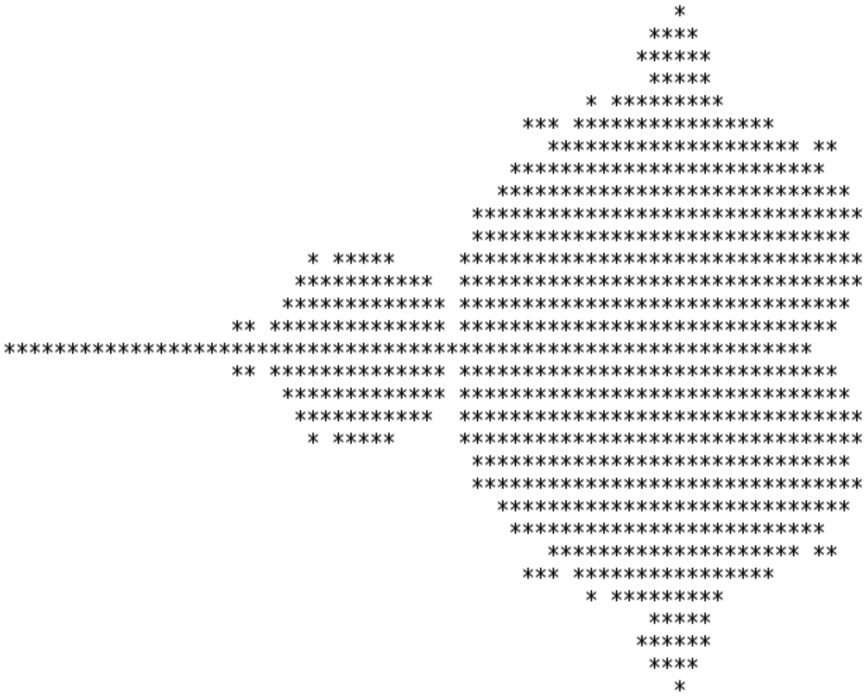
Incrementalizing Statix

A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs



Taico Aerts

Incrementalizing Statix

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Taico Aerts
born in Drachten, the Netherlands

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

Cover picture: The first published picture of the Mandelbrot set.
Brooks, Robert, and J. Peter Matelski (1978). “The dynamics of 2-generator subgroups of PSL (2, C).” In: *Riemann surfaces and related topics: Proceedings of the 1978 Stony Brook Conference (State Univ. New York, Stony Brook, NY, 1978).* Vol. 97, pp. 65-71.

Incrementalizing Statix

Author:	Taico Aerts
Student id:	4345797
Email:	T.V.Aerts@student.tudelft.nl

Abstract

Statix is a language which generates a type checker from a declarative specification. However, Statix is not fast enough for quick feedback in IDEs because it always has to reanalyze all files. In this thesis, we improve the analysis time of Statix by applying the ideas of separate compilation to create a model for incremental analysis. Statix uses a scope graph for representing the scoping and declarations of a project. We split the scope graph over multiple smaller modules which can be analyzed in isolation. Our model automatically detects dependencies between modules and supports creating scope graph diffs to determine modules affected by changes with high precision. The modules from our model can be solved in parallel, already yielding performance improvements of up to 40% compared to the original implementation. Finally, we implement an optimistic strategy with our model and show that it is effective whenever changes are small, reducing analysis time by up to 5 times for large projects.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. R. J. Krebbers, Faculty EEMCS, TU Delft
Committee Member:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Daily Supervisor:	Ir. H. van Antwerpen, Faculty EEMCS, TU Delft

Preface

Taico Aerts
Delft, the Netherlands
September 25, 2019

I would like to thank many people for their time and help during this process. I could not have completed this thesis without their sharp minds, valuable feedback and undying support.

First and foremost, I would like to thank Eelco for his supervision and especially for the late evening meetings with me even when he had no time to spare. I thank Hendrik for the helpful ideas, many sparring sessions and the collegial way he approached me.

I would also like to thank many friends for their feedback and honest criticism, in particular Bram, Luka, Chiel, Kevin, Rens, Caro and Marieke. Thank you for going through many versions of my thesis and always having useful observations.

I would like to thank my family, especially my parents and my sister Lody, for having patience with me and for their continued interest in my progress. In addition, I would like to give special thanks to my pake and beppe for their compassion and encouragement.

In the same manner I would like to thank my roommates and good friends Kevin and Rens, for their support throughout this process and all the little ways they've helped me, both in an academic way and otherwise. Without their help, this thesis would not have been there.

I would like to thank Marieke for her help with the planning of my thesis, all the good advice that she gave on the structure of my thesis, but most of all for her continuous faith that I would succeed.

And lastly, I would like to thank Caro for her unrelenting support, her faith in me and being there when I did not have much faith myself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Type checking is a technique used in many programming languages as a way of guaranteeing a certain level of correctness of code. Type checkers can be integrated with Integrated Development Environments (IDEs) to offer real time feedback to developers. Having feedback on the correctness of the code from a type checker has been shown to reduce the amount of bugs in the code [19]. However, domain specific languages (DSLs), in contrast to general purpose languages, are lacking in this area. For DSLs, it would be beneficial to have some form of type checking and IDE support. Unfortunately, creating a type checker for a programming language is complex and conventionally requires a lot of work. In addition, this effort often needs to be specialized to the target language in order to be fast enough for real time analysis, which means that reusing existing type checkers is more difficult.

Meta-languages address this problem by offering ways to create static analysis tools from declarative rules such as parsing, type checking, name binding, static analysis and compilation. A meta-language is a language which describes some aspect of another language. For example, a meta-language for syntax can create a parser from the description of a grammar. A language designer can then spend effort on describing the syntax of their language instead of spending effort on creating a parser. Some meta-languages also integrate with IDEs to offer the IDE functionalities that conventional languages enjoy.

Statix is such a meta-language in which users can express the typing and name binding rules of a language in a declarative way [3]. Statix then automatically generates a type checker from those rules. With Statix, language developers can create a type checker for their language without having to do a lot of engineering. Statix differentiates itself from other type checking approaches by using scope graphs to represent the scoping of programs. Scope graphs are a novel way to represent scoping and name binding. As such, it has not been used a lot in type checkers so far. More research is necessary to determine how existing optimization techniques can be applied to Statix.

While analyzing single files is possible in real time, i.e. in the order of seconds, with scope graph based type checkers like Statix, analyzing a project consisting of multiple files quickly becomes a problem. When programming projects grow to the size of tens to hundreds of files, we have found that analysis of projects with Statix becomes too slow, in the order of tens of seconds to minutes. While techniques like concurrency and code optimizations which cause linear speedups certainly help, they do not solve the underlying problem. Analyses of entire programs do not scale because the runtime of these analyses is linear in the size of the program [43]. If the project grows too large, type checking the entire code base will take too long for real time feedback. Instead, type checkers and compilers usually apply techniques such as incrementality to reduce the amount of work [4].

Incremental computation is the practice of only recomputing the information that depends on the changed data [8]. It reduces the amount of work by only analyzing the parts of the project that are affected by a change. The effects of the changes to these parts, if any, are then propagated throughout the project until we end up with the same solution as when we would have done a complete analysis. By keeping analysis up to date as changes are made, changes are kept small and the overall impact is limited [21]. This makes incrementality an effective technique to make type checkers fast enough to run in real time. The trade-off with incrementality is that initial analysis will take more time than if incrementality is not used. This is due to the additional bookkeeping that is required to be able to perform incremental analyses.

This thesis investigates different ways to improve analysis time of type checkers that use scope graphs, focussing on incrementality in particular. It is our aim to find a general solution to achieve incrementality out-of-the-box, i.e. without having to rely on large conceptual changes to the original type checker. We apply the idea of separate compilation to Statix in order to create a proof-of-concept implementation of an incremental type checker using scope graphs. Finally, we use this implementation to evaluate the effectiveness and applicability of these techniques in different scenarios.

As such, the central research topic of this thesis is *"How can we improve analysis time of scope graph based type checkers such as Statix?"*. While the focus will be on the Statix language, the techniques used are also applicable to other type checkers based on scope graphs. The main research topic is split up into four research questions as follows.

- **Research Question 1: What are existing techniques to improve analysis time as found in the literature?**
  We look at literature for existing techniques used by type checkers and compilers to improve analysis time. We investigate if and how we could apply these techniques for incremental analysis in scope graph based type checkers (Section 3.5).

