently used query resolution algorithm is explained, which is used as a baseline for the evaluation.

Any algorithm that can perform queries for scope graphs is called a query resolution algorithm (A_{query}). A query resolution algorithm is an algorithm that takes in a scope graph, a start scope, and the four query parameters and returns all matching environments. In this thesis we will consider two such algorithms: the current state-of-the-art (A_{cur}) and a memoised version of it (A_{mem}). In this chapter, we discuss A_{cur} and surrounding implementation details, while in Chapter 4 we explain how to add memoisation to query resolution and create A_{mem}.

3.1 Current Query Resolution Algorithm: A_{cur}

The first query resolution algorithm was formulated in Neron et al. [4]. The algorithm has evolved over the years to match new scope graph formalism [16, 1] and to improve performance [5]. While there exists recent literature on improving query resolution performance [17, 2], the core algorithm has stayed similar to what it was before. The current state-of-the-art query resolution algorithm [5] is designed using the actor paradigm [18] to perform query resolution concurrently between separate compilation units. For simplicity, a sequential version of the algorithm is discussed here that is functionally equivalent [1]. We call this simplified sequential version A_{cur}.

Before explaining the algorithm itself, we discuss some implementation details that are important to understand the algorithm.

3.2 Implementation details

While A_{cur} is essentially a form of graph traversal, there are still some details that require extra attention. These are how the path regex (R_l) is checked against the current path, and the behaviour when cycles in the graph occur.

3.2.1 Path Regex

As discussed in Section 2.4, one of the four query parameters is the label well-formedness or the path regex (R_l). This is a regular expression that is used to validate the paths during

[1] Only the syncing of state between the different actors are left out. The rest is unchanged.

query resolution; if a path matches R_l, then that path is valid. We refer to all the labels in a path as a string. A regular expression can only match full strings however. This is disastrous for performance, as that means paths must be evaluated that stop matching the regular expression after only a few labels. Assume that we have a set of labels A, B, C and a path regex A^*B, meaning that a well-formed path should contain zero or more A and exactly one B at the end. If we encounter an edge with label C during graph traversal, that path will never be well-formed, however a full string must be constructed before this can be evaluated.

There needs to be a way to determine whether partial strings can match the regex, so query resolution can be halted when an invalid label is encountered. In scope graph query resolution, this is done by taking the Brzozowski derivative [19, 20] of R_l.

Brzozowski Derivative

An example language is used to explain the derivative. $\mathcal{A}$ is an alphabet consisting of characters $a, b, ..., z$, R and P are regular expressions (regexes) that $R, P \in \mathcal{A}$. $\varnothing$ denotes the empty set, and ϵ denotes an accepting or nullable state. $v(R)$ is a function that determines whether R is in a nullable state.

$$
\begin{aligned}
\delta_a l &= \epsilon \text{ if } l = a \\
\delta_a l &= \varnothing \text{ if } l \neq a \\
\delta_a R* &= \delta_a R \cdot R* \\
\delta_a R \cdot P &= \delta_a R \cdot P + v(R) \cdot \delta_a P \\
\delta_a R + P &= \delta_a R + \delta_a P \\
\delta_b \delta_a R &= \delta_b(\delta_a R)
\end{aligned}
$$

Figure 3.1: Definition of Brzozowski derivatives (δ_a) on regular expressions. ϵ represents an accepting state, $\varnothing$ represents the empty set and P and R are arbitrary regexes.

$$
\begin{aligned}
v(\varnothing) &= \varnothing \\
v(\epsilon) &= \epsilon \\
v(a) &= \varnothing \\
v(R*) &= \epsilon \\
V(R \cdot P) &= v(R) \wedge v(P) \\
v(R + P) &= v(R) + V(P)
\end{aligned}
$$

Figure 3.2: Examples of nullability of certain regular expressions.

In Figure 3.1, the Brzozowski derivative for several regular expressions can be seen. Similar to a partial derivatives in calculus, the Brzozowski derivative is taken with respect to a single character. δ_a means taking the derivative with respect to a, similar to how $\frac{\partial}{\partial x}$ takes the partial derivative with respect to x. These derivatives make it possible to partially match a certain string of characters. Recall that a match was defined as an accepting state for a certain regex. A partial match can then be defined as follows: a string of characters $(a, a_1, ..a_n)$ partially matches a regex R if $v(\delta a_n ..\delta_{a_1} \delta_a R)$.

To do this algorithmically, a nondeterministic finite automaton (NFA) can be constructed using the Brzozowski derivative. In fact, the Brzozowski derivative was created in order to create finite state machines to represent regular expressions. The NFA that is created has states that represent (partial) regular expressions, and the state transitions represent the next character in the input string.

Figure 3.3 shows the NFA that is generated using the Brzozowski derivative for the regular expression P^*D. Each state represents a (partial) regular expression. Matching a string with an NFA is done as follows: starting in the initial state, the leftmost character is popped

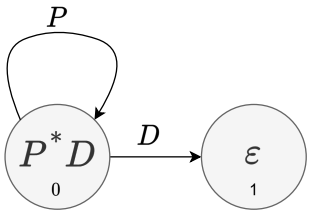


Figure 3.3: Example of a NFA for the regular expression P^*D. The next state transition is chosen based on the next character in the input string. If no state transitions are possible, the null state is entered. The number indicates the state index.

of the input string and the state transition in that state is taken. If the input string is empty, then it matches *iff* the current state's regular expression is nullable.

If no state transition exists, the string does not match the regular expression. For example, the input string PPD would reach states $0, 0, 1$ in that order, while the input string PPA would reach states $0, 0, \varnothing$.

Constructing such an NFA is done by taking the Brzozowski derivative for every character in an alphabet in every state, starting with the initial regular expression. This results in a new state with regular expression $D = \delta_a R$ and a state transition a. If two derivatives result in the same regular expression, only a state transition is added. This is done until every derivative has been computed in every state.

For the query resolution, what is most important to understand is that when taking the Brzozowski derivative with respect to some label, we are actually traversing this state machine. Alternatively, the Brzozowski derivative can also be seen as a way to check the path regex for partial strings. Furthermore, paths in query environments not only store the label of an edge, but also the state of the NFA after that edge has been traversed.

Circular paths

Scope graphs can contain cycles, but whether it does depends on what language the scope graph is constructed for. For example, circular imports can be represented with cycles in the scope graph. In general, paths that contain cycles are not allowed, because they are not useful; a path with a cycle is never the shortest path.

The query resolution algorithm will not explore an edge if that results in a cyclic path, in order to prevent infinite recursion. Paths store both the labels and regex automaton states, and a cycle is detected if two path segments share both.

For example, we have a scope graph with scopes s_1, s_2 and edges $s_1 \xrightarrow{A} s_2$ and $s_2 \xrightarrow{A} s_1$ and path regex AAB. If the traversal starts in s_1, then the algorithm will traverse to s_2 and back to s_1. In the second visit to s_1 however, the regex automaton is in a different state, namely a state that would only match B.

If the path regex was A^*B instead, then the automaton would be in the same state in both the first and second visit to s_1. In that case, traversal back to s_2 is rejected, since another path segment exists with both the same label and automaton state.

3.3 The Current Algorithm (A_{cur})

The goal of any A_{query} is to start in some scope and find matching environments in the scope graph. Recall from Section 2.4 that environments (E) are defined as one piece of data in the graph, together with the path to reach that data. A_{cur} uses four query parameters to determine what scopes and data are reachable. As mentioned previously, A_{cur} is a simplified, sequential version of the current state-of-the-art query resolution algorithm found in Van Antwerpen and Visser [5].

Algorithm 1: A_{cur}

```
1  function query(s, R, <_l, D, ≃_d):
2      return resolve(path(s), R, <_l, D, ≃_d);
3  function resolve(p, R, <_l, D, ≃_d):
4      return GetEnv(p, R, <_l, D, ≃_d)
5  function GetEnv(p, R, <_l, D, ≃_d):
6      L = {l | l ∈ labels(target(p)), ∂_l R ≠ ∅}
7      if ϵ ∈ R then
8          L = L ∪ $
9      return GetEnvForLabels(L, p, R, <_l, D, ≃_d);
10 function GetEnvForLabels(L, p, R, <_l, D, ≃_d):
11     E = ∅
12     L_max = ← get maximum labels from L using <_l
13     for l_max ∈ L_max do
14         L_low = ← get labels less than l_max
15         E = E ∪ GetShadowedEnv(l_max, L_low, p, R, <_l, D, ≃_d)
16     return E;
17 function GetShadowedEnv(l, L, p, R, <_l, D, ≃_d):
18     E_low = GetEnvForLabels(L, p, R, <_l, D, ≃_d);
19     E_max = GetEnvForLabel(l, p, R, <_l, D, ≃_d);
20     return Shadow(E_low, E_max, ≃_d);
21 function GetEnvForLabel(l, p, R, <_l, D, ≃_d):
22     if l = $ then
23         return {(p, d) | d = data(target(p)), D(d)}
24     else
25         P' ← step path towards edges with label l;
26         return {resolve(p', R, <_l, D, ≃_d) | p' ∈ P'}
27 function Shadow(E_1, E_2, ≃_d):
28     return E_1 ∪ {(p_2, d_2) | (p_2, d_2) ∈ E_2, ∄ (p_1, d_1) ∈ E_1, d1 ≃_d d2}
```

Algorithm 1 shows pseudocode for A_{cur}, with `query` as the entry point. The function takes in a starting scope (s), and four query parameters: the label regex (R), the label order ($<_p$), the data well-formedness (D) and the data order ($\simeq_d$). R also keeps track of the current state in the NFA (see Section 3.2.1). The algorithm contains some helper functions that are not defined in Algorithm 1 itself. These are `path(s)`, `target(p)`, `data(s)` and `labels(s)`, where p is a path and s is a single scope. `path(s)` constructs a path with a single segment from the given scope s. `target(p)` returns the scope at the tail end of p, which can be viewed as the "current" scope. To give some intuition, `target(path(s)) = s`. `data(s)` returns the data stored in s and `labels(s)` returns all unique labels of all outgoing edges of s.

The entry point `query` calls `resolve` (line 2), which then proceeds with immediately stepping into the next function `GetEnv` (line 3). This level of indirection seems unnecessary, but we keep it for a clearer comparison with A_{mem} in the next chapter.

A_{cur} functions locally on a single scope and traverses the graph by recursively calling `resolve` (line 26). We look at a single scope at a time, which is the scope at the tail end of p, accessed by `target(p)`. `GetEnv` (line 9) returns the environments for a certain path p. It does so by checking which labels can be traversed (L) and then calling `GetEnvForLabels` on line 10. On line 6, the Brzozowski derivative with respect to label l it taken on R, where l iterates over all labels on outgoing edges from the current scope. Labels that do not match the regular expression and thus the paths are not considered. On lines 7-8, the case is considered where R is in an accepting state and a special label \$ is considered as well. This special label represents the end of a path and is usually given the highest priority in the label ordering.

The functions `GetEnvForLabels` and `GetShadowedEnv` together perform shadowing for L. This is done by separating L into two sets of labels, L_{max} and L_{low}. To follow pre-existing

literature [1, 5, 17, 2, 21], we consider the most important label to have a minimum value, which follows the notation $D < P$. This means that D has a higher priority than P, because D is a lower value than P. L_{max} is the set of *lowest* priority labels and is obtained on line 12 by applying the $<_p$ on L. For each label l_{max} in L_{max}, we compute L_{low}, which is the set of labels with a *higher* priority than l_{max}. `GetShadowedEnv` recursively calls `GetEnvForLabels` (line 18), which effectively results into L_{low} being split into a new L_{max} and L_{low}, until L_{max} is empty.

Table 3.4: Calls to `GetShadowedEnv` if `GetEnvForLabels` is called with $L = \{A, B, C, D\}$ and $<_p = \{A < B, B < C, B < D\}$. The first column indicates the *ith* call.

`GetShadowedEnv`$_i$	l	L
1st	C	$\{A, B\}$
2nd	B	$\{A\}$
3rd	A	$\{\}$
4th	D	$\{A, B\}$
5th	B	$\{A\}$
6th	A	$\{\}$

To illustrate what exactly happens, we use the example in Table 3.4. In this table, `GetEnvForLabels` is called with $L = \{A, B, C, D\}$ and $<_p = \{A < B, B < C, B < D\}$. The table shows the values for arguments l and L in `GetShadowedEnv` for each (recursive) call to that function. Inside `GetEnvForLabels`, L_{max} will initially contain $\{C, D\}$, since those two labels do not have priority. In this first iteration of the loop on line 13, $l_{max} = C$ and $L_{low} = \{A, B\}$, and `GetShadowedEnv` is called on line 15. This is the call represented by the first row in Table 3.4. We repeat this process until we call `GetShadowedEnv` on line 15 with $l = B, L = \{A\}$. On line 18, `GetEnvForLabels` is called for the third time, inside which `GetShadowedEnv` is called again with $l = A, L = \varnothing$ (row 3 of the table). The next call to `GetEnvForLabels` on line 18 will result in an empty set, since L_{max} is empty (line 12) and the iteration (line 13) is never started. Inside `GetShadowedEnv`, E_{low} and E_{max} are created on line 18-19, which are the environments for L and l respectively. The call to `shadow` on line 20 shadows E_{max} with E_{low} environments. Taking the recursion into account, what is actually being called is `shadow(`A`, shadow(`B`, shadow(`C`)))`; we recursively shadow the subenvironments for the lower priority labels.

The table also shows an inefficiency in the algorithm. In the first call to `GetEnvForLabels`, $L_{max} = \{C, D\}$ (line 12) and up until now, only the iteration where $l_{max} = C$ has been considered. The second iteration, where $l_{max} = D$ results in the same recursive calls, which means that the environments for A and B are recomputed. This happens when $L_<$ is not a total order, meaning at least two labels do not have a defined ordering. Such orderings can occur when querying in a language with multiple types of inheritance. For example, Java 1.5 has class extension (E label) and interface implementation (I label) to inherit either methods from a class or an interface. In the Statix specification for Java 1.5, queries exist where the ordering between labels E and I is unspecified [15]. Current implementations usually cache these subenvironments [13, 14], however, formally this is not part of the algorithm as described in Van Antwerpen and Visser [5].

Continuing on line 19, `GetEnvForLabel` finds the environment for a given path for a single label. Recall from `GetEnv` that a special end-of-path label \$ exists, for which we return the data from the current scope (lines 22-23). Otherwise, we recursively call `resolve` for every adjacent scope that is connected by an edge with label l (lines 25-26). This can be viewed as traversing the graph, as for every scope we repeat the entire process. The algorithm continues until every reachable scope has been traversed (line 25).

3.4 Conclusion

In this chapter we have seen what a query resolution algorithm (A_{query}) entails. We have seen some implementation details surrounding query resolution and how the current (simplified) state-of-the-art query resolution A_{cur} works. We have also encountered an inefficiency, namely that subenvironments are possibly recomputed when the label ordering is not a total order in `GetEnvForLabels`. In the next chapter, we will discuss more performance limitations of A_{cur} and explore how memoisation can be added to a query resolution algorithm, along with its implementation.

Chapter 4

Bottom-up Query Resolution

In the previous two chapters, Chapter 2 and Chapter 3, we have discussed how scope graphs can be used for name resolution, and how this is algorithmically done with A_{cur}. This chapter will discuss some limitations of A_{cur}, and a way to ameliorate those limitations using memoisation.

4.1 Shortcomings

In Chapter 1, we discussed that query resolution performance is poor compared to hand-written type checkers. This section will discuss this in more detail, and explore what exactly makes it slow.