- **Research Question 2: Based on the conclusions drawn from the literature, how can we create a model for incremental analysis for Statix?**
  Our goal is to create a model for Statix which supports different types of incremental analysis, called strategies. First, we try to apply the concept of modularization to Statix, to create modules which are mapped to the modules of the language we are analyzing. Then, we use these modules as the unit for incrementality by identifying dependencies and by adding features to our model to determine the impact of changes (Section 5.5).

- **Research Question 3: How can we apply concurrency to Statix?**
  With concurrency, we can directly improve analysis time, as long as our task is parallelizable. We try to show that we can apply concurrency to our model in the form of parallel execution of multiple modules (Section 7.2.5).

For our model, we identify two families of strategys. We choose to implement an optimistic strategy, which is based on the assumption that changes will have a small impact. We call this strategy the optimistic name-based strategy. This leads to Research Question 4.

- **Research Question 4: How well is our optimistic name-based strategy suited for incremental analysis?**
  We analyze the incremental effectiveness of our optimistic name-based strategy in different scenarios. With different scenarios we mean we apply changes which are common when editing a project in an IDE. We identify those scenarios that work well or poorly and identify the reasons for this behavior. Finally, we will compare the incremental analysis time to the non-incremental analysis time for differently sized Java projects (Section 7.3.4).

1.1 Contributions

The contributions of this thesis are the following:

- We introduce a model for incremental analysis for Statix based on the ideas of separate compilation (Chapter 4 and Chapter 5).
- We extend the semantics of Statix-core to support modules and split scope graphs and show that our approach retains the soundness that Statix-core enjoys (Section 4.4).
- We extend the implementation of Statix with our incremental model (Section 4.3).
- We extend our model and its implementation with support for concurrency in the form of solving multiple modules in parallel (Section 5.4).
- We design an implement an optimistic name-based incremental strategy for our model (Chapter 6).
- We evaluate the effectiveness of our optimistic name-based strategy on differently sized Java projects (Chapter 7).

1.2 Overview

This thesis is structured as follows. First, Chapter 2 explains what Spoofax and Statix are in more detail and introduces the core concepts of Statix relevant for the other chapters. Then, Chapter 3 discusses different approaches found in literature to achieve improved analysis time and how these approaches can be applied to scope graph based type checkers. Next, Chapter 4 explains how projects can be split into multiple modules and how Statix can be adapted to analyze multi-module projects. Chapter 5 describes our model for incremental analysis in Statix and the different features we support which allow for a range of different incremental strategies. This is followed by Chapter 6 which explains how the features of our model can be combined into strategies. We present two strategies for incremental analysis and explain why certain types of strategies are better suited to our approach. Chapter 7 follows by evaluating the effectiveness of our optimistic name-based strategy on different incremental changes. Finally, Chapter 8 concludes this thesis and describes future research opportunities opened by our work.

Chapter 2

Spoofax and Statix

This chapter explains Statix in detail and introduces the concepts that are relevant for this thesis. It also describes the formal syntax and semantics of Statix. A first time reader might consider skipping the formal specifications and revisit them when they become relevant in other chapters. This chapter is built up as follows. First, Section 2.1 describes what Spoofax is and how Spoofax supports meta-languages. Then, Section 2.2 introduces Statix itself. Section 2.3 explains the concept of scope graphs and how we can do name resolution with them. Section 2.5 discusses how Statix uses constraint solving and unification to do type analysis. Then, Section 2.4 explains how the type system of a language can be described in Statix specifications and how they are built up. Finally, Section 2.6 describes the current architecture of Statix.

2.1 The Spoofax Workbench

Spoofax is a language workbench for creating domain specific languages and meta-languages [25]. Spoofax aims at creating an IDE for a language from declarative specifications of its syntax and semantics. From these specifications, Spoofax can generate parsers, type-checkers, interpreters and compilers for a language. In addition, Spoofax integrates with Eclipse to offer IDE functionalities such as syntax highlighting, formatting, auto completion and static analysis feedback from these same declarative specifications.

Spoofax offers these functionalities through multiple different meta-languages. A meta-language is a language which describes some aspect of a language. For example, the meta-language SDF3 offers the parsing, coloring, formatting and auto completion functionalities of Spoofax. Statix is a meta-language for Spoofax which offers type checking and name binding functionalities.

2.2 Statix

Statix is a meta-language for the specification of static semantics in Spoofax [3]. With Statix, language designers can specify the type system and the name binding rules of their language in a declarative way. From the specification and the AST of a program, Statix constructs a scope graph to represent the name binding structure of the program while at the same time checking types in the program with this scope graph. In addition to giving feedback to the user, the scope graph and type checking information is also made available to other meta-languages in Spoofax. This means it can be used by, for example, a compiler or an interpreter to make their lives easier. We explain scope graphs in more detail in Section 2.3.

The rules of a language are expressed in a Statix specification (Section 2.4). These rules are defined in terms of constraints. Statix takes as input the AST of a program and a specification for a language, and combines those to create a set of constraints. These constraints are then solved by the Statix solver, constructing scope graphs for the input program and performing type checking at the same time. We explain the constraint solving of Statix in more detail in Section 2.5.

2.3 An Introduction to Scope Graphs

Scope graphs were introduced by Néron et al. [34] and further refined by Antwerpen et al. [2]. Scope graphs offer a language independent way of representing the naming structure of programs. A scope graph represents the lexical scoping of a program. A node in the scope graph is either a scope or a declaration. Scope graphs are essentially a mapping from the structure of a program to a graph. The different components of a scope graph are as follows.

- A *scope* is a node in the scope graph. It is an abstraction of a program region or set of AST nodes that behaves uniformly with respect to name resolution. In diagrams, scopes are represented with circles with a number in them, which represents the identity of the scope.

- A *declaration* is an occurrence of an identifier which introduces a name. We write declarations as x_i where x is the name being introduced and where i denotes the position of the identifier in the program. In diagrams, declarations are represented as boxes with their identifier in them.

- An *edge* is a connection of two scopes in the scope graph. An edge is directed and labelled. Language designers can use labels to distinguish between different types of edges. For example, the label I could be used to describe an import edge. In diagrams, edges are represented as arrows with a label. In text, edges are represented as $s \xrightarrow{l} s'$, i.e. scope s has an edge to scope s' with label l.

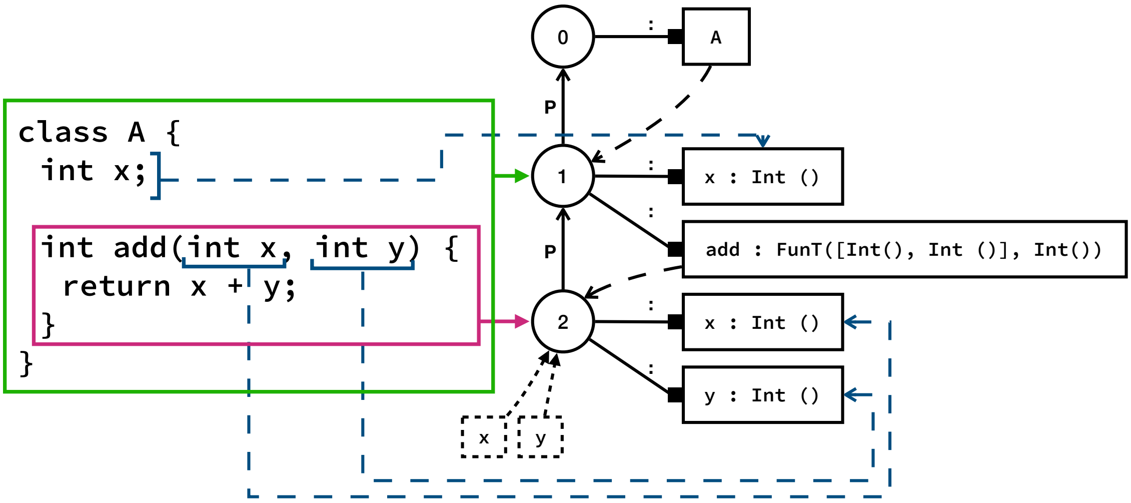
- A *data edge* is a connection of a scope to a declaration. A data edge has a direction and a label just like other edges. In diagrams, data edges are represented as a labelled line with a box instead of an arrow. In text, data edges are represented as $s \overset{r}{-\!\!\blacksquare}\, t$, i.e. scope s has an edge to a term t with relation label r.

An example of a scope graph is given in Figure 2.1. On the left, a Java program is shown, while on the right the corresponding scope graph is shown. The scope labeled with a 0 is the global scope, i.e. the scope in which globally available information is stored. The notation used in figures is shown in Figure 2.2.

2.3.1 Name Resolution with Scope Graphs

Name resolution is the process of finding the declaration to which a name application in a program refers [21]. Name resolution in scope graphs happens in the form of queries. A query is a question to retrieve certain declarations from the scope graph. For example, a query can search for the declaration of a class A whenever it encounters a reference to A. Queries in Statix can specify which declarations should be retrieved, which paths can be followed and the order in which paths should be considered. A query always starts in a specific scope, and then attempts to find paths in the scope graph to a matching declaration. In the scope graph, we show queries in the form of references. If relevant, the path followed by the reference is highlighted.

The paths that can be followed are represented in the form of a regular expression. Each edge in the scope graph has a label that can be matched by this regular expression. For example, in Figure 2.1 there is a reference for x and a reference for y. The paths that may be followed by those queries can be described as $\mathsf{P}^{?}$:. This means that the query is optionally



```
class A {
  int x;
  int add(int x, int y) {
    return x + y;
  }
}
```

Figure 2.1: A simple Java program (left) and its corresponding scope graph (right)

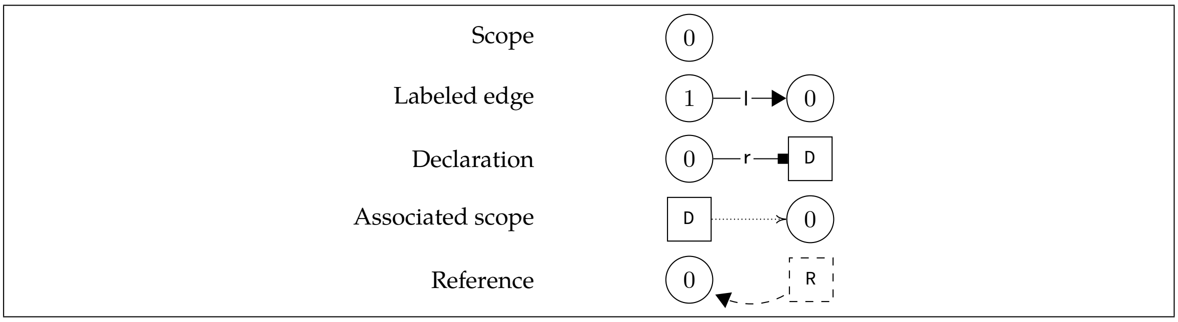


Figure 2.2: Overview of the notation used for scope graphs.

allowed to follow a single P edge and then a : edge. In this example this would give two matching paths, i.e. to the x parameter in the method with the path : and to the field x in the class with the path P :. As such, the query also needs a ordering to determine which reference is the correct one. For java, local declarations shadow outer declarations, so the ordering is described as : $<$ P, i.e. prefer : edges before P edges.

2.3.2 Formal Specification

A first time reader might want to skip the formal semantics and only revist them when they become relevant later in the thesis.

The formal syntax for scope graphs is given in Figure 2.3. The syntax and resolution calculus of Statix presented here are as defined in Antwerpen et al. [3], with minor changes to notation. This section is a summary of the relevant formal semantics from section 2.1 of the paper.

Resolution Calculus The resolution calculus of Statix is presented in Figure 2.4. There are three different judgements relevant to the formation of paths. The judgement $\mathcal{G} \vdash p : s_1 \twoheadrightarrow s_2$ states that scope s_2 can be reached from scope s_1 via path p, if there is a sequence of edges

Syntax Parameters

data terms	$d \in \mathcal{D}$	a set of data terms
labels	$l \in \mathcal{L}$	a set of edge labels
relations	$r \in \mathcal{R}$	a set of relation names

Syntax Definitions

scopes	$s \in S$	:=	some finite set
edges	$e \in E$	::=	$s \xrightarrow{l} s$
data	$d \in D$	::=	$s \overset{r}{\longrightarrow\!\!\blacksquare} d$
scope graphs	$\mathcal{G} \in Graphs$	::=	$\langle S_{\mathcal{G}}, E_{\mathcal{G}}, D_{\mathcal{G}} \rangle$
paths	$p \in P$	::=	$s \mid s \cdot l \cdot p$
extended labels	$\hat{l} \in \hat{\mathcal{L}}$	:=	$\mathcal{L} \cup \{\$\}$ where \$ indicates the end of a path

Figure 2.3: The syntax of scope graphs as used throughout this thesis. The syntax of scope graphs was originally presented by Antwerpen et al. [3]. This version includes minor modifications in notation.

starting at s_1 and leading to s_2. The path p records the scopes and edge labels that it passes through. Cyclic paths are not allowed.

The judgement $\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash s \overset{r}{\rightarrowtail} d$ states that the data term d can be reached with path p from scope s via relation r. The label well-formedness predicate *WFL* restricts which paths are valid. The data well-formedness predicate *WFD* ensures that the data term matches what we are looking for. For example, to specify that we are looking for a class with a specific name rather than for all classes.

The last judgement, $\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \xmapsto{r} d$ expresses the visibility of data terms. It states that data term d is visible from scope s under relation r via path p. In addition to label well-formedness and data well-formedness, an ordering on data and on labels is specified. The label ordering specifies whenever a path p should be chosen over a path p'. Only the smallest paths according to the label ordering are reported. The data ordering specifies which declarations shadow each other whenever there are multiple declarations reachable via equally small paths.

2.4 Defining Type Systems with Statix

The type system of a language can be described in Statix by giving a Statix specification of that language. A Statix specification consists of three aspects. First, it contains a description of the signature of the AST of the object language (Section 2.4.1). Second, it has a description of name resolution rules for the language (Section 2.4.2). Finally, there are rules which map the different elements of the AST to constraints (Section 2.4.3).

2.4.1 Signature of the AST

The first aspect of a Statix specification is the signature of the AST of the object language. It describes the different sorts and constructors of the AST elements of the language. We give an example of a signature for a small language with classes and fields, with a very basic type system in Listing 2.1. A corresponding syntax definition is given in the comments.

Visibility Parameters

data term well-formedness	$WFD \subseteq \mathcal{D}$	
label well-formedness	$WFL \subseteq \mathcal{L}^*$	defined as a regular expression
data order	$\leq_d \subseteq \mathcal{D} \times \mathcal{D}$	partial order
label order	$<_l \subseteq \hat{\mathcal{L}} \times \hat{\mathcal{L}}$	strict partial order

Path Well-formedness $\boxed{WFL \vdash p \text{ OK}}$

$$\frac{(l_1 \ldots l_n) \in WFL}{WFL \vdash (s_1 \cdot l_1 \cdot \ldots \cdot s_n \cdot l_n \cdot s_{n+1}) \text{ OK}}$$

Visibility Order $\boxed{<_l \vdash p <_p p}$

$$\frac{<_l \vdash p_1 <_p p_2}{<_l \vdash s \cdot l \cdot p_1 <_p s \cdot l \cdot p_2} \qquad \frac{\$ <_l l}{<_l \vdash s <_p s \cdot l \cdot p} \qquad \frac{l <_l \$}{<_l \vdash s \cdot l \cdot p <_p s} \qquad \frac{l_1 <_l l_2}{<_l \vdash s \cdot l_1 \cdot p_1 <_p s \cdot l_2 \cdot p_2}$$

Paths $\boxed{\mathcal{G} \vdash p : s \twoheadrightarrow s}$

$$\text{(NR-Refl)}\frac{s \in \text{scopes}(\mathcal{G})}{\mathcal{G} \vdash s : s \twoheadrightarrow s} \qquad \text{(NR-Cons)}\frac{s_1 \xrightarrow{l} s_2 \in \text{edges}(\mathcal{G}) \quad \mathcal{G} \vdash p : s_2 \twoheadrightarrow s_3 \quad s_1 \notin \text{scopes}(p)}{\mathcal{G} \vdash s_1 \cdot l \cdot p : s_1 \twoheadrightarrow s_3}$$

Reachability $\boxed{WFD, WFL, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d}$

$$\text{(NR-Rel)}\frac{\mathcal{G} \vdash p : s \twoheadrightarrow s' \quad s' \overset{r}{\longrightarrow\!\blacksquare} d \in \text{data}(\mathcal{G}) \quad WFL \vdash p \text{ OK} \quad d \in WFD}{WFD, WFL, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d}$$