A few works have focused on improving performance for Statix [5, 17, 2]. While all three works have resulted in significant performance improvements for query resolution in Statix, only one has focused on improving the performance specifically of A_{cur}. Zwaan [17] reports that Statix is slow due to the query resolution algorithm and identified two main bottlenecks in A_{cur}: computing L_{min} and L_{max}, and computing the Brzozowski derivatives. Computing L_{max} and L_{min} in `GetEnvForLabels` scales quadratically with the number of labels, and the number of computed Brzozowski derivatives in `GetEnv` scales linearly with the number of labels. This introduces significant overhead, and Zwaan [17] created an intermediate language and a specialiser to optimise queries at compile time. We saw in Section 3.3 that A_{cur} contained an inefficiency when computing subenvironments with a partial label ordering. One optimisation that becomes possible with the intermediate language is eliminating the recomputation of these common subenvironments. Even with these optimisations, the performance of type checkers generated by Statix – and thus query resolution – is behind traditional compilers. A full compilation of the Apache Commons IO library [22] takes 3 seconds with `javac`, while only type checking the same library with the specialiser's optimisations applied takes an estimated 7 to 9 seconds [17]. This estimate was computed by multiplying the type checking time of the Statix solver before optimisations were applied, with the measured speedup.

The optimisations only remove duplicate computations of common subenvironments within a single query instance. Between multiple query instances, no such optimisations apply and duplicate work becomes a real possibility.

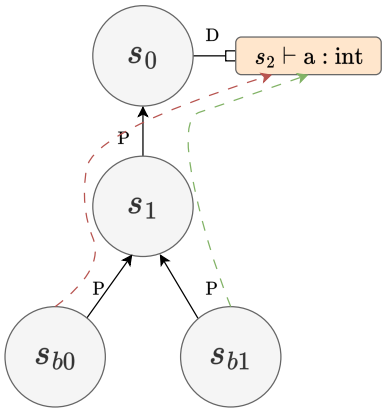


Figure 4.1: Example scope graph with a tree-like structure. Two queries are performed, starting in s_{b0} and s_{b1} respectively. The red and green arrows show the resolution paths.

In the example in Figure 4.1, a scope graph with a tree like structure is shown. Two queries are performed, one starting in s_{b0}, and one starting in s_{b1}. Both query instances use the exact same query parameters and are both matching data with name `"x"`. Observe that both queries share the same resolution path. In fact, apart from their starting scope, their traversal through the graph is identical. These two query instances performed mostly duplicate work. Even worse, performing duplicate work in this manner does not require two separate query instances.

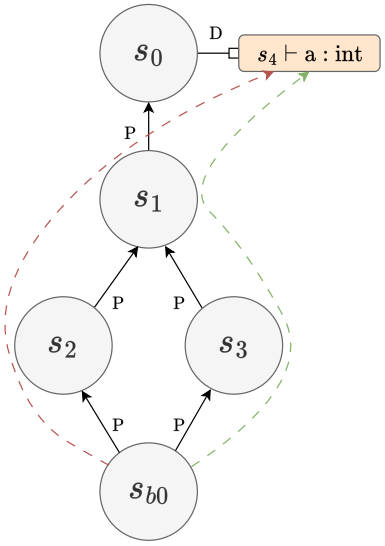


Figure 4.2: Example scope graph with a diamond-like structure. One query is performed, starting in scope s_{b0}. The red and green arrows show the resolution paths through the left and right scope of the diamond respectively.

In Figure 4.2 a scope graph with a diamond structure is shown. One query is performed starting in s_{b0} that looks for a name `"x"`. The resolution path splits for each scope in the diamond structure, after which they join together in s_1. Each path through this diamond structure is a separate resolution, meaning that any work performed after s_1 is duplicated for every scope in the width of such a diamond structure. These kinds of structures not only exist in real scope graphs, but can be common. This is elaborated upon in Chapter 5.

The core problem is that query instances do not share any performed work, resulting in potentially performing duplicate work. Different query instances can recompute the same environments, and for some patterns even the same query instance has redundant computation. We believe that introducing memoisation can alleviate these concerns. This is not

trivial however, and the rest of this chapter will focus on how caching can be added to scope graph query resolution algorithms.

4.2 The Cache

Caching intermediate results obtained during scope graph query resolution can potentially improve the performance of query resolution algorithms. However, we need to determine what to cache and how we can access cache entries. In Section 3.3, we saw that A_{cur} operates per scope. For each scope, the valid edges are traversed, or if the special \$ label is encountered, an environment containing the scope's data is returned.

The memoised version of A_{cur} that we propose, A_{mem}, should similarly store a cache on a per-scope basis. Crucially, this cache should store *all* data encountered during query resolution, not just well-formed data such as in A_{cur}. This ensures two things: cache entries are only created on demand and every resolution path through a scope is only considered once. Any subsequent query instance that visits an already visited scope can then read the cache.

The main concern with A_{mem} is to ensure validity of cache reads. Reading a cache entry should give no different results than performing A_{cur} by graph traversal. To ensure this, we must consider how we identify each cache entry, which we now refer to as the *key* of an entry.

The cache in each scope stores the environment that would be obtained by querying using A_{cur}. This means that keys in this cache must encode the parameters that A_{cur} uses. Using the entire path as a key is too limiting. The part of the path that matters is the current scope and the current state of the regex automaton (see Section 3.2.1). When using a path regex like P^*D, the number of P edges traversed does not matter for the validity of the key, only if we currently are still in the automaton state that represents P^*D.

Unfortunately, this is not as straightforward as it seems. In the next section we discuss the internal representation of the query parameters and how that affects our ability to use them as a cache key.

4.3 Attempt to use Query parameters as a Key

Keys are required to index our cache and these keys have two important properties:

1. Keys should be comparable for equality.
2. Using the same keys should guarantee the same query output.
3. The same data should result in the same key.

The first of these properties is obvious, if keys are not comparable, then the cache becomes unreadable. The second and third properties mean that we must choose a key such that reading from the cache gives the same environments as performing a query via graph traversal.

In the previous chapter, we found that part of the key includes the current scope and the current state of the regex automaton. Both of these parts satisfy both requirements. The missing portions of the key include the four query parameters, and in this section we will see whether the query parameters fulfill both conditions. To satisfy the second, we must use all query parameters in the key. This section will attempt to use a key that is a tuple of the aforementioned current scope and current state of the regex automaton, and the four query parameters.

The query parameters are either known statically and do not change per query instance, or are dynamic and are different in each query instance. The label ordering ($L_<$) and label well-formedness (R_l) are both known statically and do not change per query instance. Each

instance uses the exact same label ordering and well-formedness. For a key, this means we can search for a match in the cache with the currently used R_l and $L_<$. Additionally, there exist limited sets of all R_l and $L_<$ used in all queries combined. The cache can thus only contain the R_l and $L_<$ contained within these sets. This makes both the R_l and $L_<$ useful as a key, since they can consistently be found in the cache. Both these parameters are comparable for equality and would guarantee the same query output, meaning the first and second properties are fulfilled. Since these query parameters are unrelated to data, the third property is fulfilled by absence.

The data well-formedness (D_{WFD}) and data equality predicate ($D_\approx$) are dynamic. While the function they perform is known statically (i.e. match the name of a datum), the inputs change between query instance, which makes them more difficult to reason about. For example, a D_{WFD} that matches a name `"x"` is considered different from one that matches a name `"y"`, while they perform the same function: matching a name. This is a severe limitation if used as a key, as that means cache entries are only valid if the D_{WFD} is the exact same. In other words, the third property is unfulfilled. The same data would be represented differently when using the D_{WFD} as a key. The result is that only query instances that look for the same name will be able to share work. If another query instance looks for a different name, it will have to do the graph traversal over again, defeating the point of a cache.

Furthermore, even if it was somehow possible to identify a D_{WFD} by their functionality (i.e. this D_{WFD} matches names), there is another limitation. The D_{WFD} can only distinguish between data that matches its predicate and data that does not. This limits the functionality of the cache. In A_{cur}, shadowing is performed for data that matches the D_{WFD} predicate. Since A_{mem} should store all data encountered, this means only environments containing well-formed data can be shadowed, while non-well-formed data cannot. The result is a cache that must store much more data than should be necessary. Again, this is because the D_{WFD} cannot distinguish between different non-well-formed data.

4.3.1 Data Projection as a Query parameter

In the previous section we have seen that the R_l and $L_<$ are both easily usable as a key, however the D_{WFD} and $D_\approx$ are not. A_{mem} should store all data encountered during graph traversal, meaning the scope of the algorithm changes compared to A_{cur}. The distinction between well-formed and non-well-formed data that A_{cur} can make is suboptimal for A_{mem}. The memoised variant must be able to make distinctions between all data. This allows for an important optimisation, namely that shadowed environments can exist in the cache for each encountered datum individually.

To achieve this, we propose a replacement query parameter for both the D_{WFD} and $D_\approx$ that is both statically known, and can distinguish between multiple data. All R_l and $L_<$ used by every query are known at compile time and this new query parameter should be similar. We propose a data projection function (P_d), which returns a projection of a datum d. For example, `name(`d`)` is a projection function that returns the name of the given datum d. There must exist a set of all projection functions $\mathcal{P}$ that is known at compile time. We can then use P_d as a key in the cache. The P_d always gives the same projection for some datum d, meaning this satisfies the third property of our desired cache key.

Previously, shadowing was performed by applying $D_\approx$ on data that is known to be well-formed. One property of P_d is that equivalent data must give identical projections. To shadow data, we can simply compare the projections resulting from P_d. Furthermore, finding well-formed data is now done by providing a well-formed projection ($d_{P,WFD}$), which is a result of P_d that is considered well-formed. Using the `name(`d`)` example from above, `"x"` should be provided as the well-formed projection to find data with name `"x"`.

In summary, instead of the D_{WFD} and $D_\approx$, we have P_d, which is a function that projects data to some other form. In order to find well-formed data, a $d_{P,WFD}$ must be provided that

contains the projection for data that is considered well-formed.

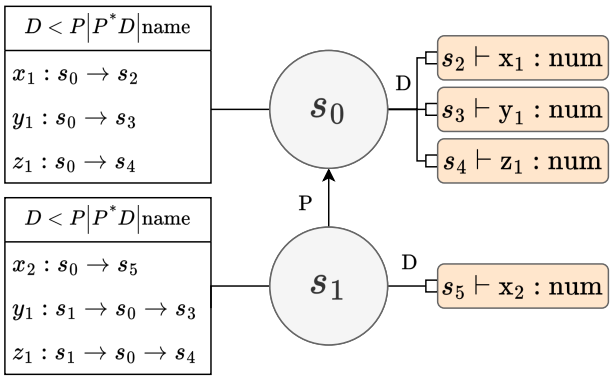


Figure 4.3: Example of a scope graph queried by A_{mem} with a filled in cache. The subscripts for the names x, y, z represent the ith occurrence. The boxes represent the cache entries, with each header being the $L_<$, R_l and P_d respectively. Each entry contains the name of the data and the matching environment. The regex automaton state for every cache entry is left out for brevity.

Figure 4.3 shows an example scope graph that has been queried using A_{mem} and has a populated cache. The rectangles on the left represent the cache, with the top representing the key, and the bottom part contains the stored environments, shown as the data name and the path to the data. A notable difference with A_{cur} is that A_{mem} stores environments for all data, not just well-formed data. Furthermore, in the cache belonging to s_1, we can see that x_1 was shadowed by x_2, as the entry for x_1 is replaced by a new entry for x_2. In fact, the cache entries are the environments one would find with an equivalent A_{cur} query starting in that scope.

Changing the query parameters to better suit a cache naturally comes with some limitations, which are discussed in Section 4.4.

4.4 Memoised Query Resolution Algorithm: A_{mem}

In the previous sections we discussed how the cache works and how it is accessed. This section focuses on integrating that cache within A_{cur} to create A_{mem}, and will explain how the algorithm works, and all changes compared to A_{cur}.

Algorithm 2 shows A_{mem}, with the colored text being differences with A_{cur} (see Algorithm 1). The entry point is still `query`, however the query parameters have changed. The D_{WFD} and $D_{\approx}$ have been replaced by P_d and $d_{P,WFD}$, which are the data projection function and data projection well-formedness respectively. `query` now no longer just calls `resolve`. In it, the $d_{P,WFD}$ is applied by only returning the environment where the datum's projection matches it (line 5).

On line 3, an instance of the cache is obtained. This cache is globally shared between query instances and the cache itself is defined on line 1. It is a set of key-value pair mappings. The key is a tuple of a scope s, the index of a regex automaton R_i, the path regex R, the label ordering $<_l$ and the used data projection function P_d. The value is an environment E. A cache entry is accessed by indexing the cache, denoted using $\mathcal{C}[K]$, where $\mathcal{C}$ is an instance of the cache and K is a key. At each step in the query resolution algorithm, the current environment is stored in the cache. Essentially, a read from the cache should give no different results than performing a query. For implementations it is recommended to use a hierarchical mapping instead, since three of the five elements in the key stay constant during the query.

Compared to A_{cur}, `resolve` (line 6) is also expanded. In this function, we now first check whether an existing cache entry already exists and return it (lines 7-9). If that is not the case, we traverse the graph by calling `GetEnv` on line 10. After obtaining the environment for

Algorithm 2: A_{mem}. The colored text indicate changes compared to Algorithm 1

```
1  Cache : {(s, R_i, R, <_l, P_d) → E} → cache definition
2  function query(p, R, <_l, P_d, d_{P,WFD}):
3      C = GetCache() → get an instance of the cache
4      E = resolve(p, R, <_l, P_d, C);
5      return {(p, d) | (p, d) ∈ E ∧ P_d(d) = d_{P,WFD}}
6  function resolve(p, R, <_l, P_d, C):
7      E_cache = GetCachedEnv(p, R, <_l, P_d, C)
8      if E_cache ≠ ∅ then
9          return E_cache
10     E = GetEnv(p, R, <_l, P_d, C);
11     CacheEnv(E, p, R, <_l, P_d, C)
12     return E
13 function GetCachedEnv(p, R, <_l, P_d, C):
14     K = (target(p), index(R), R, <_l, P_d)
15     return C[K]
16 function CacheEnv(E, p, R, <_l, P_d, C):
17     K = (target(p), index(R), R, <_l, P_d)
18     C[K] = E
19 function GetEnv(p, R, <_l, D, C):
20     ...
21 function GetEnvForLabels(L, p, R, <_l, P_d, C):
22     ...
23 function GetShadowedEnv(l, L, p, R, <_l, P_d, C):
24     ...
25 function GetEnvForLabel(l, p, R, <_l, P_d, C):
26     if l = $ then
27         return {(p, d) | d = data(target(p))} → D_WFD is not used anymore
28     else
29         P' ← step path towards edges with label l;
30         return {resolve(p', R, <_l, P_d, C) | p' ∈ P'}
31 function Shadow(E_1, E_2, P_d):
32     return E_1 ∪ {(p_2, d_2) | (p_2, d_2) ∈ E_2, ∄ (p_1, d_1) ∈ E_1, P_d(d1) = P_d(d2)}
```

the current scope, we first store it in the cache before returning it (line 11). For every scope encountered during graph traversal, a cache entry is made, which ensures that for a given set of query parameters, scopes are only traversed once. Any subsequent call will be a cache read. The function bodies for `GetEnv`, `GetEnvForLabels` and `GetShadowedEnv` (lines 19-24) are all unchanged compared to A_{cur}. The only difference is the arguments they take, which are only used to perform recursive calls to `resolve`.