Visibility $\boxed{WFD, WFL, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\longmapsto} d}$

$$\text{(NR-Vis)}\frac{\begin{array}{c} WFD, WFL, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d \\ \nexists p'd'. \left(\left(WFD, WFL, \mathcal{G} \vdash p' : s \overset{r}{\rightarrowtail} d'\right) \wedge (<_l \vdash p' <_p p) \wedge (d' \leq_d d)\right) \end{array}}{WFD, WFL, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\longmapsto} d}$$

Figure 2.4: The resolution calculus of Statix, as presented by Antwerpen et al. [3]

2.4.2 Description of Name Resolution

The second part of a Statix specification is a specification of name resolution rules. It introduces the labels that can occur on edges in the scope graph, the different types of declarations that we want to store in the scope graph and the paths that queries for these declarations should follow. We give an example of name resolution rules in Listing 2.2. On line 3 and 4, the edge labels are defined. We define two different edge labels, one for parent edges and one for import edges. These are the labels that can occur in the scope graph. Then, on lines 5-8, we define how the resolution of classes works. Whenever we perform a query for a class, the query is allowed to follow any number of P edges, but no other edges. On line 6, the ordering is defined to prefer declarations, indicated with $ over following P edges. Similarly, the resolution for methods is defined on line 7 and 8. The regular expression for paths is now P*I* :, with a preference of declarations over following P and I edges, and following P edges before I edges. On lines 10-12, the namespaces are defined. Here we define the declarations that can appear in the scope graph. In the example, the declarations that appear are classes and methods, both identified with a single string which corresponds to the name of the class or method. Finally, lines 14 and 15 define the type relation. It specifies that occurrences in the scope graph can have an associated type.

2.4.3 Rules from AST Elements to Constraints

The final part of a Statix specification contains the rules. Rules are recursive predicates that match on fragments of the AST of the object language program. These rules contain the constraints that need to be solved whenever the rule is called. An example of a Statix rule is

```
signature
  sorts id = string                   // <id> is a string

  sorts Program constructors          // Program ::=
    Program : list(Class) -> Program  //             <Class*>

  sorts Class constructors            // Class ::=
    Class : id * list(Field) -> Class //           class <id> { <Field*> }

  sorts Field constructors            // Field ::=
    Field : id * Type -> Field        //           <Type> <id>;

  sorts Type constructors             // Type ::=
    IntT   : Type                     //          int
    BoolT  : Type                     //        | bool
    ClassT : id -> Type               //        | class(<id>)
```

Listing 2.1: Example of a Statix signature for a simple language with classes and fields. In comments, a corresponding syntax definition is given

```
signature
  name-resolution
    labels P    // parent
           I    // import
    resolve Class filter P*
                  min $ < P
    resolve Field filter P*I*
                  min $ < P, $ < I, P < I

  namespaces
    Class : string
    Field : string

  relations
    type : occurrence -> Type
```

Listing 2.2: Example of name-resolution, namespaces and relations definitions in Statix

```
1  rules
2    programOk : Program
3    programOk(Program(classes)) :-
4      {programScope}
5      new programScope,
6      classesOk(programScope, classes).
7
8    classesOk maps classOk(*, list(*))
9    classOk : scope * Class
10   classOk(programScope, Class(name, fields)) :-
11     programScope -> Class{name@name} with type ClassT(name),
12     //...
```

Listing 2.3: Example of a Statix rule

given in Listing 2.3. Note that we use the signature definition from our previous example. On the second line, the type of the rule is defined. The rule matches on sorts with the type `Program`. On the third line, an instantiation of the rule is shown. The rule pattern matches on the `Program` constructor, which has a list of classes. The list of classes is unified with the variable $classes$ through pattern matching. In essence, this introduce a equality constraint of the form $classes ==$ ..., where the dots are filled with the actual list from the AST. On the fourth line, the variables of the rule are declared. We declare a single variable called $programScope$. On line 5, the constraints of the rule start. First, a new scope is created and unified with the $programScope$ variable. Then, on line 6, the rule $classesOk$ is called with the $programScope$ and the $classes$. The solver will expand this rule call into the corresponding set of constraints based on the description of the rule and the pattern matching. Line 8 through 11 show the handling of classes, specifying how a class declaration can be added to the scope graph and how its type can be associated.

2.5 Constraint Solving

As mentioned, Statix contains a constraint solver. A constraint solver attempts to satisfy a set of constraints. For example, for the constraints $a = b + c$ and $a = 5$, the solution $b = 4 \wedge c = 1$ would satisfy the constraints. Statix takes as input a Statix specification and the encoding of an input program e. The input program, represented as one or multiple ASTs, is converted to a set of constraints with the specification. The Statix solver then attempts to satisfy these constraints by constructing a scope graph $\mathcal{G}$ and a substitution φ in the form of a unifier. It either finds a solution $\mathcal{G}\varphi$ which satisfies the constraint set, or, if the constraint set is not satisfiable, produces an error.

2.5.1 Constraints

We distinguish four types of constraints: (1) constraints which assert the structure of the scope graph, (2) constraints which retrieve information from the scope graph, (3) constraints which assert values of variables and (4) all the remaining constraints. The first category consists of constraints which 'build' the scope graph, i.e. the creation of scopes, edges and data edges. We call these constraints *scope graph building constraints*. The second category consists of the *query constraint*, which expresses name resolution in the scope graph. The third category consists of the *equality constraint*, which attempts to find a substitution to make the equality true, and the *existential constraint*, which introduces a fresh variable. The final category consists of constraints that do not do anything previously mentioned. It contains constraints for true, false, conjunction, guarded disjunction and predicates.

2.5.2 Unification

Statix uses unification to assign values to variables. Unification is the process of trying to find a substitution such that two terms are equal [5]. Unification can occur whenever Statix reduces an equality constraint. Equality constraints have the form $t = t'$, where t and t' are terms.

Equality constraints are satisfied by Statix by unifying variables with terms. A unifier is a substitution of variables in such a way that no equalities are false. For example, for the constraints $a = \mathit{IntT}()$ and $a = b$, a possible unifier would be $\{a = \mathit{IntT}(), b = \mathit{IntT}()\}$. We write a unifier as a mapping from variables to values. In the given example, both the variable a and the variable b are mapped to the term $\mathit{IntT}()$. If we substitute the variables from the unifier in the constraints, we should end up with simple equalities, e.g. $\mathit{IntT}() = \mathit{IntT}()$.

Let us look at a more complex example. For the constraints $a = \mathit{FunT}(b, c)$ and $b = c$, a possible unifier would be $\{a = \mathit{FunT}(\mathit{IntT}(), \mathit{IntT}()), b = \mathit{IntT}(), c = \mathit{IntT}()\}$. However, this unifier assumes additional information on the free variables b and c. The function represented by the first equality constraint is a function from any type to any type, but we instantiate it as a function from integers to integers. To avoid this scenario, Statix always creates a most general unifier. A most general unifier is a unifier that makes the least amount of assumptions, i.e. the least amount of substitutions. A most general unifier for our example would be $\{a = \mathit{FunT}(c, c), b = c\}$. All the constraints are still satisfied, but no unnecessary assumptions are made. We simply leave the value of c open.

2.5.3 Answer Stability

Since constraints are represented as a set, there is no explicit ordering to them. However, the satisfaction of some constraints depends on the order in which they are executed. Queries are the only constraint in Statix that can retrieve values from the scope graph. As such, the answer to the query depends on the current state of the scope graph. Our intention is that a query is only answered when its answer is stable. That is, if there are no other constraints which, when executed first, would affect the result of the query.

To address this, Rouvoet et al. [38] introduce the concept of *critical edges*. A critical edge is an extension to the scope graph that will be added by a constraint which is yet to be evaluated. Critical edges ensure a form of staging in Statix. The critical edges are the extensions to the scope graph that would invalidate the answer of a query. However, determining the absence of critical edges is a notion too strong for a solver to verify. As such, Statix uses a slightly weaker notion for critical edges, namely weakly critical edges. Weakly critical edges are extensions to the scope graph that *might* be added by the constraints that are yet to be solved. Before a query is executed, weakly critical edges are determined from the constraints that are left. If there exists a weakly critical edge that would be traversed by the query, the query cannot be answered yet.

In addition, Statix asserts a static property on the specification to limit the places where scope extensions can occur, called *permission to extend*. A predicate with *permission to extend* a scope s is allowed to introduce new edges and data edges in s. A predicate has permission to extend a scope s if it was created in that predicate, or if it was passed to it by a predicate with permission to extend s. This notion prevents predicates from extending scopes that have been obtained through queries. This simplifies the determining of weakly critical edges significantly.

2.5.4 Valid Solutions

With constraint solving problems, there are often multiple possible solutions. With Statix however, we want to find a specific solution. We are not interested in a solution which adds arbitrary scopes and edges to the scope graph to satisfy the constraints, since that no longer corresponds to the input program. In addition, Statix should make as few assumptions as possible on the instatiations of variables, to avoid 'inventing' program sections or types that are not specified. To achieve this, the solutions $(\mathcal{G}, \varphi)$ provided by Statix adhere to the following three properties.

1. The fully substituted scope graph $\mathcal{G}\varphi$ satisfies the input constraint set
2. The unifier φ is most general
3. The scope graph $\mathcal{G}$ contains no scopes, edges or data edges that are not asserted by a constraint in the constraint set.

In the case that a constraint set is unsatisfiable, Statix reports an error. In the case that Statix cannot find a valid solution that satisfies the constraint set (but cannot determine that no such solution exists), it reports that it is stuck.

2.5.5 Formal Specification

The formal syntax for Statix is given in Figure 2.5, as presented by Antwerpen et al. [3].

$$
\begin{array}{l}
\textbf{Signature} \\
\qquad\qquad \text{function symbols} \quad f, g \in \mathcal{F} \quad \text{with arity}(f) \in \mathbb{N} \\
\qquad\qquad \text{predicates symbols} \quad c, d \in C \quad \text{with arity}(c) \in \mathbb{N} \\
\textbf{Definitions}
\end{array}
$$

$$
\begin{array}{rrcll}
\text{term variables} & x & \in & \mathcal{V} & := \text{some countable set} \\
\text{terms} & t, u & \in & \mathcal{T} & ::= x \mid f(\bar{t}) \mid s \mid p \mid l \mid [] \mid [t|t] \mid (t, t) \\
\text{guards} & G & \in & \mathit{Guards} & ::= \top \mid G \wedge G \mid t = t \mid \exists x.G \\
\text{constraints} & C & \in & \mathcal{C} & ::= \top \mid \bot \mid t = t \mid C \wedge C \mid c(\bar{t}) \mid \exists x.C \mid G \mathbin{?} C : C \mid \nabla t \mid t \xrightarrow{l} t \\
 & & & & \quad \mid \; t \overset{r}{\longrightarrow\!\blacksquare} t \mid \text{query } (c(\bar{t}), c(\bar{t}), c(\bar{t}), c(\bar{t})) \text{ in } t \overset{r}{\longmapsto} t \mid t \backsim^{\mathsf{P}} \mathit{re} \\
\text{predicates} & \mathit{pred} & \in & \mathit{Preds} & ::= c(\bar{y}) \leftarrow C \\
\text{program} & P & \in & \mathit{Progs} & ::= \text{let } \mathit{pred}^{*} \text{ in } C
\end{array}
$$

Figure 2.5: The syntax of Statix, as presented in Antwerpen et al. [3]

2.6 Architecture of Statix

The main part of Statix is the Statix solver. It is implemented as step reductions of individual constraints, which progress the state. The solver has a `constraint store`, a `completeness` and a `state`. It has a single `solve` function which takes as input a set of constraints and outputs a solver result (solution or failure). First the solver adds all constraints to the constraint store. Then it keeps requesting constraints from the store, reducing them one by one, until the store reports that there are no constraints left. Finally, the solver creates a *solver result* and returns it. Note that in contrast to what is specified in the formal specification of Statix, the implementation does continue reducing remaining constraints whenever a constraint is found to be unsatisfiable. This is required to give meaningful feedback to the user whenever parts of the program contain errors.

Constraint Each constraint has a solve function which contains the implementation of solving that constraint. This is equivalent to a single step in the state of the solver. The solve function can return in one of three ways. First, it can solve correctly. The constraint returns the new state and, optionally, a set of new constraints that it reduces to. For example, a predicate constraint is expanded into the constraints of said predicate. Second, a constraint can be delayed on a critical edge or on the instantiation of a variable. For example, an edge cannot be asserted until the scopes it connects are created. In the case of a delay, the constraint will not be considered again until the delay is resolved, which is handled by the constraint store. Third, a constraint can fail, in which case it reports a failure. Failed constraints are effectively removed from the constraint set, and an error is recorded at the location of the corresponding or closest AST element.

Constraint Store The constraint store keeps track of the constraints that still need to be solved and the constraints that have been delayed. Whenever a delayed constraint can be reactivated, the store is informed and adds the constraint back to the set of constraints to solve. The solver requests constraints from the store one by one, adds new constraints to the store and reports delayed constraints to the store.

Completeness The completeness keeps track of critical edges, based on the constraints that are yet to be solved. Whenever a query constraint is being solved, it checks the completeness to see if there are critical edges along its paths. If there is a critical edge along its path, the constraint is delayed on that edge. Whenever the critical edge gets resolved, the query constraint is reactivated and can be attempted again. The solver keeps the completeness up to date by informing it whenever constraints are solved and added.

State The solver progresses the state step by step by reducing constraints. The state contains the unifier and the scope graph. Each constraint that is reduced influences either the unifier, the scope graph or neither.

Solver Result The solver result is either a Success or a Fail. It contains the state, all the constraints that failed and all the constraints that are stuck. A success indicates that the state has a scope graph and unifier which satisfy the input constraints. A fail indicates that the solver was not able to satisfy the constraint set, but still includes a state with a solution which satisfies the input constraint set excluding failed and delayed constraints.

Chapter 3

Background and Related Work

We look at literature to find existing techniques used by type checkers, compilers and build systems to improve analysis time. For each of the techniques we find, we explain if and how they could be applied to Statix.

One of the main topics we find in literature is incremental analysis/compilation. Incremental analysis is based on the idea of determining the differences of the current version of the code base compared to the previous version of the code base. These differences can be described in the form of, for example, files that have changed or the specific lines in the files that have changed. From the analysis result of the previous version of the code base, we try to infer what the analysis result of the current version of the code would be based on the changes that were made. The idea is that the impact of the change on the rest of the program can be estimated, from which the correct analysis result can be determined without redoing everything from scratch.There are different granularities of incremental analysis, ranging from redoing all dependencies of a file when it changes, to precisely determining which amount of work needs to be redone as a result of a particular change.

Definition 3.0.1. ***Incremental analysis****: the analysis of a program which potentially uses information from a previous analysis to determine its result. An incremental analysis might skip reanalyzing certain modules if their reanalysis is not necessary to compute the new result*

Definition 3.0.2. ***Clean run****: a single analysis of a project which analyzes all modules in the project without using information from previous analyses.*

Definition 3.0.3. ***Incremental run****: a single analysis of a project which has access to information from previous analyses, possibly allowing incremental computation of the new analysis result.*

3.1 Attribute Grammars

Attribute grammars are a technique for attributing syntax trees with attributes. These attributes can carry different values, and can be used to do different forms static analyses. Attribute grammars have been studied extensively, and there are different solutions for incremental recomputation of attributes when changes are made to the tree. In addition, attribute grammars have been extended in many different ways to support all kinds of static semantics such as type checking and name binding. In the following sections, we discuss the different kinds of attribute grammars in more detail, focussing on the techniques for incremental reanalysis of attributes. Finally, we evaluate if we can apply these incremental techniques as well.

Canonical Attribute Grammars Attribute grammars were introduced by Knuth [30]. Attribute grammars define equations to associate the values of *attributes* with abstract syntax tree nodes. The attribute values of each node are computed by combining the values of the parent and/or the children of the node. The basic form as introduced by Knuth was later renamed to canonical attribute grammars. Canonical attribute grammars are described with context free grammars. These context free grammars are extended with attributes for all nonterminals and semantic rules for productions. Attributes either transmit information upward or downward in the AST. An example of an attribute grammar is given in Figure 3.1. This attribute grammar describes the computation of the value of a binary number represented as a tree. The attributes are shown in Figure 3.2. Both figures are due to [30].

$$\begin{array}{lll}
B \to 0 & v(B) = 0 & \\
B \to 1 & v(B) = 1 & \\
L \to B & v(L) = v(B), & l(L) = 1 \\
L_1 \to L_2 B & v(L_1) = 2v(L_2) + v(B), & l(L_1) = l(L_2) + 1 \\
N \to L & v(N) = v(L) & \\
N \to L_1 \cdot L_2 & v(N) = v(L_1) + v(L_2)/2^{l(L_2)} &
\end{array}$$

Figure 3.1: An example of an attribute grammar for computing the value of a binary number represented as a tree. From Knuth [30], Figure 1.3.