Reading and writing to and from the cache is handled by `GetCachedEnv` (line 13) and `CacheEnv` (line 16) respectively. In both functions, the key is computed using the query parameters and the current path (line 14, 17). In `GetCachedEnv`, the cache is read on line 15, which returns an environment E. In `CacheEnv`, an environment E is stored in the cache (line 18). An astute reader might wonder why the $d_{P,WFD}$ is not checked when reading the cache. The reason is that the cache must be propagated through the graph. By matching all data in the environment here, we ensure all data is propagated.

The final two differences compared to A_{cur} are on lines 27 and 32 respectively. First, on line 27, we no longer check the D_{WFD} since all reachable data in the graph should be stored in the cache. Furthermore, the D_{WFD} is no longer used as a query parameter. Second, on line 32, shadowing is now performed by comparing whether the projection of two data is equal ($P_d(d_1) = P_d(d_2)$), rather than using the $D_{\approx}$. This is very important, as the implicit assumption made with using P_d is that data is considered equal if their projections are equal.

This is also how the data is stored in the cache.

One implementation detail regarding paths has been left out for brevity. Consider the following scenario: we traverse the graph from some starting scope s and reach scope s_c that has a valid cache entry. When we read that cache entry, all paths in the environment will have the form $s_c \rightarrow s_d$, where s_d is some scope that holds a datum. For a query starting in scope s, the paths should be of form $s \rightarrow s_d$, which is the path from s_c to s_d, prefixed by the path from s to s_d. Implementations must thus ensure the path is properly prefixed.

Limitations

Using data projection functions is more limiting than the D_{WFD} and $D_{\approx}$ that A_{mem} uses. There can now only be a limited set of projection functions ($\mathcal{P}$) instead of being able to use arbitrary functions that the D_{WFD} and $D_{\approx}$ allow. Some more advanced usages of the $D_{\approx}$, such as subtyping, become impossible with data projection methods.

However, we argue that P_d can cover the common case. In the Statix specification for Java, 25 unique queries are declared [15]. Of those 25 queries, 23 queries use a D_{WFD} that either always returns `true`, or matches the names of data. For all those 23 queries, the $D_{\approx}$ always returns true. In other words, those D_{WFD} and $D_{\approx}$ are easily expressible using a P_d. The remaining two queries are more advanced queries that are used to resolve subtyping by performing more queries (subqueries). While those queries for subtyping themselves cannot use a P_d, the subqueries can be, meaning the cache is still usable here.

This limitation is acceptable, as it is uncommon to have D_{WFD} and $D_{\approx}$ that are not representable by a data projection function. Furthermore, as shown in the previous section, A_{mem} is designed to be very similar to A_{cur}. It would be possible to fall back to A_{cur} if using A_{mem} is not possible. This means that the queries that represent subtyping can be performed using A_{cur}, while every other compatible query is executed using A_{mem}.

Circular graphs

As discussed previously, scope graphs can be circular and some special care must be taken to ensure query resolution is correct. In A_{cur}, circular paths are rejected, as this prevents query resolution from recursing infinitely. This is enough, as all query instances are completely separate from each other. For A_{mem}, this is not the case, as query resolution results are memoised in between instances. Normally, this causes no complications, except for circular graphs.

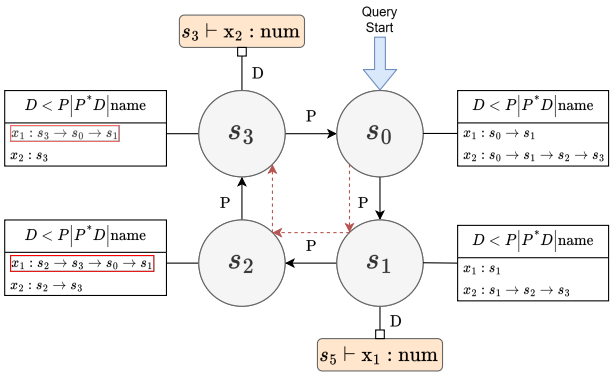


Figure 4.4: Example circular scope graph with cache that is incorrect after querying. The boxes next to each scope represent the cache entry for that scope. The red boxes indicate that that cache entry is missing. The dashed, red arrows show the resolution path. s_0 is not reached from s_3 since that would result in a cycle.

In Figure 4.4, a circular scope graph is shown. A query is started in scope 1, looking for a name `"x"` and all found environments are subsequently cached in each traversed scope.

The problem is that the algorithm does not reach s_0 from s_3, as that would create a cycle. Subsequently, $\mathtt{x}_1$ does not appear in the cache for s_2 and s_3. If a second query would be started in s_2, then it would read a cache entry and interpret that as the found environment, thus $\mathtt{x}_2$ would seem unreachable from s_2. This is obviously wrong, as a path $s_2 \rightarrow s_4$ exists.

The fundamental problem is that A_{mem} assumes that the environment is found when querying once. This assumption breaks in cycles, as we can see in the example above. This is a difficult problem to solve properly and will not be solved in this thesis. As we will see in Chapter 5, cycles are relatively uncommon. For now, the chosen solution is to not access the cache if the current scope exists in a cycle. This induces some extra overhead and also forces the graph to be re-traversed. However, this solution preserves correctness. This check only has to be enabled when it is known that cycles exist in a graph, and this will also be the case during the evaluation.

We will discuss a potential solution to store the cache in cycles. The cache does not store the path taken to create a certain cache entry, only the current scope. The cache should store the path that was used to create that entry (p_{cache}). If a cache entry is read in an instance where the current resolution path p overlaps partially with p_{cache}, then that cache entry is invalidated. A partial overlap in this case is defined as the set intersection of the scopes in p_{cache} and p. If this set is not the empty set, then there is partial overlap. Note that this should only apply in scopes that appear in cycles, otherwise this would erroneously find overlaps where there are none.

Another edge case that exists in circular graphs is related to cache propagation. In the example above during the second query instance, the cache in s_0 is valid as there is no partial overlap in the paths. However, propagation of the cache entry for $\mathtt{x}_1$ to s_3 is invalid, as this results in a circular path in the environment. The resolution path traverses from $\mathtt{x}_1$ around the circle to s_0, passing s_3 in the process. This means that propagated environments must also be rejected when they are circular.

4.5 Conclusion

In this chapter we have seen the implementation of a cache for scope graph query resolution and A_{mem}. To implement the cache, the parameters D_{WFD} and $D_{\approx}$ were swapped in favour of P_d and $d_{P,WFD}$. In the coming chapters, we will focus on evaluating A_{cur} and A_{mem}.

Chapter 5

Patterns

This chapter focuses on the preparatory analysis required for the evaluation of A_{mem}. Scope graphs from benchmarks used in previous works are analysed in order to find common patterns, which can be used in the evaluation of A_{mem}.

This section will answer the following question:

- $Q_{pattern}$: *What kind of patterns occur in real-world scope graphs?*

5.1 Goals

While scope graphs abstract away the syntax of a language, the name binding structure of a language remains. Depending on what name binding mechanics a language has, different patterns can occur in the graph. In general, scope graphs for C programs will be structured differently than those for Java programs, however similar patterns can still be observed. For example, variable declaration in C and Java create a similar scope graph, but classes as they exist in Java do not exist in C and hence this part of the graph would be different.

This thesis aims to provide language designers with more options regarding query resolution in scope graphs. Therefore, the evaluation will not simply run A_{cur} and A_{mem}, as this can only provide information on query resolution for Java 1.5.

Performance evaluations on scope graphs for different languages give different results, and cannot explain why one A_{query} outperforms another. A controlled setup would allow us to give much more insight into what parts of scope graphs cause performance differences, if any. To create such a controlled setup, we must understand what kind of patterns can occur in scope graphs and why they occur. From this, a synthetic dataset can then be created.

To this end, we will analyse the scope graphs for the three projects and extract some common patterns, which will then be linked to the name binding mechanism that created them. Finally, benchmarks are performed on a synthetic dataset that is based on the analysed patterns, which will give language designers the tools to understand the performance implications of certain name binding mechanisms.

5.1.1 Dataset

We reuse the same dataset that previous work focusing on query resolution performance has used [5, 2, 17]. The dataset consists of three Java projects in varying sizes: the Apache Commons CSV, IO and Lang3 libraries. An existing Statix specification for Java 1.5 was used to type check these projects. This specification contains all Java 1.5 language features, except for enums and generics. Recall from Chapter 1 that Statix uses scope graphs internally; all three projects were type checked using scope graphs. This results in three pieces of data per project: the scope graph, the executed queries and results those queries produced.

Table 5.1: Patterns and a rough estimate of their occurrences found in the real-world dataset. Each library's scope graph also contains the Java 1.5standard library. The table has three sections for each library, the first row shows the total number of scopes in the scope graph.

Pattern	Count	Average size (n scopes)	Median size (n scopes)	Min size (n scopes)	Max size (n scopes)
Commons Lang3	103016				
Linear Chain	2379	4.98	5	4	9
Fanout	2512	11.43	6	2	225
Tree	319	24.11	3	2	3560
Diamond	1062	3.27	3	2	39
Cycle	0	0.00	0	0	0
Commons CSV	108801				
Linear Chain	5539	5.93	5	4	21
Fanout	2570	11.35	6	2	225
Tree	451	19.26	3	2	3620
Diamond	1313	3.22	3	2	39
Cycle	33	4.00	4	4	4
Commons IO	140769				
Linear Chain	22399	6.54	6	4	23
Fanout	3158	10.03	5	2	225
Tree	1166	11.87	2	2	3943
Diamond	2941	3.26	3	2	39
Cycle	354	4.00	4	4	4

Table 5.1 contains the patterns found in the scope graphs from the real-world datasets, which include the Commons Lang3, CSV and IO projects[1]. The table is divided into three sections, one for each of the projects. The leftmost column contains the pattern that was searched for, with the other columns indicating how many times it was encountered and how large the encountered patterns where. This table is a very rough estimate and the patterns that were searched for were found by visually inspecting the scope graphs. Finding the exact set of different patterns is not important, hence a visual inspection is adequate. Rather, this table informs us where to look for the next section, when we find what language feature corresponds to what pattern.

The patterns looked for were allowed to overlap, as we are only interested in a rough estimate of the occurrences, not an exact number. These patterns and their origin are discussed in more detail the next section.

5.1.2 Language feature to pattern

In van Antwerpen, Hendrik et al. [1] the authors discuss multiple examples of scope graphs for different languages. These languages include an STLC, an extended STLC with a record system and subtyping, Featherweight Java [23], and Featherweight Java with support for subtyping of classes.

This is used as a basis for the mapping of name binding structures in Java 1.5 to the corresponding scope graph. Each of these features create a specific pattern in the scope graph, which we can analyse.

These languages have three important features with respect to scope graphs: name declaration, (extensible) records/classes and an import system. They represent the possible patterns that can occur in a scope graph.

For this section, an example language is used that is similar to Java 1.5 in syntax. Visibility modifiers (`public`, `private`) are omitted, as these do not change the structure of the scope

[1]The cover of this thesis is a scope graph of the standard library that was compiled alongside the three libraries.

graph. Furthermore, we use a limited set of labels: lexical parents (P), declarations of any type (D), class extensions (E), interface implementations (I) and package imports (IMP). In reality, more labels are used, such as different labels for different types of declarations [15]. For simplicity we use fewer labels, as we are mostly interested in the patterns that form.

The rest of this section provides examples of modelling different language features using scope graphs and how these graphs are queried.

Name declaration

Name declaration is the action of binding some value to a name, such as a variable or a class. As was seen in Section 2.3, this can be modelled in scope graphs using two new scopes.

```
1    int a = 3;
2    class Stack {}
3    Stack s = new Stack();
```

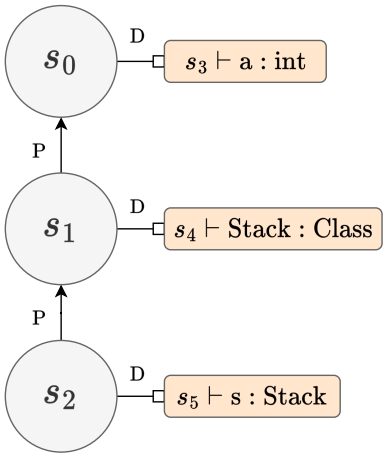


Figure 5.2: Example program declaring a few names (left) and the scope graph representation of it (right).

Figure 5.2 shows a small example program along with the scope graph representation of that program. Each name that is declared creates two new scopes: one representing the lexical child of the previous scope and another scope that holds the declared name. Lines 1, 2 and 4 declare a variable, while line 3 declares a class. In the scope graph, the class declaration uses the same label as variable declarations.

Name declarations result in a pattern of chained scopes, as every new declaration creates a new scope.

Function declaration

Function declaration is a feature found in many languages. A function is declared, along with its arguments. In most languages, the body of a function is in a separate scope and this is reflected in scope graphs.

```
1    int double(int x) {
2      return x + x;
3    }
4    int y = 3;
5    double(y);
```

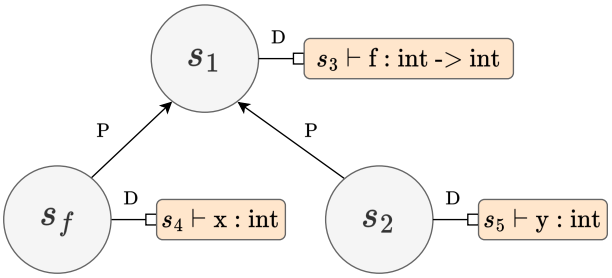


Figure 5.3: Example program with function declaration (left) and the scope graph representation of it (right). The root scope s_0 has been left out for brevity.

Figure 5.3 contains a small example with a function declaration and application. Line 1 actually does two things: it declares a function with type `int -> int` and binds this function to the name `double`. In contrast with the scope graph of the STLC seen in Chapter 2, the scope representing the function body has an edge with the scope that declares the function itself. This enables recursion. The result is a tree-like structure, as the function body's scope creates a new branch.

Compound datatypes

Compound datatypes are types that contain a collection of fields (and methods) that can be any type. For example, structs in C and Rust or classes in C++ or C#. In Java 1.5, compound types are named classes, which have members that are either fields or methods. A field is some variable that is stored within the class, while a method is a function that exists within the scope of a class. Methods have access to all members of the class they exist in.

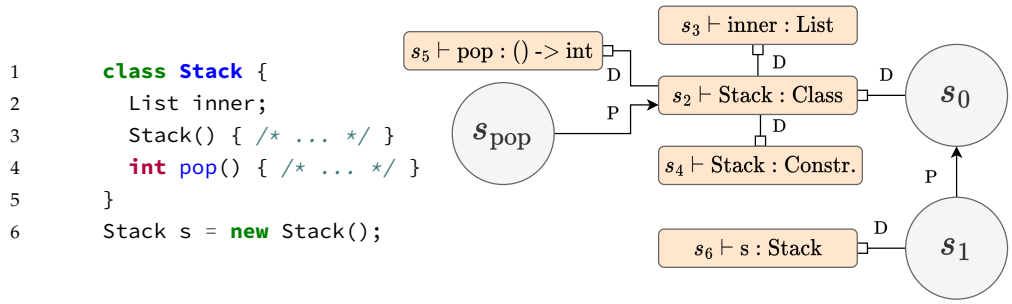


Figure 5.4: Example program with a class declaration (left) and the scope graph representation of it (right). The bodies for `Stack` and `pop` are left out for brevity.