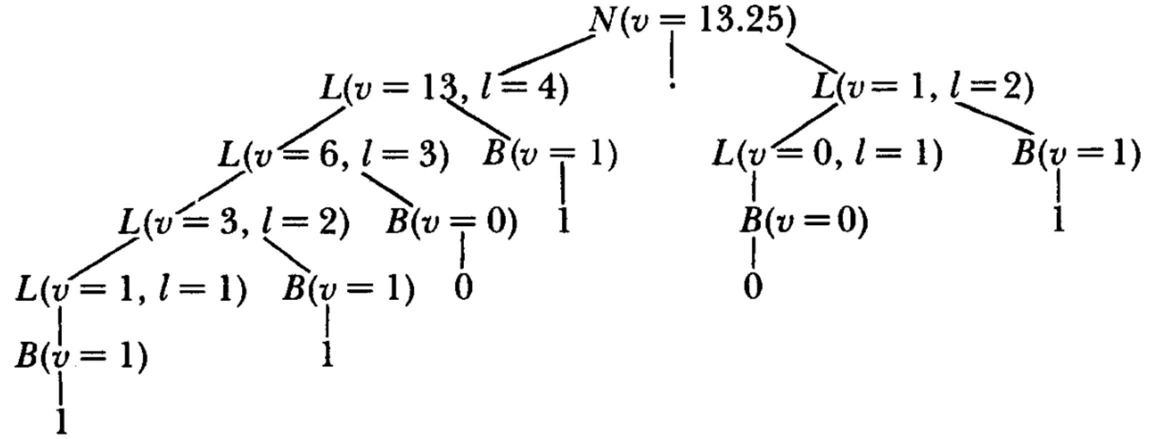


Figure 3.2: A tree representing a binary value, attributed with the values as determined by the attribute grammar in Figure 3.1. From Knuth [30], Figure 1.4.

Through the semantic rules, it is possible to describe analyses like type analysis. However, they are less suited to describing problems with non-local dependencies such as name analysis [22]. This is because the use site and the declaration site can be arbitrarily far away from each other in the tree. To describe name binding with canonical attribute grammars, it would be necessary to propagate information between all the nodes between use and declaration sites [22].

There are many different approaches to develop incremental analyzers for canonical attribute grammars. It is even possible to generate incremental static-semantic analyzers *automatically* from these grammars [37, 36, 23, 31]. These approaches mainly focus on being able to identify which attributes are derived from which attributes. Changes can then be propagated along the AST until all nodes have been updated. These techniques all use the fact that dependencies follow from the tree structure of the AST, and that an evaluation order can be statically computed.

Door Attribute Grammars Hedin [21] presents *Door Attribute Grammars*, a technique which can be used for incremental semantic analysis in languages with objects and references. They extend standard attribute grammars by allowing objects and references as attributions of a syntax tree. With this technique, incremental updates to the syntax tree can be processed efficiently. However, incremental attribute evaluators cannot be generated automatically, as different languages require different handling [21].

An interesting notion Hedin [21] use, is a distinction between *access dependencies* and *actual dependencies*. An *access dependency* is created whenever the computation of an attribute accesses the value of another attribute. However, these access dependencies usually give rise to a very pessimistic recomputation strategy. Instead, they determine actual dependencies. These are the dependencies that would actually be affected by a particular change. This is useful whenever a larger amount of information has to be requested when searching for one particular value. We can use a similar distinction for dependencies in Statix, to make them more precise.

Reference Attribute Grammars Reference attribute grammars (RAGs) are a simplification of Door Attribute Grammars. They extend canonical attribute grammars with reference attributes. A reference attribute references a node in the AST, i.e. a block or the body of a class declaration. The referenced node acts as an environment, in which declarations can be looked up. In addition, they define parameterized attributes, which are functions which take a name as a parameter and return a reference to the appropriate declaration. These parameterized attributes allow defining the name resolution rules in a programmatic way [22]. The JastAdd attribute grammar system [11, 12] uses these reference attribute grammars to generate compilers from declarative specifications. The approach is similar to the approach taken by Statix, with both approaches effectively constructing a graph to represent scoping and name resolution [3]. Because of these similarities, we also run into similar problems for incrementality, as we will explain in the next paragraph.

Söderberg and Hedin [39] present a technique for incremental evaluation of reference attribute grammars. The dynamic dependencies introduced by RAGs cannot be accounted for by the incremental attribution algorithms for attribute grammars. Earlier incremental algorithms for AGs work under the assumption that all attributes are known before-hand and that dependencies between them can be (mostly) statically determined [39]. However, in the case of RAGs, neither holds: the set of evaluated attributes depends on the tree and the dependencies are dynamic. As such, Söderberg and Hedin [39] construct a dependency graph during evaluation and restore consistency after edits using these dependencies. They transitively clear all the values that are dependent on the changed AST nodes. All the attribute values that remain are now consistent. Then, the new values can be computed by evaluating their corresponding equations.

They note that this does also clear values even if the values of changed attributes would not change. However, this is required to maintain consistency in the case of circular dependencies. Whenever there is a circular dependency, they cannot determine if the value of an attribute is affected by a change using outdated information. This is because the value that is determined might depend on another value which also depends on the original value in some way. Instead, all the values need to be recomputed from scratch. A similar problem applies to Statix, which we will discuss below.

3.1.1 Applicability to Statix

Statix, just like RAGs, has both dynamic dependencies and does not know the set of attributes to evaluate before hand. The ideas of the techniques used for RAGs can also be applied to Statix, if dependencies can be tracked well enough. To apply this to Statix, we would need to track the flow of information, which occurs both through the scope graph in the form as queries, as through the unifier in the form of changed values. In Section 5.3.2 we show that changes to the unifier can be represented as changes to the scope graph instead, removing the need to track dependencies through the unifier.

In addition, we also use a notion similar to *actual dependencies* to improve incremental performance. Whenever a query is looking for a specific name, i.e. `Class x`, we store that information in the dependencies created by it. Those dependencies are then only considered for reanalysis if that particular name is added or removed. This is explained in more detail in Section 6.2.

In Statix, the structure of the scope graph does not necessarily follow the AST. This is intentional to allow for virtual scopes, scopes which are not represented by any representative in the program. Examples of this are sequenced statements and selective imports. Sequenced statements cannot refer to all the declarations in their surrounding block, but only those declarations that appear before the statement in which the reference appears. Likewise, selective imports represent some subsection of the scope being imported rather than the entire scope [38]. However, this freedom does mean that we can not map changes in the AST to changes in the scope graph in a straightforward way. A scope can be created freely by typing rules, not necessarily following from any AST element. In fact, the interface of a module can depend directly on the interface of another module, and does not necessarily even have to use its AST to determine its interface. Because of these problems, we decided not to use an approach where we map changes in the AST to changes in the scope graph directly.

Circular Dependencies Whenever there are circular dependencies between modules, one would assume we cannot safely use information from old analyses. This is because the information that we base our value on might have an effect on the interface of the other module. However, in Statix, a scenario where two values are circularly dependent on each other always results in the solver getting stuck, because it is not able to find an appropriate ordering. If an ordering is found, then the circular dependency between the modules is only circular on a module level, not on a constraint level. So either, the original solver cannot solve such a scenario, or the circular dependency is not truly circular, and values from each scope graph can safely be used.

3.2 Separate Compliation

Separate compilation is the technique of typechecking and compiling separate program fragments. These program fragments cannot be compiled in isolation, but can be compiled in an environment with type information about the missing fragments [7]. We investigate the field of separate compilation since compilers deal with type checking and name binding problems just like Statix. Separate compilation and related techniques allow for incremental solutions, where only parts of the program need to be recompiled, achieving faster compilation times. We will conclude by discussing the applicability of these techniques to Statix.

The notion of separate compilation was made precise by Cardelli [7]. Separate compilation of a program fragment a can be seen as determining the type of the fragment in the context of some typing environment. The typing environment needs to contain the specification of sufficient type information of all the variables mentioned in a to be able to determine its type. Because the actual values of the items in the typing environment are not available, compilation is incomplete, but can still be carried out separately, i.e. without the missing values [7].