Figure 5.4 shows an example of a `Stack` class that contains a constructor and some methods. The stack's scope is represented by s_2, which has the name of the class and the type `Class`. Each member of the class is declared within the class' scope. In the scope graph, this is represented with outgoing D edges from s_2. Furthermore, each class method generates two scopes, similarly to a function. In the figure this is shown for `pop()`, the body scope for the constructor is left out for brevity. The constructor is also given a special type `Constr.` to differentiate it from a normal method.

Inside class methods, all class members should be reachable. This is indeed the case in the scope graph, as s_{pop} has an outgoing P edge to s_2, meaning it can reach all class members.

Compound datatypes' field and method accesses require more than a single query to resolve. If in Figure 5.4 the method `s.pop()` was called, then two names must be resolved here. First, `s` must be resolved to be an instance of class `Stack` and second, `pop()` must be resolved to be a method of that class. When a class contains another class and a nested field access is performed, for example when doing `s.inner.length` in the example above, three queries are needed. One to resolve `s` to an instance of `Stack`, another one to resolve `inner` to an instance of `List` and finally a query to resolve `length` to a member of `List`. The query used to find a class name is different than the one used to find a class member, as different labels are allowed.

In scope graphs, compound datatypes result in a fan-out type pattern. For example, a class declaration results in a single scope with outgoing edges for each of its fields and methods. The type of class fields does not impact the pattern of the graph, only how the graph is queried. Since method bodies live inside a class, longer chains of scopes can also be found here, but this is in fact the same pattern as in Section 5.1.2.

(Multiple) Inheritance

Java 1.5 supports class inheritance and multiple inheritance in the form of interfaces. A class inheriting another class inherits all fields and methods from its parent (or superclass). The child class (or subclass) can also override methods for a more specialised implementation.

```
1   class ThreadSafeStack extends Stack {
2     ThreadSafeStack() {
3       /* ... */
4     }
5     @Override int pop() {
6       /* ... */
7     }
8   }
```

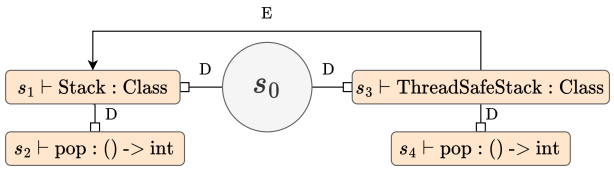


Figure 5.5: Example program with class inheritance and method overriding (left) and the scope graph representation of it (right). The method bodies are left out for brevity.

Figure 5.5 shows an example program where the previous `Stack` class is extended by a new class `ThreadSafeStack` (s_3). This class represents a stack that is thread safe to use, and overrides the `pop()` method to achieve that behaviour. In scope graphs, inheritance is modeled in two steps: creating a scope graph for the child class and creating an E edge to represent the superclass relation.

Overriding methods is done by redeclaring the method in the subclass' scope. Queries can then prioritise the overridden methods by specifying a label ordering $D < E$. Furthermore, the subclass can access all superclass fields and methods because the E edge can be followed. Queries must use the E label in their path regex to follow these edges.

Class extension results in a similar pattern to class declarations. Each class creates a fan-out type pattern and extensions link these together, which results in longer chains of scopes. Class fields can also be types which extend other classes, but this does not matter for the patterns found in the scope graph. Again, the type of fields only impacts the performed queries, not how the graph is structured.

Java 1.5 only allows a single parent class, in order to avoid the so called "Deadly diamond of death" [24], which refers to ambiguous names when two parent classes share a field with a grandparent class. For example, if two parent classes have a field called `x`, then the name `x` in the child class becomes ambiguous. However, Java 1.5 does support multiple inheritance through interfaces. While Java 1.5 does reject programs with ambiguous references because of multiple implemented interfaces sharing names, a scope graph would still have to be created to detect that.

```
1   interface Clearable {
2     void clear();
3   }
4
5   class Stack implements Clearable {
6     List inner;
7     Stack() { /* ... */ }
8     int pop() { /* ... */ }
9     // interface's methods
10    void clear() { /* ... */}
11  }
```

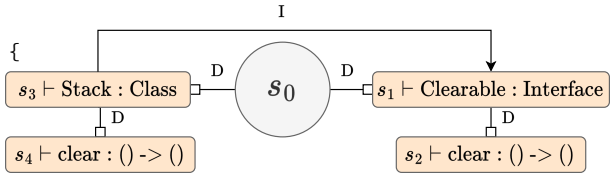


Figure 5.6: Example program with class inheritance and method overriding (left) and the scope graph representation of it (right). The scopes for the bodies of `pop` and `clear` are left out for brevity.

Figure 5.6 shows an example of how scope graphs model interfaces. This is actually very similar to how inheritance is handled. The only difference is that an I label is used instead

of an E label, to distinguish extended classes and implemented interfaces. This is needed to check if all methods from an interface are implemented, to give an example. Furthermore, `clear` inside the `Clearable` has no function body, so no scope exists for its function body. Note that in both the program and the scope graph, there is actually no difference between an implementation of an interface's method (`clear`) and an overridden method (`pop`).

```
1   interface Iterable {
2     Iterator iterator();
3   }
4
5   interface Summable extends Iterable {
6     int sum();
7   }
8
9   interface Countable extends Iterable {
10    int count();
11  }
12
13  class Stack implements Countable, Summable {
14    List inner;
15    Stack() { /* ... */ }
16    int pop() { /* ... */ }
17    Iterator iterator() { /* ... */ }
18    int count() { /* ... */ }
19    int sum() { /* ... */ }
20  }
```

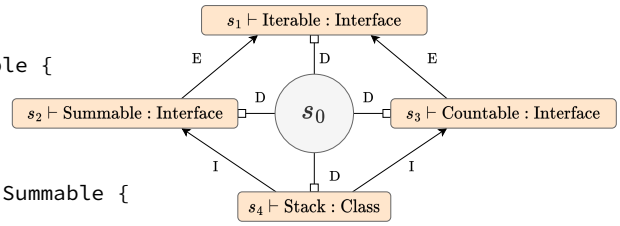


Figure 5.7: Example program with multiple interfaces inheriting each other (left) and the scope graph representation of it (right). The bodies of the (implemented) methods in `Stack` are left out for brevity.

Figure 5.7 shows a program where a class inherits two interfaces that both extend the same parent interface. In this example, the two interfaces `Countable` and `Summable` extend `Iterable`, and `Stack` implements both `Countable` and `Summable`. In a scope graph, this results in a diamond pattern, as both implemented interfaces have a separate edge to `Iterable`[2].

A Java 1.5 specific detail that is very important to mention is that all classes and interfaces extend the `Object` class [25]. This means a diamond pattern will form as soon as a class extends or implements two or more other classes or interfaces. This explains the large number of diamonds found in Table 5.1.

Namespaces and Imports

Namespaces allow programmers to compartmentalise names. In Java 1.5 namespaces are called packages and are commonly used, as only one class can be defined per file. Other classes must be imported from other namespaces.

Figure 5.8 shows a by now familiar example, except that `List` is now imported using `import java.util.List`. In Java 1.5, this means that the name `List` is imported from the `java.util` package. The scope graph representation of an import is very simple, it is a single edge between the scope that imports and the name that is imported. This edge is given a label that signifies it imports an item.

On its own, a single import does not create a very interesting pattern. However, when multiple imports are used, interesting patterns start to emerge. Two interesting patterns we observed occur with diamond imports and circular imports.

[2]Name ambiguity as a result of diamond patterns ("diamond problem") can be detected using scope graphs, but it is up to the language designers how this is solved. Java 1.5 chooses to make the implementor reimplement the method.

```
1  import java.util.List;
2  class Stack {
3    List inner;
4    Stack() { /* ... */ }
5    int pop() { /* ... */ }
6  }
```

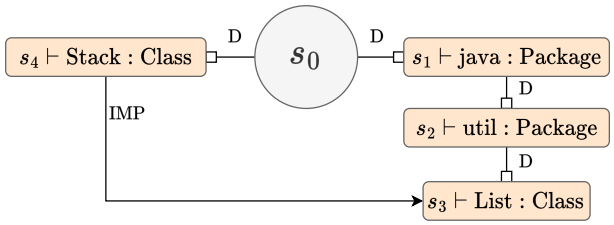


Figure 5.8: Example program with an import statement. On the right is the scope graph representation.

```
1   // File: ParentObject.java
2   import ChildObject;
3   class ParentObject {
4     ChildObject child;
5     ParentObject() { /* ... */ }
6   }
7
8   // File: Child.java
9   import ParentObject;
10  class ChildObject {
11    ParentObject parent;
12    ChildObject() { /* ... */ }
13  }
```

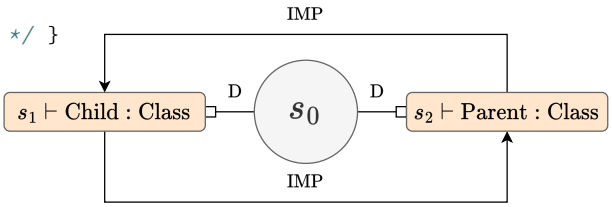


Figure 5.9: Example program with a circular dependency. On the right is the scope graph representation.

Figure 5.9 shows a program that contains a circular dependency. `ParentObject` holds a reference to its child, while `ChildObject` holds a reference to its parent. In a scope graph, this means that the scope representing the file `ParentObject.java` must import the scope representing `ChildObject.java` and vice versa. This results in a circular pattern in the scope graph.

5.2 Synthetic dataset

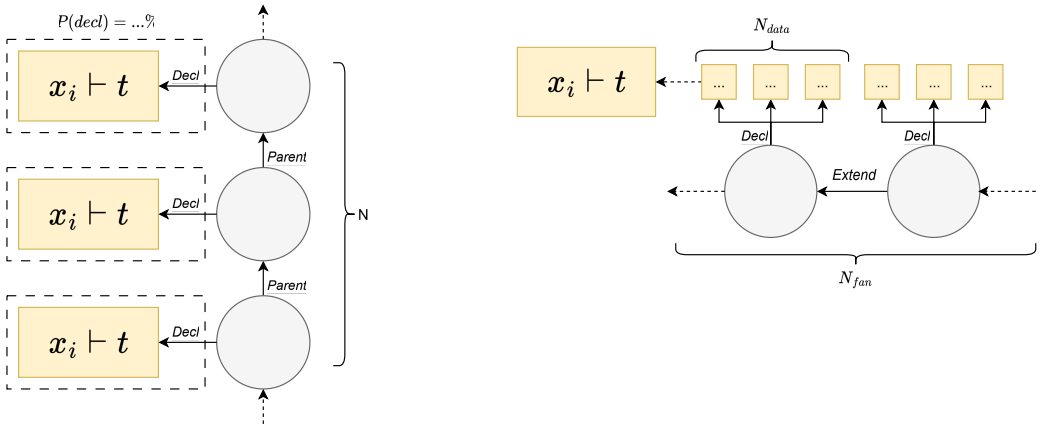
As explained previously, running a benchmark on a real scope graph is not very informative, which means that a synthetic dataset is needed. The goal of this synthetic dataset is that it represents portions of a real-world scope graph, such that it is possible to better analyse performance differences. Previously, some patterns were extracted that often occur in real scope graphs and these patterns serve as the basis for the synthetic dataset.

Since this scope graph does not represent any real language, the labels and data used are arbitrarily chosen to simulate real scope graphs. The labels used are P, E and D, which correspond to a lexical parent, a class extension and a declaration. These labels are common across different languages [1]. The data in this graph are typeless, named variables. This means that more complex types of queries, such as resolving function overloading, cannot be done in this dataset. However, that does not impact the benchmark significantly. The underlying query resolution that is performed for both simple and complex queries is the same, the difference is the parameters provided and the interpretation of the resulting environments.

The dataset consists of a number of *subjects*, which are all scope graphs. These subjects consist of the following three subgraphs: a *head*, a *body* and a *tail*, which together form a single scope graph. The head is the section of the scope graph where the data lives, the body are the patterns that are benchmarked and the tail is where queries start. In the coming sections, each of the chosen heads, tails and bodies are discussed. Many parameters are used to create the dataset, and these are summarised in Table 5.13

Head

The head of a subject is where the data lives in the scope graph. From the scope graphs found in the real-world dataset, it was found that most data declared were either class members or variables in a method. To simulate this, we use two different types of heads.



(**a**) Linear head

(**b**) Fanout head

Figure 5.10: Head sections in the synthetic dataset.

Figure 5.10 shows the two types of heads: the linear and fanout heads, which simulate a function body and classes respectively. The yellow boxes containing $x_i \vdash t$ represent a declaration of some name x_i with type t. Both types of heads use a different R_l and $L_<$, since different labels appear in both. The D_{WFD} and $D_\approx$ is the same for both heads; the D_{WFD} is defined as $D : x_i = y_i$, where x_i is the name data in the graph and y_i is some other name. The $D_\approx$ is defined as $\simeq_d$: $x_i = x_j$. For A_{mem}, an equivalent P_d is used: $P_d : name(d)$, where *name* is a function that returns the name of the given data. Both heads use a fixed size for both the number of scopes and the number of data declared. This is because the amount of data in the graph is expected to influence the performance results of A_{mem} and varying this randomly could make the results harder to interpret.

Variable names are generated randomly and can also overlap. It is possible for a head section to contain two declarations of the same name in different scopes. This allows the dataset to be used to test if variable shadowing works properly. x_s is the set of names that is queried for, which contains more names than there are generated in the graph. Every run, a name is randomly chosen from this set to query for. It is entirely possible that some of the names in the graph will not be queried for during a run, which is similar to an unused variable. The dataset can thus also be used to simulate shadowing, unused variables and undefined references.

The linear head (Figure 5.10a) is a chain of scopes, where each scope has a certain chance ($P(decl)$) to have declared data. This head contains $N_{head,lin}$ scopes and the edges between each scope uses a *Parent* label. As we have seen in Section 5.1.2, each declared variable or function creates a new lexical scope, and is represented in scope graphs by an additional scope, connected with a P label. The linear head, together with $P(decl)$ simulates both types of declarations happening in a random order. The randomness is implemented such that exactly $P(decl)$% of scopes contain a declaration. The difference between subjects with this head is not the amount of data, but only in what scopes it is declared. This head uses $P * D$ as its R_l and $D < P$ as its $L_<$.

The fanout head (Figure 5.10b) is a chain of scopes of length $N_{head,fan}$, where each scope represents a class. Each scope has a large number of declared data ($N_{fan,data}$), which repre-

sents all class fields and members. The classes are connected using *Extend* (E) labels, which represents inheritance. The fanout head simulates classes with inheritance, as found in the real-world dataset. This head uses P^*E^*D as its R_l and $D < P, E < P$ as its $L_<$.

The main difference between the two heads for query resolution is the amount of data in the graph. The fanout head has much more data per scope, and also more data in total when looking at the entire scope graph. For A_{mem} this can give insight into the performance of transferring the cache between scopes as the amount of data grows.

Tail

The final section, the tail, is the part of the subject where queries start. The tail is kept small relative to the body, as the goal is to measure performance based on the body, not the tail. The tail is the only part that varies randomly between each subject. The tail connects to the ”bottom” scope of the body. If there are multiple, such as for the tree, then every one of these scopes gets a separate tail.

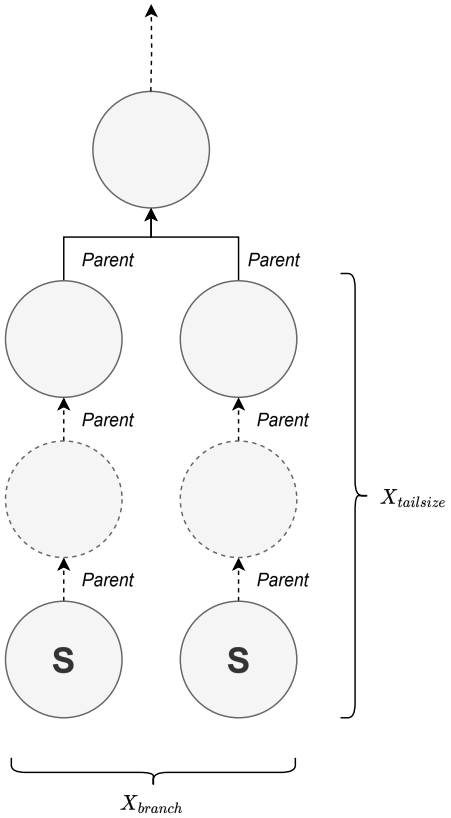


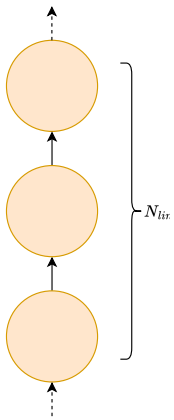
Figure 5.11: Tail section of a subject. The scopes marked with an ”S” are where queries can start.

Figure 5.11 shows the tail section of subjects. It is a tree with X_{branch} branches of length $X_{tailsize}$. The number of branches and the length of each branch varies randomly between each subject. The final distribution chosen for these can be found in Table 5.13. Each edge uses a P label.

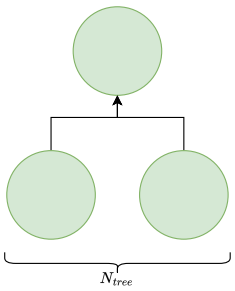
The tail is constructed in this way to simulate how queries are performed in the real-world dataset. The branches each simulate the body of a different method where queries start. Each run, queries start in a random branch. The reason the size of the tail is random is to introduce some noise to make the eventual performance benchmark more realistic. Furthermore, adding randomness removes some bias by preventing the compiler from optimising parts of the query away at compile time.

Body

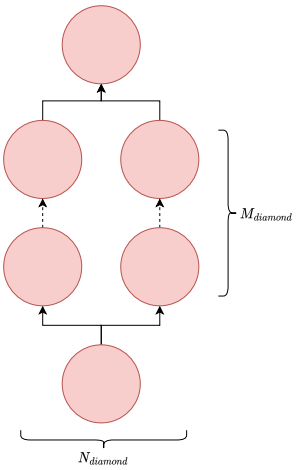
The body is the largest part of each subject and the part that the is measured for performance. The body consists of relatively large scope graphs that represent the various patterns discussed in Section 5.1.2. This is the part that will be used for performance measurements of A_{cur} and A_{mem} during the evaluation



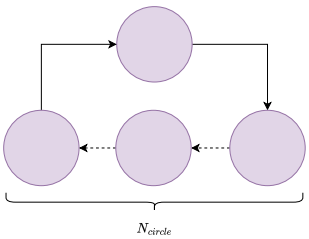
(**a**) Linear pattern.



(**b**) Tree pattern.



(**c**) Diamond pattern.



(**d**) Circular pattern.

Figure 5.12: Body sections of the synthetic dataset.

Figure 5.12 contains the patterns that are used for the body section of a subject. There are four in total, each with one or two variable size parameters: the linear, tree, diamond and circular patterns. The first two are very common in the real-world scope graph, while the latter two are rarer, but have interesting performance implications.

The linear pattern (Figure 5.12a) is a chain of scopes similar to the linear head, except with no declarations. N_{lin} refers to the number of scopes in this linear pattern. A long linear chain of scopes means that most of the work during query resolution is performed on graph traversal. A_{mem} will likely also spend a lot of time on transferring the cache down the scope graph.

The tree pattern (Figure 5.12b) is a tree of scopes where each branch has length 1. N_{tree} refers to the number of branches in this tree. This pattern is much shorter, but much wider compared to the linear pattern. Each query with A_{mem} has to always go to at least the bottom scope of the head, as that is the first shared scope between queries. With this pattern, queries

will have short resolution paths, meaning we evaluate whether caching is worth it for short paths.

Diamond patterns (Figure 5.12c) are a pattern where a single scope branches out into multiple scopes, and then converge back into a single scope again. $N_{diamond}$ refers to the number of branches of the diamond (width), while $M_{diamond}$ refers to the length of each branch in the diamond (height). For simplicity, every "branch" of the diamond will have the same height.

Diamond patterns are, as mentioned in Section 5.1.2, formed by inheritance. In Java 1.5 every class extends the `Object` class, meaning that only one other class or interface needs to be extended or implemented for a diamond pattern to form. This is emphasised, as the performance of diamond patterns can be very poor with A_{cur}.

The versions of A_{cur} and A_{mem} used in this thesis will always traverse the entire graph that the R_l allows. The only exception to this that only applies to A_{mem}, is when a valid cache entry already exists in a scope. In every other case the full graph is traversed.

Diamond patterns cause the graph to branch out into multiple scopes, that later converge into the same scope. This means that every single scope following a diamond pattern is evaluated $N_{diamond}$ times, resulting in a massive amount of duplicate work.

It is expected that A_{mem} will perform much better for these patterns, as a cache can be stored in the scope at the top of the diamond, where all branches converge. Only one of the paths through the branches will evaluate the graph following the diamond, while the other paths only read this cache. The trade-off is that the cache size will inflate instead of the execution time. Every path through the diamond is a separate cache entry.

Circular patterns (Figure 5.12d) are similar to linear patterns in that they are long chains of scopes. The only difference, as apparent by the name, is that circular patterns contain a cycle. Cycles are difficult to handle in a cache, as discussed in Section 4.4. To summarise, the solution to this complexity is to not store a cache in scopes that are in a cycle. This means that no cache is stored inside the body of this pattern. Effectively, most queries will then have to traverse through the body to reach the bottom scope of the head, which does have a cache entry.

5.2.1 Subjects

Using the head, body, and tail, a subject can be constructed.

Table 5.13: Parameters and their values for the synthetic dataset.

Variable	Value	Description
$P(decl)$	50%	Chance for declaration in linear head
$N_{head,lin}$	50	Length of linear head
$N_{head,fan}$	25	Length of fanout head
$N_{fan,data}$	15	Number of declarations per scope in fanout head
N_{lin}	$20, 40, 80$	Length of linear pattern
N_{tree}	$40, 80, 160$	Number of branches in tree pattern
N_{circle}	$20, 40, 80$	Length of circle pattern
$N_{diamond}$	$4, 8, 16$	Width of diamond pattern
$M_{diamond}$	$1, 2, 4$	Height of linear pattern
X_{branch}	6..12	Number of branches in tail tree
$X_{tailsize}$	2..10	Number of branches in tail tree

The parameters used when generating these subjects can be found in Table 5.13. A graph consisting of the body section with a filled in length is called a *variation*. For the diamond body, we have variations where either the width or height varies. The sizes are arbitrarily chosen, but are relatively large compared to the average sizes found in the real-world dataset.

Table 5.14: All generated variations with their sizes. A variation is constructed by filling in the size for one of the table rows. For the diamond, N refers to the width, while M refers to the height.

Head	Body	Body sizes
Linear	Linear	$[20, 40, 80]$
Linear	Tree	$[40, 80, 160]$
Linear	Diamond (width)	$N = [4, 8, 16]$, $M = 1$
Linear	Diamond (height)	$N = 1$, $M = [4, 8, 16]$
Linear	Circular	$[4, 16, 64]$
Fanout	Linear	$[20, 40, 80]$
Fanout	Tree	$[40, 80, 160]$
Fanout	Diamond (width)	$N = [4, 8, 16]$, $M = 1$
Fanout	Diamond (height)	$N = 1$, $M = [4, 8, 16]$
Fanout	Circular	$4, 16, 64$

Table 5.14 shows all generated variations. For all variations, we construct a subject by adding the head and tail sections. We create 100 subjects with the linear head, and 100 subjects with the fanout head. The tail section is randomly generated each time, using the parameters from Table 5.13.

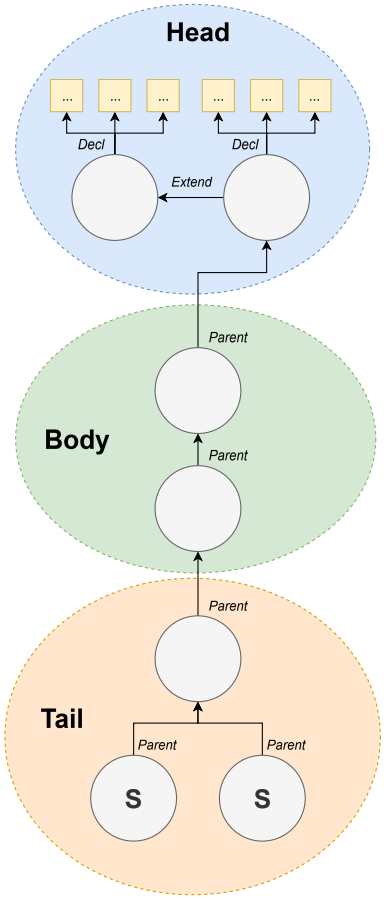


Figure 5.15: Example of a single subject, using a fanout head and a linear body. The parameters used are $N_{head,fan} = 2$, $N_{fan,data} = 3$, $N_{lin} = 2$ and the tail has two branches with length 1.

Figure 5.15 contains an example of a generated subject. Each of the scopes at the bottom of the body section has its own tail section. For the tree pattern, each branch has its own tail. The top of the body section is connected to the bottom scope of the head via a single edge. Both the edges from the tail to the body and body to the head use a parent label.

5.3 Conclusion

In this chapter, we have seen some of the more frequent patterns that occur in real world scope graphs of Java 1.5 programs. Using this information, we found the language features that are likely to have created those patterns. We created a synthetic dataset based on these patterns. Three distinct scope graph sections are used in generating this set: the head, body and tail. A variation is a combination of a head and a body with its size filled in. A subject is a variation with a randomly generated tail. In total, we generate 100 subjects for each variation. In the next chapter, both A_{cur} and A_{mem} are evaluated using this synthetic dataset and the obtained results are discussed.

Chapter 6

Evaluation

This chapter focuses on the evaluation of both Java 1.5 and A_{mem}. From the three research questions, this chapter answers two of them, namely:

1. $Q_{efficacy}$: Does A_{mem} produce the same results as A_{cur}?
2. $Q_{performance}$: How does the performance of A_{cur} compare to A_{mem} for the found patterns?
   a) Q_{speed}: How much faster is A_{mem} compared to A_{cur}?
   b) Q_{mem}: How much more memory does A_{mem} require compared to A_{cur}?

In the coming sections, all questions are explained in more detail and answered. The benchmarks ($Q_{performance}$) are performed on a synthetic dataset, not real-world scope graphs. The reason for this is explained in Chapter 5.

Both algorithms are reimplemented in Rust, to ensure that implementation details play no role in the performance difference between the two algorithms. The graph traversal portion of both algorithms is identical, which is why using the same implementation is preferable. This also means both algorithms use the same regex engine, label order matcher and (memory) representation of the scope graph. The difference between the two is then truly only the memoisation and changed query parameters.

6.1 Benchmark setup

The benchmarks consists of multiple *runs*. Each run does 1, 5 and then 10 query instances in a row, each instance starting in a randomly picked branch of the tail. A set of these queries instances is called a Q_{set}. The first query in each Q_{set} starts with an empty cache. Multiple queries are performed, otherwise the cache is not read. The single query case is also interesting, since that shows the overhead of the cache.

Three warm up runs are done per item, after which 5 runs are performed per Q_{set}. The final result is the average of these 5 runs. We record the following information: execution time of graph traversal, time spent accessing the cache, and the size of the cache. Furthermore, the results of the queries are compared and checked for equivalence.

As mentioned in Section 4.4, the performance of A_{mem} is measured with and without the check for circular graphs, as it is expected to be computationally expensive.

6.1.1 Hardware and Implementation details

The benchmarks are run on an Intel i5-13600kf @ 3.49 GHz on a single thread, one subject at a time. Both algorithms are written in Rust, and compiled using Rust 1.89. A_{cur} is about 300 lines of code, with the cache for A_{mem} adding about 200 lines of extra code. An open-source

repository containing the implementation and benchmarks can be found at *Achtuur/scope-graph* [26]. The algorithms were run using `cargo bench`, using the default benchmark profile. It was found that running the benchmark on multiple threads dramatically impacts results.

A_{cur} and A_{mem} use the same implementation of the algorithm, with the only difference being that A_{mem} has the cache enabled. This ensures that the efficacy of the cache is compared, and not implementation details between the two algorithms. To confirm this, the benchmark results will record time spent in the cache separately from the rest of the algorithm.

Furthermore, as mentioned in Section 4.4, A_{mem} has an edge case involving cycles, which is expected to be quite slow. The benchmark will also report this time separately.

6.2 Results

This section discusses the results obtained from the aforementioned benchmarks and answers both $Q_{efficacy}$ and $Q_{performance}$.

We expect A_{cur} to scale linearly in time with the number of queries performed in a row. After all, it does the same work every query instance. A_{mem} should be slower than A_{cur} for a single query, because of the overhead incurred by caching found data. When multiple queries are performed in a row, A_{mem}'s graph traversal time should stay relatively constant, while the cache access time should increase. This is because subsequent queries have to do far less graph traversal and can read from a cache entry. Furthermore, the cycle check is expected to be expensive.

Table 6.1: Table showing performance results for A_{cur} compared to A_{mem}. Speedups are marked in green, while slowdowns is marked in red. The column on the right shows after how many queries A_{mem} achieves a speedup of at least 1.0x.