A *linker* then takes these separately compiled units and combines them into a larger unit, not necessarily complete. An incomplete linking produces a *library*, while a complete linking produces a program. Libraries can be linked again into larger libraries or into complete programs, completing the process Cardelli [7].

Modularization The process of separate compilation and linking requires that the program is split up into smaller fragments. Modularization is the concept of breaking a program into such fragments, which are then called *modules*. Modularization became necessary for compilation as the size of programs grew. With the increasing size of system libraries, these libraries had to be compiled separately from the user programs using them. As such, libraries acquired interfaces that minimized compilation dependencies, allowing them to be linked together later or even be linked dynamically at runtime Cardelli [7].

Besides their aforementioned use in allowing separate compilations for libraries, modularization also has great advantages in terms of large-grain program structuring [35]. All of these advantages combined gave rise to many modularization concepts in programming languages, such as classes in Java and C++, packages in Ada, and structures and functors in ML, in turn allowing for some form of separate or incremental compilation [7, 32].

Languages like Pascal, Ada, Modula, C and C++ distinguish between *interface modules* and *implementation modules*. This distinction allows compilers to compile implementation modules separately from other implementation modules, in the context of all the interfaces they import. Implementation modules and interface modules do not have to be in separate files, as long as the interface can be determined from the implementation in some way [4]. ‘

C The C programming language was created with separate compilation as a key feature. As mentioned, the increasing size of programs and libraries necessitated the need for better modularization and separated linking. Functions are the unit of modularization for C. C allows for separate compilation of source files. These separately compiled source files can then be combined together with previously compiled libraries of functions. A linker combines these separately compiled functions by either creating jumps (in assembly) between functions, or by directly inlining functions at the call sites [27].

Separate compilation of C is arguably the simplest. Each file can contain includes of some header files, which specify functions and structs that are in its interface. The compiler first checks if all called functions are made available through some header file and that they typecheck. It then compiles each function with what are effectively annotations at the places where the function calls another function. A linker can then combine these separately compiled functions by either creating jumps between functions, or by directly inlining functions at the call sites.

Ada Following the example of C, other languages also wanted to enjoy the benefits of separate compilation. Some of the earliest separate compilers were created for the Ada programming language [10, 6, 26]. They are mainly based on giving a description of an elementary program library. The system works by identifying the dependencies of a compilation unit and recompiling a unit whenever its dependencies have changed. These solutions mainly focus on how to identify and limit the dependencies that are created, to avoid unnecessary recompilation. A main challenge to limit dependencies in separate compilation in Ada is the ability to export imported declarations. Consider the example in pseudo-code in Listing 3.1.

While unit B depends on A, the removal of x in A will not change the interface of B. However, this change to A might affect C, so C needs to be invalidated. These dependencies need to be noted in order to achieve sound incremental compilation.

```
unit A:
    export x

unit B:
    import A
    export A as C

unit D:
    import B
    use B.C.x
```

Listing 3.1: An example of exporting imports. The dependency of D on A is hidden by the reexport of A under a different name by B

Standard ML Appel and MacQueen [4] introduce a system for incremental recompilation for Standard ML. Incremental recompilation is a restricted form of separate compilation. Whenever a changed module is recompiled and its export interface does not change, dependent modules are not recompiled. The absence of circular dependencies in Standard ML makes this approach sound. The reason they cannot achieve full separate compilation is because Standard ML has a very powerful module system which allows for inter-implementation dependencies, i.e. a dependency on a specific implementation rather than an interface. This means that for Standard ML, it is not always possible to compile an implementation with only the interfaces of imported modules, and inspection of the implementation modules might be necessary to even determine the interface in the first place.

Haskell Haskell has many of the same challenges to separate and incremental compilation as Standard ML due to the support of similar functionalities like functors. A key difference however is that in Haskell, one cannot specify interfaces, which means that modules always depend on implementations and have to be processed in dependency order. There have been implementations to extend Haskell with interfaces to allow for better incremental compilation [28], but these require either a separate build system or restrict the language.

Fourtounis and Papaspyrou [15] suggest a different approach of using defunctionalization, which transforms higher-order programs to first-order programs. The key technique to their approach is moving part of the code generation to link time. By storing additional information in the separately generated code, the code required for closures constructors and dispatching functions can be generated at link time.

In addition, incremental compilation is further obstructed by the fact that the most used Haskell compiler, GHC, is non-deterministic. This means that even with the same input, it can produce different outputs [17]. This is quite problematic to incremental compilation, as even when nothing changes, the compiled file can change, triggering recompilation of dependencies as well. There are efforts to identify the sources of non-determinism and make them deterministic.

Java The (Oracle) Java compiler supports separate compilation, i.e. compilation of compilation units in the presence of the interfaces of all used modules. However, there is also work to extend this to true independent compilation of files, where the linker then checks the integrity of the binaries to be combined. Modular linking in object-oriented languages is problematic because of some complex languages features regarding classes and objects. The separation between components depends on what can be hidden, or abstracted, between components. In Java, inheritance makes this abstraction more complicated. A subclass can introduce a method that would be fine in isolation, but which is invalid if it conflicts with a method provided by a superclass. Other operations such as checking for cyclic inheritance also requires knowledge of the superclasses [33].

Ancona, Lagorio, and Zucca [1] give an approach for separate compilation in Java which addresses this problem. In their approach, each file can be analyzed in isolation. Each analyzed file supplies a set of type requirements, expressing the information that must be present for it to be correct. For example, a type requirement can be that a particular class X needs to be available, and must have a method with a particular signature. These type requirements can be checked later when classes are combined by a linker.

For Statix, each module effectively expresses requirements on other modules in the form of queries on the scope graph. If the result of a query causes a constraint to be unsatisfiable, that means that the requirement was not met. If we link the creation of dependencies to these requirements, then we could determine dependencies automatically.

3.2.1 Application to Statix

Defining Modules and their Interfaces Given the high availability of module concepts in programming languages, we can use these to define our modules in Statix. We add the ability to express modules in Statix specifications to allow language designers to map them to the module concepts of the language they are specifying. We will use these modules as a basis for our incrementality, as explained in Section 4.1.

Then, we need to define what the interface of our module is. In essence, the scope graph for a module has to contain *at least* all the declarations that a module exports and *possibly* also declarations which it does not. These unexported declarations however, must not be visible to other modules. As such, we can define the interface of a module to be the subgraph of its scope graph with only those scopes, edges and declarations that can be reached from the global scope (see Section 5.3.4). With this definition, Statix is very suitable for an approach similar to separate compilation.

Do note however that just like in ML, we cannot build the scope graphs of modules in complete isolation. Determining the interface of a module requires analysis of the complete module. This is because the declarations that a particular module exports in its interface can depend on the declarations exported by other modules. In Statix, it is perfectly possible to write a specification such that the scope graph of module A is empty if module B exports a declaration x in its scopes graph, and consists of a declaration y if module B does not export a declaration x.

However, Statix has very different specifics. In order to use what would be considered implementation details for ML in another module, Statix requires that this information is available in the interface. While this means that no information can be hidden, it also means that determining the interface is more difficult, requiring us to record such implementation details as well. In fact, there is no separation between implementation modules and interface modules in our model (besides the fact that we exclude parts of the scope graph that are unreachable anyways for efficiency reasons). It is their usage, and specifically how this is expressed in the Statix specification that determines the incrementality that we can achieve. Because these implementation details need to be present in the interface, we expect the scope graphs of these modules to be larger and expect that there are more dependencies. However, we also counter this problem with the fact that we can do very precise determine if a module is affected by changes.

Interfaces of ML files can only be determined by executing them, due to the interactions between higher order functions and run-time constructed data. This is similar to how the determination of an interface in Statix requires the satisfaction/execution of its corresponding constraints, which can depend on values and can create function closures in the form of scopes. In contrast to ML however, it is possible to describe some types of cyclic dependencies in Statix. This complicates our quest for incremental analysis, since there might be no safe order in which modules can be analyzed. Luckily, the cyclic dependencies that would actually matter for our incremental compilation cannot be solved by the Statix solver. In particular, the overapproximation of critical edges (Section 2.5.3) to achieve query answer stability causes the solver to get stuck in the scenarios where no static ordering can be determined. In those scenarios, our incremental solution either gives a possible answer which would satisfy the constraints (i.e. a 'more valid' answer), or diverges. This is explained in more detail in Section 6.1.