Head	Pattern	A_{cur} 1/5/10 Queries (ms)	A_{mem} 1/5/10 Queries (ms)	Speedup 1/5/10 Queries	Break Even
Fanout	Tree ($N = 40$)	0.29 / 1.33 / 2.61	0.36 / 0.50 / 0.65	0.82x / 2.68x / 4.00x	2
Fanout	Tree ($N = 80$)	0.36 / 1.36 / 2.68	0.41 / 0.49 / 0.64	0.89x / 2.79x / 4.20x	2
Fanout	Tree ($N = 160$)	0.29 / 1.74 / 2.59	0.53 / 0.50 / 0.67	0.55x / 3.48x / 3.86x	2
Linear	Tree ($N = 40$)	0.06 / 0.30 / 0.59	0.08 / 0.10 / 0.12	0.73x / 3.03x / 4.87x	2
Linear	Tree ($N = 80$)	0.06 / 0.31 / 0.61	0.09 / 0.10 / 0.13	0.73x / 2.98x / 4.55x	2
Linear	Tree ($N = 160$)	0.06 / 0.30 / 0.61	0.08 / 0.10 / 0.13	0.74x / 2.90x / 4.89x	2
Fanout	Linear ($N = 20$)	0.59 / 2.61 / 5.11	0.69 / 0.75 / 0.80	0.85x / 3.48x / 6.43x	2
Fanout	Linear ($N = 40$)	0.80 / 4.25 / 7.85	1.00 / 1.07 / 1.10	0.80x / 3.97x / 7.13x	2
Fanout	Linear ($N = 80$)	1.34 / 6.14 / 12.29	1.70 / 1.73 / 1.89	0.79x / 3.55x / 6.51x	2
Linear	Linear ($N = 20$)	0.09 / 0.46 / 0.96	0.12 / 0.13 / 0.14	0.78x / 3.49x / 6.96x	2
Linear	Linear ($N = 40$)	0.12 / 0.65 / 1.25	0.17 / 0.19 / 0.20	0.74x / 3.42x / 6.36x	2
Linear	Linear ($N = 80$)	0.20 / 0.99 / 1.94	0.28 / 0.28 / 0.28	0.72x / 3.55x / 6.83x	2
Fanout	Diamond ($N = 4, M = 1$)	1.12 / 5.74 / 11.32	0.50 / 0.71 / 0.89	2.24x / 8.06x / 12.66x	1
Fanout	Diamond ($N = 8, M = 1$)	2.28 / 11.32 / 22.17	0.72 / 1.15 / 1.45	3.15x / 9.81x / 15.32x	1
Fanout	Diamond ($N = 16, M = 1$)	4.43 / 22.47 / 44.78	1.10 / 1.91 / 2.56	4.03x / 11.74x / 17.48x	1
Linear	Diamond ($N = 4, M = 1$)	0.25 / 1.22 / 2.36	0.10 / 0.13 / 0.14	2.47x / 9.50x / 17.25x	1
Linear	Diamond ($N = 8, M = 1$)	0.54 / 2.38 / 4.59	0.13 / 0.16 / 0.18	4.18x / 14.59x / 24.83x	1
Linear	Diamond ($N = 16, M = 1$)	1.09 / 4.62 / 9.08	0.15 / 0.24 / 0.28	7.16x / 19.52x / 32.16x	1
Fanout	Diamond ($N = 4, M = 1$)	1.12 / 5.74 / 11.32	0.50 / 0.71 / 0.89	2.24x / 8.06x / 12.66x	1
Fanout	Diamond ($N = 4, M = 2$)	1.24 / 5.93 / 11.75	0.57 / 0.81 / 1.04	2.17x / 7.35x / 11.35x	1
Fanout	Diamond ($N = 4, M = 4$)	1.33 / 6.24 / 12.53	0.71 / 1.07 / 1.04	1.86x / 5.82x / 12.07x	1
Linear	Diamond ($N = 4, M = 1$)	0.25 / 1.22 / 2.36	0.10 / 0.13 / 0.14	2.47x / 9.50x / 17.25x	1
Linear	Diamond ($N = 4, M = 2$)	0.26 / 1.26 / 2.43	0.12 / 0.16 / 0.15	2.13x / 8.03x / 16.63x	1
Linear	Diamond ($N = 4, M = 4$)	0.26 / 1.30 / 2.58	0.12 / 0.14 / 0.17	2.04x / 9.11x / 15.36x	1
Fanout	Circular ($N = 4$)	0.31 / 1.32 / 2.69	0.67 / 0.99 / 1.30	0.46x / 1.32x / 2.07x	5
Fanout	Circular ($N = 16$)	0.34 / 1.42 / 2.94	0.71 / 1.08 / 1.45	0.48x / 1.32x / 2.03x	5
Fanout	Circular ($N = 64$)	0.32 / 1.45 / 2.91	0.71 / 1.11 / 1.45	0.45x / 1.31x / 2.01x	5
Linear	Circular ($N = 4$)	0.06 / 0.31 / 0.62	0.18 / 0.23 / 0.27	0.34x / 1.31x / 2.30x	4
Linear	Circular ($N = 16$)	0.06 / 0.33 / 0.64	0.17 / 0.24 / 0.30	0.36x / 1.34x / 2.16x	4
Linear	Circular ($N = 64$)	0.08 / 0.40 / 0.80	0.19 / 0.33 / 0.42	0.40x / 1.22x / 1.90x	4

Table 6.1 shows the results of the benchmark for both A_{cur} and A_{mem}. The first two columns indicate the head and body of the subject, while the leftmost two columns show the execution time of 1, 5 and 10 queries for A_{cur} and A_{mem}, in that order. Figure 6.3, Figure 6.5 and Figure 6.4 show the table as graphs. The x-axis indicates the variation that was run and the y-axis shows the execution time. The numbers on top of the bars indicate how

Table 6.2: Table showing performance results for A_{mem} for all subjects. Caches that are at least 10 times larger are marked in red. The data is obtained with $Q_{set} = 10$.

Head	Pattern	Traversal Time (%)	Cache Time (%)	Circle Check Time (%)	Cache size (compared to Scope Graph)	Same result as A_{cur}
Fanout	Tree ($N = 40$)	95.28	4.72	0.00	3.94x (5.60 MB)	100%
Fanout	Tree ($N = 80$)	94.94	5.06	0.00	2.16x (5.62 MB)	100%
Fanout	Tree ($N = 160$)	95.24	4.76	0.00	1.14x (5.64 MB)	100%
Linear	Tree ($N = 40$)	89.24	10.76	0.00	0.63x (0.79 MB)	100%
Linear	Tree ($N = 80$)	85.82	14.18	0.00	0.32x (0.79 MB)	100%
Linear	Tree ($N = 160$)	88.91	11.09	0.00	0.17x (0.79 MB)	100%
Fanout	Linear ($N = 20$)	95.16	4.84	0.00	13.28x (8.70 MB)	100%
Fanout	Linear ($N = 40$)	95.45	4.55	0.00	19.62x (13.33 MB)	100%
Fanout	Linear ($N = 80$)	96.22	3.78	0.00	30.90x (22.50 MB)	100%
Linear	Linear ($N = 20$)	89.83	10.17	0.00	5.15x (1.05 MB)	100%
Linear	Linear ($N = 40$)	91.08	8.92	0.00	4.42x (1.42 MB)	100%
Linear	Linear ($N = 80$)	91.64	8.36	0.00	6.08x (2.26 MB)	100%
Fanout	Diamond ($N = 4, M = 1$)	92.44	7.56	0.00	13.96x (8.89 MB)	100%
Fanout	Diamond ($N = 8, M = 1$)	91.06	8.94	0.00	22.77x (14.64 MB)	100%
Fanout	Diamond ($N = 16, M = 1$)	89.11	10.89	0.00	39.91x (26.14 MB)	100%
Linear	Diamond ($N = 4, M = 1$)	89.17	10.83	0.00	5.71x (1.06 MB)	100%
Linear	Diamond ($N = 8, M = 1$)	89.73	10.27	0.00	7.94x (1.52 MB)	100%
Linear	Diamond ($N = 16, M = 1$)	90.07	9.93	0.00	11.99x (2.44 MB)	100%
Fanout	Diamond ($N = 4, M = 1$)	92.44	7.56	0.00	13.96x (8.89 MB)	100%
Fanout	Diamond ($N = 4, M = 2$)	93.53	6.47	0.00	15.28x (9.80 MB)	100%
Fanout	Diamond ($N = 4, M = 4$)	93.23	6.77	0.00	17.87x (11.64 MB)	100%
Linear	Diamond ($N = 4, M = 1$)	89.17	10.83	0.00	5.71x (1.06 MB)	100%
Linear	Diamond ($N = 4, M = 2$)	89.33	10.67	0.00	5.95x (1.13 MB)	100%
Linear	Diamond ($N = 4, M = 4$)	89.25	10.75	0.00	6.41x (1.28 MB)	100%
Fanout	Circular ($N = 4$)	48.51	2.29	49.20	7.41x (4.71 MB)	100%
Fanout	Circular ($N = 16$)	49.01	2.27	48.72	7.24x (4.71 MB)	100%
Fanout	Circular ($N = 64$)	52.64	2.42	44.93	6.64x (4.71 MB)	100%
Linear	Circular ($N = 4$)	41.09	3.92	54.99	3.70x (0.68 MB)	100%
Linear	Circular ($N = 16$)	46.38	4.13	49.48	3.43x (0.68 MB)	100%
Linear	Circular ($N = 64$)	61.11	4.40	34.50	1.94x (0.68 MB)	100%

many queries were run in a row. The purple bars show the execution time for A_{cur}. The execution time for A_{mem} is split up into three parts: graph traversal (green), cache accesses (yellow) and time spent doing the circular check (blue). The circular check was only enabled for the circular pattern, as that is the only pattern where cycles can exist.

Table 6.2 shows the results of the benchmarks for A_{mem} only. The first few columns indicate what variation was run, while the other columns show the performance results of that run. The results are shown for $Q_{set} = 10$, meaning 10 queries were performed in a row.

These results are discussed and interpreted in the coming sections.

6.2.1 Efficacy ($Q_{efficacy}$)

Before any potential performance improvements for A_{mem} can be evaluated, it is important to determine its efficacy. After all, the algorithm would be useless if the results it generates are different. We will check efficacy of A_{mem} by comparing its results with A_{cur}.

In Section 4.4, several limitations of A_{mem} were discussed. Most relevant here is the use of a P_d instead of the D_{WFD} and $D_{\approx}$. In short, the P_d is limited in its potential functionality compared to the D_{WFD} and $D_{\approx}$. For the benchmarks, we chose D_{WFD} and $D_{\approx}$ for which an equivalent P_d exists, namely one that compares the names of data. In the dataset we use, this is the common case [15]. Second, another limitation was discussed with regard to cycles in graphs. To summarise, A_{mem} does not store a cache in scopes that are in a cycle. This should not change the obtained environment, only the performance of the algorithm.

The rightmost column in Table 6.2 shows the percentage of runs for which the same environment was obtained from both A_{cur} and A_{mem}. As can be seen, these are the same for every query.

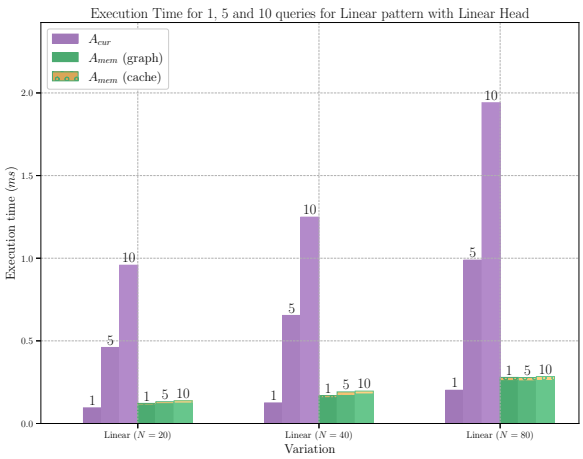
Furthermore, Statix has integration tests covering query resolution [27], which were reimplemented in the Rust implementation. All tests pass.

Summary To answer $Q_{efficacy}$, A_{cur} and A_{mem} produce the same environments in all performed runs in this benchmark, with a caveat that a D_{WFD} and $D_{\approx}$ must be chosen for which

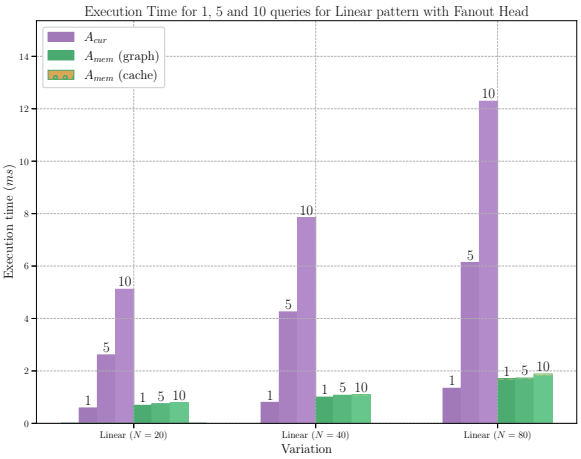
an equivalent P_d exists. Both algorithms also pass the reimplemented Statix test suite.

6.2.2 Performance ($Q_{performance}$)

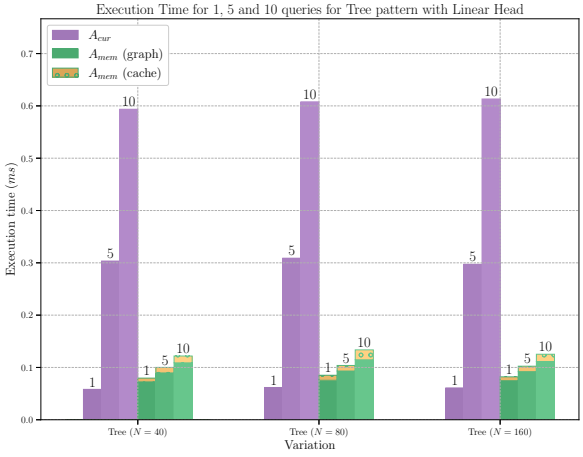
This section will discuss the performance results obtained from the benchmark and compare the results of both A_{cur} and A_{mem}.



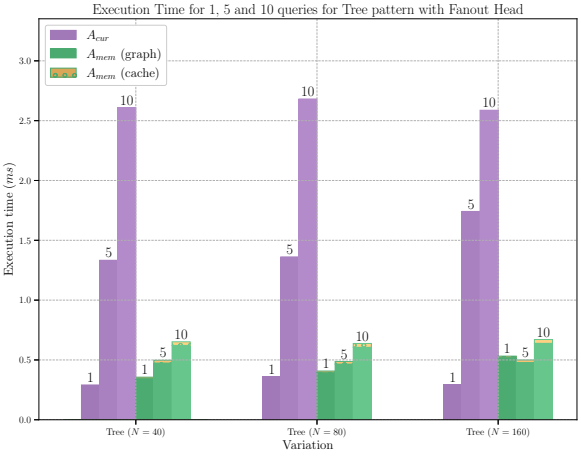
(**a**) Linear pattern with linear head.



(**b**) Linear pattern fanout head.



(**c**) Tree pattern with linear head



(**d**) Tree pattern with fanout head.

Figure 6.3: Evaluation results of the linear and tree patterns. On the x-axis is the pattern size. The number on top of the bars indicate the number of queries performed in a row, starting with an empty cache for A_{mem}.

Table 6.2 shows the cache size for A_{mem} after 10 queries were executed in the second to last column. The only difference in memory usage between A_{cur} and A_{mem} is the cache. This means that this column shows how much more memory A_{cur} uses compared to A_{mem}. Important to stress is that A_{mem} always uses *more* memory than A_{cur}. The worst results are obtained for the linear and diamond patterns. For linear patterns this is because the cache is stored in each scope and the pattern contains a large number of scopes. For the diamond pattern, each path through the diamond creates a new path, so the data is stored in the cache N times, where N is the width of the diamond. The cache size, and thus memory usage, increases with the number of scopes and data and this is why diamond patterns show an especially large

cache. The increased memory usage is acceptable, as the absolute cache size is still only in the megabytes and, as we will see, the speedup more than makes up for it.

Table 6.1 shows the execution times for both A_{cur} and A_{mem}, while Table 6.2 shows the performance of A_{mem} for the $Q_{set} = 10$ case.

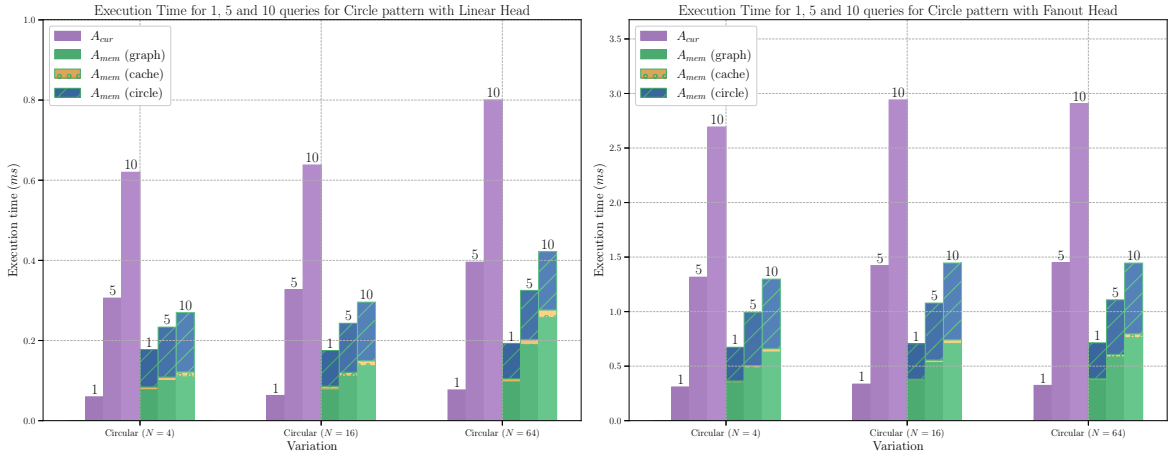
The execution times of A_{cur} seem to scale linearly with the number of queries performed in a row. This makes sense, as queries with A_{cur} do not share work, meaning every instance does roughly the same amount of work. The graph traversal time for A_{mem} (shown in green in Figure 6.3) stays almost constant in time with increasing number of queries. This is because only the first instance needs to do a full graph traversal; subsequent query instances read from the cache with minimal traversal.

Overall, A_{mem} outperforms A_{cur} for every variation in the $Q_{set} = 5$ and $Q_{set} = 10$ cases. For $Q_{set} = 1$, A_{cur} is faster for most benchmarked patterns. A_{mem} induces some overhead to construct the cache in the first query. However, subsequent query instances are able to read this cache, which is why A_{mem} is only faster when multiple queries are performed.

Recall from Section 5.2 that the linear head contains less data but is longer, while the fanout head is shorter but contains more data. Both algorithms consistently perform better with the linear pattern compared to the fanout pattern, because the latter contains more data. For A_{mem}, this difference is larger, because it has to also cache this data. The size of the cache is also much larger for the fanout head, because there is more data to cache.

The linear and tree pattern show expected results. Both patterns break even after 2 queries are performed, since all query instances past the first are able to access the cache. The tree pattern has much lower absolute numbers for the execution time, because the resolution paths are much shorter. It also has a relatively small cache, which is also due to the shorter paths. A shorter path means less scopes need to contain a cache, resulting in a smaller cache. The circular and diamond patterns deviate more from the median and are discussed separately.

Circular patterns



(**a**) Circular pattern with linear head.

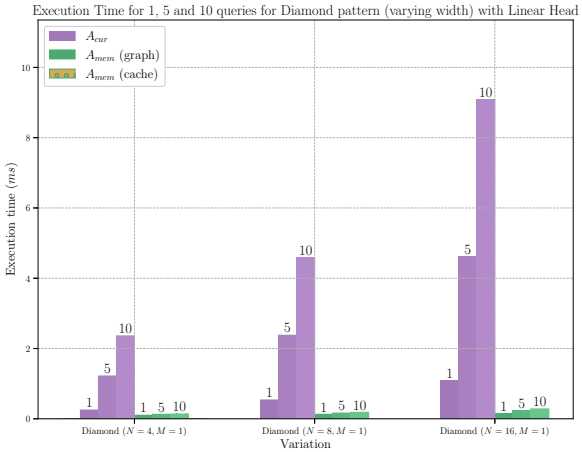
(**b**) Circular pattern with fanout head.

Figure 6.4: Evaluation results for circular patterns, with the cycle check enabled. On the x-axis is the pattern size. The number on top of the bars indicate the number of queries performed in a row, starting with an empty cache for A_{mem}.

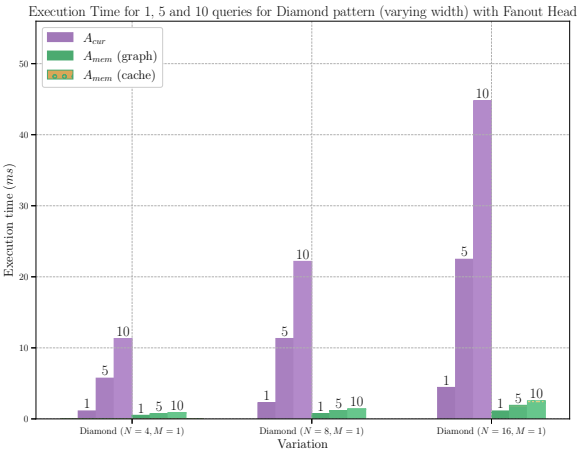
Cycles in graphs were handled separately, due to it being difficult to retain a coherent cache in that case (see Section 4.4). The solution was to not store a cache in scopes that exist

in a cycle. While this results in correct query resolution, this does introduce some inefficiencies. First, detecting cycles is relatively expensive compared to query resolution, and second, less caching means A_{mem} is less effective. Figure 6.4b and Figure 6.4a show the performance results for the circular pattern. The blue bar, showing the time it takes to perform this extra cycle check, is noticeably large and almost doubles the query resolution time for A_{mem}. Furthermore, not caching in scopes that are contained within a cycle means more time is also spent on graph traversal. Unlike the other patterns, the graph traversal increases linearly with the number of queries here. A_{mem} still breaks even with A_{cur} after 4 to 5 queries, which is because the bottom scope of the head contains a cache entry. The head is only traversed once.

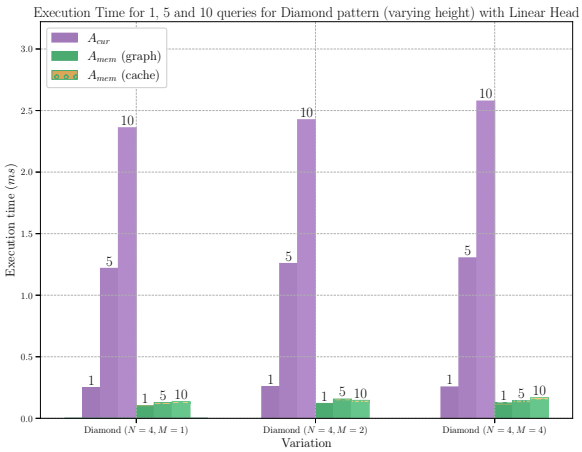
Diamond pattern



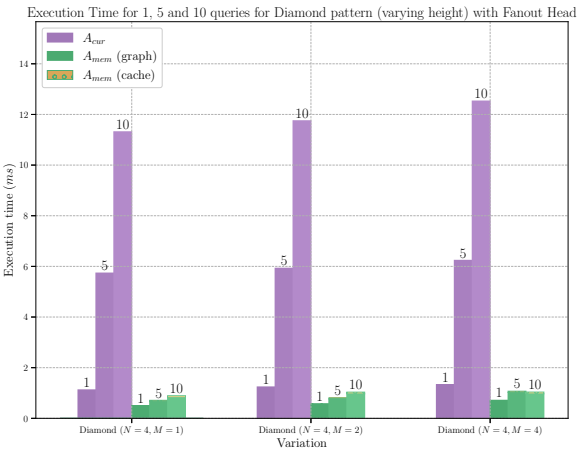
(**a**) Diamond pattern with linear head. The width is varied between 4 and 16.



(**b**) Diamond pattern with fanout head. The width is varied between 4 and 16.



(**c**) Diamond pattern with linear head. The height is varied between 1 and 4.



(**d**) Diamond pattern with fanout head. The height is varied between 1 and 4.

Figure 6.5: Evaluation results for diamond patterns. On the x-axis is the pattern size. The number on top of the bars indicate the number of queries performed in a row, starting with an empty cache for A_{mem}.

Figure 6.5 shows the results obtained from the benchmarks using the diamond pattern with varying width and height. The hypothesis in Section 5.2 was that A_{mem} would show

better performance for these patterns, because A_{cur} performs a large amount of duplicate work when querying diamond patterns. As the results show, this is indeed the case. A_{mem} consistently shows (much) better performance for all variations and number of queries. In contrast to the other patterns, even the single query is faster for A_{mem}. The trade-off for the increase in performance is an increase in memory usage. Each of the paths through a diamond are separately stored in the cache, resulting in comparatively large caches, as Table 6.2 shows. The diamond patterns consistently show the largest cache sizes.

Summary To answer $Q_{performance}$, we must answer both Q_{mem} and Q_{speed}. The memory usage of A_{mem} differs per pattern, but is always larger than A_{cur}. The exact number depends on the size of the scope graph and the number of resolution paths that exist. For example, a relatively small graph, such as for the tree patterns with a linear head, results in a memory usage increase 1.63 to 1.17. Larger graphs with many resolution paths, such as the diamond pattern with a fanout head and a width of 16 result in 40.91 times more memory being used. However, when looking at the cache size in bytes, it is definitely acceptable when considering the performance improvements. A_{mem} is faster than A_{cur} after multiple queries are executed for all tested patterns. The more queries are performed, the faster A_{mem} becomes relative to A_{cur}. For the linear and tree patterns, only 2 queries are required to break even with A_{cur} and the speedup after 10 queries ranges from 4 to 7. There are two notable exceptions: diamond patterns and circular patterns. In circular patterns, A_{mem} still shows a speedup after 5 queries are performed, but performs comparatively worse because of required checks that ensure correct query resolution in case cycles exist. For circular patterns, the speedup ranges from 1.90 to 2.30 after 10 queries. A_{mem} always performs better for diamond patterns, even in the single query case, showing speedups between 11.35 to 32.16.

6.3 Threats to Validity

There are three main threats to the validity of the evaluation, two internal threats and one external, which are discussed here.

Internal threats: A_{mem} uses a data projection function as a query parameter instead of the data well-formedness and data equivalence. As discussed in Section 4.4, the data projection is more limited and in this evaluation we chose a data well-formedness for which an equivalent data projection function exists. If an equivalent data projection function did not exist, then the efficacy of A_{mem} could decrease. However, as we have seen before, most queries can be transformed to use an equivalent data projection function. If this is not the case, the choice could be made to fall back to A_{cur} for these specific queries.

In this evaluation we used the same implementation for both A_{cur} and A_{mem}, with the only differences being the query parameters and whether caching was enabled. This was done to more clearly see the difference the cache makes. The algorithm could have been more optimised for A_{cur}, which would impact results. Furthermore, since the algorithms were reimplemented in Rust, instead of using the existing, battle-tested implementation, there could be subtle bugs. We believe that using a more optimised version for A_{cur} would make the evaluation less meaningful. We are interested in the difference the cache makes for query resolution performance, not specific implementation details. Furthermore, we have tested A_{mem} using the Statix test-suite and found that it passes all of them. This gives us sufficient confidence in the implementation of A_{mem}.

External threats: The synthetic dataset that is used is based on a visual inspection of a real scope graph. We did a rough counting of the patterns found by this inspection. Since these patterns occur from language features, there is admittedly a bias in what patterns were looked for. Furthermore, there is also a bias towards Java 1.5, since we used a Java 1.5 scope graph. Because we linked language features to scope graph patterns (Section 5.1.2), we be-

lieve these biases are minimised, since the language features found are common in languages other than Java 1.5.

6.4 Conclusion

In this chapter, we have benchmarked A_{cur} and A_{mem} using the synthetic dataset (see Section 5.2) and answered $Q_{efficacy}$ and $Q_{performance}$.

The goal of the benchmark was to measure the performance difference memoisation makes for the subjects found in the synthetic dataset. The benchmark consisted of doing 1, 5 and 10 queries in a row. Each run measured both the efficacy and the efficiency of the memoised algorithm. We found that A_{mem} produces the same results as A_{cur}. While A_{mem} uses more memory, it is found to be faster after multiple queries have been performed.

Chapter 7

Related work

This section discusses work related to topics discussed in this thesis, primarily related to scope graphs. Most of the published works regarding scope graphs is discussed, ranging from history to its applications.

7.1 History of Scope graphs and Query resolution

Many programming languages have some form of naming, where names can be declared in one part of the program and referenced later on. Resolving these references to declarations is name resolution. According to Neron et al. [4], the name resolution rules of real programming languages are usually tailored to a particular language, even when formalised. To bridge the gap between formalisation and real-world implementation, they introduced scope graphs. As described previously, scope graphs are a formalism that models name binding structures in programming languages. In its first iteration, scope graphs were more limited, as concepts like labels did not exist yet. Instead, edges between scopes represent a parent relation to model lexical scope. Different relations, such as declaration and imports, are built into scope graphs. References are also part of the graph, and name resolution is done by finding a matching declaration for a given reference. Compared to queries, this gives much less control. For example, the rules used for shadowing are fixed for the entire scope graph instead of per query.

Later work by van Antwerpen, Hendrik et al. [16] shows how to use scope graphs to build static semantic analysers. They extended the scope graph framework with uniqueness and completeness constraints to express properties such as duplicate declarations and record initialisation. Furthermore, edge labels were introduced here to model different relations between scopes, such as (transitive) imports. The parent relation is now modeled using a separate label (P in this thesis), for example.

The work by van Antwerpen, Hendrik et al. [1] introduces many concepts to scope graphs that were also referenced in this thesis. One of the most important ones is the notion of scoped relations, which model the association of types with declaration and the representation of explicit substitutions in the instantiation of parameterised types. They also introduced querying to scope graphs, which resolves scoped relations. This work has shown that scope graphs can be used to model a wide range of type systems, greatly increasing their applicability. This is also the birth of Statix, a DSL for the specification of static semantics, which is based on scope graphs.

In later work, Rouvoet et al. [12] addresses the unsoundness of name resolution in the case of unstable query answers in Statix, in order to schedule type checkers. When a language has a module system, a query on the scope graph can only give correct results if the scope graph for this module is complete. The results of a query thus depend on the evaluation order of the language, which in the then-current version of Statix results in unsound name

resolution. Their main contribution is the concept of critical edges in scope graphs, which seeks to solve the unsoundness.

7.2 Spoofax & Statix

Spoofax was developed as a language workbench for DSLs with, by then, state-of-the-art IDE support [28].

Statix is a meta-language that can be used to declaratively specify type systems [1] and is supported by Spoofax. Statix was created because traditional approaches to type system specifications do not reflect that many languages have similar name binding mechanisms. According to van Antwerpen, Hendrik et al. [1], many name binding representations are optimised for a specific language, be it a sequence of name-type associations for a simply typed lambda calculus or class tables to represent the class types of Featherweight Java. These optimisations obscure the commonality of the underlying name binding mechanisms. Furthermore, in many languages, operationalising name binding in a type checker requires carefully traversing the abstract syntax tree to collect information before it is needed.

The history of Statix mirrors the history of scope graphs quite closely, and as such the challenges it has faced are similar as the challenges for scope graphs described in the previous section. The main downside of Statix currently is its performance, which is the result of scope graph query resolution (see Section 3.3).

The scope graph data that was used in this paper was extracted by compiling Java projects using a Statix specification for Java.

7.3 Improving Statix Performance

As we have seen in Chapter 1, type checkers generated by Statix are relatively slow and in the past, some works have focused on improving this. The main difference with the work in this thesis is that the core query resolution algorithm is not changed, only the way it is used.

Van Antwerpen and Visser [5] introduces a new framework for implementing hierarchical type checkers that provide parallel execution between compilation units. It does so using scope graphs, and introduces the concept of scope states, which are used to track completeness of a scope. If a scope in incomplete, then a query between two compilation units must be paused. They implemented this new algorithm in Statix, marking it as the state-of-the-art query resolution algorithm at this point. This is a parallel version of A_{cur} and A_{mem} should be mostly compatible with the parallel algorithm, with some precautions needed to prevent data races when accessing the cache.

Zwaan [17] adds incrementality to scope graph based type checkers. Incremental type checkers reuse previous analysis in case the source code is unchanged. This means less time is spent on type checking, and response times for Integrated Development Environments (IDEs) goes up. In this work, some techniques are applied to discover what parts of the scope graph must be rebuilt if the source code changes. The optimisation to A_{cur} applied here is using it less, but it does not change the algorithm itself.

Zwaan [2] attempts to speed up scope graph query resolution using partial evaluation. They introduce an intermediate language that makes scope graph traversal and partial query result combination explicit and introduce a specialiser to translate queries to this new language. Because this intermediate language is more explicit, some optimisations become possible such as eliminating common subenvironments. Similarly to Van Antwerpen and Visser [5], this should also be compatible with A_{mem}. The underlying query resolution algorithm is not different, only the order in which (parts) of queries are performed.

7.4 Name resolution

This section will look at work related to name resolution outside of scope graphs.

Tymchuk et al. [9] identifies a problem where IDEs that support multiple languages must have multiple implementations of name resolution for each language. This is similar to the problem that was identified in Neron et al. [4], which birthed scope graphs. Their approach involves creating an Abstract Syntax Tree (AST) metamodel (called FAST) that attempts to be generic over a wide range of languages. Name resolution then works in two steps. First, they find candidate entities that could map to a name (lookup), and second, they select the actual candidate the name refers to (selection). The first part is generic, while the second part needs language-specific details as it requires information on how scoping works in that language. Scope graphs are similar in a sense, where the framework is generic over the syntax of a language, but requires more language-specific information in the form of query parameters. The difference in name resolution lies in that FAST first performs the candidate lookup in its entirety, and only afterward does selection, while scope graph queries perform (analogues to) the lookup and selection in each scope that is traversed.

Another interesting case is the work done in Konat et al. [10], which introduces NBL or NaBL (Name Binding Language), a DSL that declaratively describes name binding and scoping rules. The study of NaBL is actually what led to scope graphs [4]. Name resolution in NaBL is done in three phases: an annotation phase, a definition site analysis phase, and a use site analysis phase. In the first phase, each entry in the AST is annotated with a unique id based on its namespace and path. The second phase traverses the AST to get local information about all definitions, such as types. In the final phase, all references are resolved using the information gathered in the previous phases.

GitHub has recently adopted stack graphs [11], which are heavily based on scope graphs. The key issue they are trying to solve is name resolution at an incredibly large scale; their framework must be incremental, particularly file-incremental due to how git commits modify only a small fraction of files. While incremental compilation is possible with scope graphs [17], it is incremental over compilation units, rather than individual files. Their proposed solution are stack graphs, which is a graph that relates scopes to each other by maintaining a stack of symbols. Name resolution then works similarly to scope graphs, namely by finding a valid path through the graph. Stack graphs also do not seem to store type information.

7.5 Application of scope graphs

This section will discuss works that use scope graphs to achieve a goal not directly related to Statix or improving scope graph performance.

This thesis has focused mostly on scope graphs as a way to model name binding structures in static typing. However, scope graphs can also be used for dynamic typing with the *scopes as frames* paradigm [29]. By organising frames in correspondence with the statically known scope graph, name binding can be performed at runtime. A frame is a unit of memory with a slot (mapping from a name to a value), and links to other frames. The slots and links correspond to scopes and edges in the scope graph, creating a runtime analog of the scope graph. Bach Poulsen et al. [30] argues for writing definitional interpreters using dependently-typed languages. These interpreters operate on intrinsically-typed ASTs, in which well-typed programs can be expressed. No type checker is needed, as the program is guaranteed to be well-typed and the language in which the interpreter is written preserves the static semantics. In this work, scope graphs and the previously described *scope as frames* paradigm are used to create a intrinsically-typed encoding of scope graphs. With this, they constructed a library for structuring intrinsically-typed definitional interpreters.

DynSem is a high-level meta-DSL for specifying dynamic semantics of a language [31]. In early prototypes, the specifications could be executed by means of an automatically generated Java-based AST interpreter. Generating such an interpreter and compiling it caused long turnaround times however, and the authors chose to make DynSem interpret specifications directly instead of compiling them. While this resulted in short turnaround times, the downside is suboptimal runtime performance. In subsequent work, DynSem was expanded to provide support for Just In Time (JIT) compilation [32]. DynSem was later specified using scope graphs and the *scopes as frames* paradigm to perform more optimisations [33].

Chapter 8

Conclusion

The goal of this thesis was to introduce memoisation to scope graph query resolution and measure its performance using a dataset that is language-agnostic. To do this, we have analysed the current query resolution algorithm in scope graphs (A_{cur}) and formulated a memoised variant (A_{mem}). While we have shown that A_{mem} produces the same results as A_{cur}, there are some limitations to this new algorithm. The query parameters were changed to support caching, which means that A_{mem} does not support all the queries that A_{cur} does. Furthermore, the cache for A_{mem} uses more memory. However, A_{mem} supports the common case and has shown large performance increases over A_{cur} for our constructed synthetic dataset.

We identified some patterns found in real-world scope graphs and linked those back to the name binding structure that created them. From this, we created a synthetic dataset to evaluate A_{cur} and A_{mem}. The dataset gave insight into the performance of specific patterns and found that A_{mem} performs much better on diamond patterns. For Java 1.5, this could prove to be beneficial, as diamond patterns form faster for Java 1.5 scope graphs.

8.1 Limitations

This section will discuss several limitations surrounding this thesis that could and should be addressed in future work.

In Section 5.1.1 we have only looked at a limited set of real-world scope graphs. We have looked at three scope graphs of Java 1.5 projects, to find what patterns appear in scope graphs for our synthetic dataset. This view is limited, as scope graphs for different languages might show very different patterns. For example, scope graphs for C are likely significantly different from the Java 1.5 scope graph we found, as there is a large difference in name binding structures between the two languages. Different patterns might be more ubiquitous in C, or not exist at all, we cannot say. There might be patterns we have missed, or we might have given certain patterns too much emphasis based on the limited data available.

8.2 Future work

This section discusses future work that can be done, related to this thesis.

A_{cur} used in this thesis is a simplified version of the query resolution algorithm used in the Statix compiler. Many other additions and improvements [12, 5, 2, 17] have been made around query resolution that are not included in this thesis. While we expect A_{mem} to be compatible with these other additions to query resolution, some unforeseen complications could exist.

A_{mem} was implemented in Rust in a separate environment from where the Statix compiler currently lives. This was done for easy of experimenting and to have more control over the benchmarks. An obvious next step would be to implement A_{mem} into the Statix compiler,

as performance evaluations in this thesis seem positive. As mentioned previously, many improvements around query resolution have not been considered, and thus some challenges remain. The largest challenge is likely integrating the cache into the parallelised version of query resolution introduced in Van Antwerpen and Visser [5], as the cache will have to be made thread-safe.

Furthermore, future work can use and refine the synthetic dataset created in this thesis. It gives more fine-grained results compared to rerunning all queries for a "real" scope graph. The patterns in this thesis were focused on only a scope graph of a Java 1.5 project and the pattern set can be expanded with scope graphs from different languages.

The evaluation showed two clear areas of improvement for A_{mem}: the cache size and the cycle check. The cache is quite large and some patterns, such as diamond patterns, can make it grow even larger. In this thesis, we chose to store a cache in every scope encountered during query resolution, since theoretically a query can start in every scope. In practice however, this is usually not the case. Some scopes' cache might never be read because another scope exists that "blocks" query resolution from ever reaching it. Furthermore, smaller query resolution paths are not too costly in terms of performance, so fewer scopes can contain a cache without impacting performance too much, freeing up more memory.

The cycle check is another large bottleneck for this algorithm. It can be turned off for graphs that are guaranteed to not contain cycles, but the real-world scope graphs observed in this thesis do contain cycles. Not caching within a cycle is a correct solution, but it is also suboptimal. Future work could re-analyse this problem to find potential solutions to invalidating cache entries. Furthermore, the cycle check can be improved by only considering scopes that are visible for the current query instance.

Bibliography

[1] van Antwerpen, Hendrik et al. "Scopes as types". In: *Case Studies for Article: Scopes as Types* 2 (OOPSLA Oct. 24, 2018), 114:1–114:30. DOI: 10.1145/3276484. URL: https://dl.acm.org/doi/10.1145/3276484.

[2] Aron Zwaan. "Specializing Scope Graph Resolution Queries". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2022. New York, NY, USA: Association for Computing Machinery, Dec. 1, 2022, pp. 121–133. ISBN: 978-1-4503-9919-7. DOI: 10.1145/3567512.3567523. URL: https://dl.acm.org/doi/10.1145/3567512.3567523.

[3] B. F. Janssen. "Bootstrapping the statix meta-language". In: (2023). URL: https://repository.tudelft.nl/record/uuid:6e52270b-16e0-4052-897c-844ee5951863 (visited on 01/08/2025).

[4] Pierre Neron et al. "A Theory of Name Resolution". In: *Programming Languages and Systems*. Ed. by Jan Vitek. Vol. 9032. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 205–231. ISBN: 978-3-662-46668-1 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_9. URL: http://link.springer.com/10.1007/978-3-662-46669-8_9.

[5] Hendrik Van Antwerpen and Eelco Visser. "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers". In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 1:1–1:29. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.1. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.1.

[6] *Option Summary (Using the GNU Compiler Collection (GCC))*. URL: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html (visited on 09/22/2025).

[7] *Warn-by-default Lints - The rustc book*. URL: https://doc.rust-lang.org/beta/rustc/lints/listing/warn-by-default.html#unused-assignments (visited on 09/22/2025).

[8] *IBM Wazi Developer for Red Hat CodeReady Workspaces*. Apr. 16, 2021. URL: https://www.ibm.com/docs/en/wdfrhcw/1.4.0?topic=compiler-errorswarnings (visited on 09/22/2025).

[9] Yuriy Tymchuk et al. "Generic Name Resolution for Specific Language Models". In: (Aug. 14, 2014), pp. 131–140.

[10] Gabriel D P Konat et al. "Language-Parametric Name Resolution Based on Declarative Name Binding and Scope Rules". In: (2013).

[11] Douglas A. Creager and Hendrik van Antwerpen. "Stack graphs: Name resolution at scale". In: *OASIcs, Volume 109, EVCS 2023* 109 (2023), 8:1–8:12. ISSN: 2190-6807. DOI: 10.4230/OASIcs.EVCS.2023.8. arXiv: 2211.01224[cs]. URL: http://arxiv.org/abs/2211.01224.

[12] Arjen Rouvoet et al. "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Knowing when to Ask: MiniStatix implementation and case studies* 4 (OOPSLA Nov. 13, 2020), 180:1–180:28. DOI: 10.1145/3428248. URL: https://dl.acm.org/doi/10.1145/3428248.

[13] *Statix Resolver Implementation*. GitHub. URL: https://github.com/metaborg/nabl/blob/master/statix.test/scopegraphs/nameresolution.spt (visited on 09/26/2025).

[14] *scopegraphs crate Name Resolution Implementation*. URL: https://github.com/metaborg/rust-scopegraphs/blob/main/scopegraphs/src/resolve/lookup.rs (visited on 09/26/2025).

[15] *Spoofax Java Statix Spec*. GitHub. URL: https://github.com/metaborg/java-front/blob/e68358094dc198875b8b2652d703e441da2932e6/java8.spoofax3/src/java/names/TypeNames.stx (visited on 09/22/2025).

[16] van Antwerpen, Hendrik et al. "A Constraint Language for Static Semantic Analysis Based on Scope Graphs". In: (2016). URL: https://repository.tudelft.nl/record/uuid:59d81879-3226-408e-9485-25d429562000 (visited on 07/15/2025).

[17] Aron Zwaan. "Incremental Type-Checking for Free". In: (2022). URL: https://repository.tudelft.nl/record/uuid:e668432d-3881-4e19-9862-36f0d4b42226 (visited on 07/15/2025).

[18] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. Jan. 21, 2015. DOI: 10.48550/arXiv.1008.1459. arXiv: 1008.1459[cs]. URL: http://arxiv.org/abs/1008.1459.

[19] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/321239.321249. URL: https://dl.acm.org/doi/10.1145/321239.321249.

[20] Scott Owens, John Reppy, and Aaron Turon. "Regular-expression derivatives re-examined". In: *Journal of Functional Programming* 19.2 (Mar. 2009), pp. 173–190. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796808007090. URL: https://www.cambridge.org/core/product/identifier/S0956796808007090/type/journal_article.

[21] Aron Zwaan and Casper Bach Poulsen. *Defining Name Accessibility using Scope Graphs (Extended Edition)*. July 12, 2024. DOI: 10.48550/arXiv.2407.09320. arXiv: 2407.09320[cs]. URL: http://arxiv.org/abs/2407.09320.

[22] *Commons IO Overview – Apache Commons IO*. URL: https://commons.apache.org/proper/commons-io/ (visited on 10/17/2025).

[23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 1, 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: https://dl.acm.org/doi/10.1145/503502.503505.

[24] Robert C. Martin. "Java and C++ A critical comparison". In: (Mar. 9, 1997). URL: https://web.archive.org/web/20051024230813/http://www.objectmentor.com/resources/articles/javacpp.pdf (visited on 09/22/2025).

[25] *Object (Java 2 Platform SE 5.0)*. URL: https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html (visited on 09/22/2025).

[26] *Achtuur/scope-graph*. URL: https://github.com/Achtuur/scope-graph (visited on 10/28/2025).

[27] *Statix Name Resolution Integration Tests*. URL: https://github.com/metaborg/nabl/blob/master/statix.test/scopegraphs/nameresolution.spt (visited on 09/22/2025).

[28] Lennart C.L. Kats and Eelco Visser. "The Spoofax language workbench". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. SPLASH '10: Systems Programming Languages and Applications: Software for Humanity. Reno/Tahoe Nevada USA: ACM, Oct. 17, 2010, pp. 237–238. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869592. URL: https://dl.acm.org/doi/10.1145/1869542.1869592.

[29] Casper Bach Poulsen et al. "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics". In: (2016).

[30] Casper Bach Poulsen et al. "Intrinsically-typed definitional interpreters for imperative languages". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018), pp. 1–34. ISSN: 2475-1421. DOI: 10.1145/3158104. URL: https://dl.acm.org/doi/10.1145/3158104 (visited on 10/27/2025).

[31] Vlad Vergu, Pierre Neron, and Eelco Visser. "DynSem: A DSL for Dynamic Semantics Specification". In: *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Ed. by Maribel Fernández. Vol. 36. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 365–378. ISBN: 978-3-939897-85-9. DOI: 10.4230/LIPIcs.RTA.2015.365. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.RTA.2015.365 (visited on 10/27/2025).

[32] Vlad Vergu and Eelco Visser. "Specializing a meta-interpreter: JIT compilation of dynsem specifications on the graal VM". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes - ManLang '18*. the 15th International Conference. Linz, Austria: ACM Press, 2018, pp. 1–14. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237018. URL: http://dl.acm.org/citation.cfm?doid=3237009.3237018 (visited on 10/27/2025).

[33] Vlad Vergu, Andrew Tolmach, and Eelco Visser. "Scopes and Frames Improve Meta-Interpreter Specialization". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 4:1–4:30. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.4. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.4 (visited on 10/27/2025).

Acronyms

STLC Simply-Typed Lambda Calculus

regex regular expression

NFA nondeterministic finite automaton

DSL Domain-Specific Language

AST Abstract Syntax Tree

IDE Integrated Development Environment

JIT Just In Time