Exporting Imports We address the problem of exporting imported declarations through the way we construct dependencies. A dependency is recorded whenever a query in module a reaches a scope owned by a module b. Even if a scope of module b is exported by some other module c, if we reach b in our query we record a direct dependency of a on b. This means that no intermediate module can 'hide' the dependency through exports. In addition, if the intermediate module c does not request any information in the scope of b that it exports, it will not get a dependency on b merely for exporting the scope. This means that, in such a case, changes to the declarations in the exported scope will not cause reanalysis of c, but still trigger reanalysis of a when necessary.

The interfaces and dependencies that we determine for a module a record all the information that is used by a from any other module, even if this information is accessed transitively through other modules. The advantage is that we can easily handle scenarios of reexports and can restore consistency even without having to reanalyze the intermediary modules. Our dependencies are very precise, giving rise to very effective incrementality. The disadvantage is that the set of dependencies that we store is quite large. Luckily, the cost of tracking is low, since the implementation of queries in Statix already visits exclusively those scopes on which we can depend.

3.3 Incremental Build Systems and Incremental Pipelines

We also evaluate the techniques used by build systems to achieve incrementality. While there are more build systems available, many of the techniques they use are already covered under separate compilation. Instead, we focus on some build systems with slightly other incremental approaches.

Simple Build Systems Make, MSBuild and Ant are tools for developing build systems based on declarative rules. While these tools do support incrementality, this is limited to static dependencies between files, which also have to be specified up front in the build rules. This means that dependencies often need to be overapproximated to retain soundness. In addition, it must be possible to separately compile files, using only the specification of interfaces, or dependencies need to be overapproximated [13].

Dependencies in Statix are always dynamic, since they are not specified in the specification. We do not want the user to have to define dependencies themselves, since this would be a major limitation to our system.

Build Systems with better Granularity Erdweg, Lichter, and Weiel [13] define *pluto*, a sound incremental build system. Builders in *pluto* notify the build system of all files they dynamically require and provide, as well as other builders whose results are required. To support dynamic dependencies, they interleave dependency analysis and builder execution and enforce invariants on the dependency graph through a dynamic analysis. The enforced invariants are statements like builder A must trigger builder B before it may read files generated by B. Pluto makes dependencies more fine-grained by allowing builders to specify which parts of a file they depend on in a programmatic way, instead of relying on simple dependencies of the form builder A depends on file F.

We can use a similar notion of making our dependencies more fine grained by including the information that we actually depend on, rather than just recording dependencies of the form module A depends on module B. In fact, these dependencies are already stated in the form of queries to other modules. It would also be possible to add a way for Statix specifications to explicitly state whenever dependencies are introduced. However, such a technique would be error-prone and dependencies can easily be missed. As such, we do not apply such a technique to Statix.

3.4 Incremental Type Checkers from the Ground Up

We also look into two techniques which achieve incremental type checking from their design. For each technique, we discuss if it can be applied to Statix.

Co-contextual Type Checking Erdweg et al. [14] presents a method for constructing so-called co-contextual type systems. Their typing rules are essentially functions from expressions to a type, the typing constraints and context requirements. The constraints and requirements are merged with a bottom-up approach. The fact that no context coordination has to be done makes dependency tracking much simpler and enables incremental type checking. Because typing rules introduce context requirements instead of depending on any context, they are very suited for incrementality. They can be reanalyzed without any context and as such, changes can easily be propagated from the changed expression up to the root.

In Statix, a bottom-up approach such as the one taken by Erdweg et al. [14] might be possible. However, the incrementality for the co-contextual model relies on being able to solve constraints locally, which is not possible for queries. In addition, Statix predicates are much more free in that they do not have to correspond to AST elements at all, which means that determining the items affected by a particular change is much more complex. We cannot apply these techniques without making significant changes to the design of Statix.

Task Engine Wachsmuth et al. [43] describe a task engine for incremental name and type analysis. Their analysis proceeds in two phases, a collection phase and an evaluation phase. The collection phase is incremental per file while the evaluation phase is incremental per task. In the collection phase an index is built which presents the declarations added by a partition and the resolutions that the partition does. Tasks are created for resolutions, which are executed in the next phase. Dependencies are tracked based on an index from the resolutions that are requested and the items that are supplied to the index. Whenever changes are made, the changes to the index are determined in the form of a delta. Based on the delta and the dependencies the tasks that are be affected are identified and are recomputed.

Statix uses a scope graph, for which the delta of the changes cannot be computed as easily, since it requires a full reanalysis of the changed unit to construct the changed scope graph. However, this reanalysis already requires using possibly outdated information from other files. In addition, queries in Statix are much more expressive and powerful than the resolutions in their paper, which gives rise to a much larger sets of dependencies that would need to be recomputed. Finally, mapping predicates or constraints to tasks would require a completely different architecture for Statix, which we want to avoid.

3.5 Conclusion

Research Question 1: What are existing techniques to improve analysis time as found in the literature?

From the type checkers, compilers, IDEs and build systems, we see that the main focus is on incrementality. Our answer to "**Research Question 1: What are existing techniques to improve analysis time as found in the literature?**" is as follows:

- Separate compilation is a technique of compiling program fragments separate from each other, using only limited information about other available program fragments [7]. It has been applied to compilers for many different languages to achieve incremental compilation [10, 6, 26, 4, 28]. We found that the techniques used for separate compilation can be applied to Statix. The separate program fragments of the source language can be mapped to modules in Statix. We can then analyze these modules separately from each other in the context of the interfaces of other modules, with the scope graph as a representation of the interface of a module.

- Incremental rebuilding of attribute grammar trees is another example where incrementality is used. Attribute grammars associate attributes to nodes in the abstract syntax tree of a program, allowing the definition of some static analyses. Whenever parts of the tree change, it is possible to invalidate only some of the attributes and recompute them, without having to recompute them for the entire tree [37, 36, 23, 31]. However, extending this to Statix would be difficult, since the ordering of constraints is complicated and not as clearly defined as for attributes.

- Hedin et al. [22, 39, 21] extend attribute grammars with support for objects and references. They effectively build a graph on top of the syntax tree, describing the scoping and name resolution. These reference attribute grammars require a different incremental approach, because dependencies are dynamic and are not known before hand. They present a technique for recording these dynamic dependencies automatically as their graph is built. The graph they build on the syntax tree bears a lot of similarities to the scope graphs used by Statix. Their way of tracking dynamic dependencies can be used in Statix as well, given that queries are similar to their name resolution rules.

- There are type checkers which take a completely different approach, gaining incrementality directly from their design [14]. These techniques cannot be applied to Statix without significantly changing the architecture, which we want to avoid.

Chapter 4

Modules and Separation

This chapter describes how the concept of modularization can be applied to Statix. This modularization forms the basis of our model for incremental analysis. First, Section 4.1 explains how we create modules for Statix. Next, Section 4.2 introduces collections of modules in the form of libraries. Then, Section 4.3 describes how we split the solving process in Statix over multiple, separate modules. Finally, Section 4.4 formally shows how Statix-core can be extended with the semantics of modules and provides a proof that these changes retain the soundness of Statix-core.

4.1 Splitting a Project into Modules

The first step towards achieving separate compilation-like incrementality is to define the modules of our system. In this section, we give a definition of modules and their interfaces (Section 4.1.1). We extend the syntax of Statix with module boundaries to allow the creation of so-called *child modules* (Section 4.1.2) and provide a formal specification of modules and their scope graphs (Section 4.1.3).

4.1.1 What should be a module?

The first question that need to be answered is what a module in Statix should be. Each programming language has different language constructs to represent modules. For example, Java has classes and C has functions. A module represents some part of the program, that is in some way logically separate from other parts of the program. Some examples of modules are namespaces/packages, classes, structs, and methods/functions. Files can also be modules, if they are treated separately by the programming language (or by the IDE). The given list of examples is by no means exhaustive. Depending on the language, there might be modules which are even smaller.

The idea is to use the already existing modules from languages and map them to modules in Statix. That is, the module fragments of the source language of a Statix specification are mapped to modules in Statix. These modules will become the granularity of the incrementality of our model for incremental analysis for Statix.

Generally speaking, modules in Statix should map to all the modules of a language. However, there is also some overhead in the bookkeeping required for the separation into modules. Thus, there is a trade-off between more, smaller modules with better incremental granularity and the additional cost for maintaining precise dependencies between them. Since this is highly dependent on the source language,

The Module Tree Modules are laid out as a regular tree. At the root, there is the project module. Each program fragment which contributes to the project will have an associated

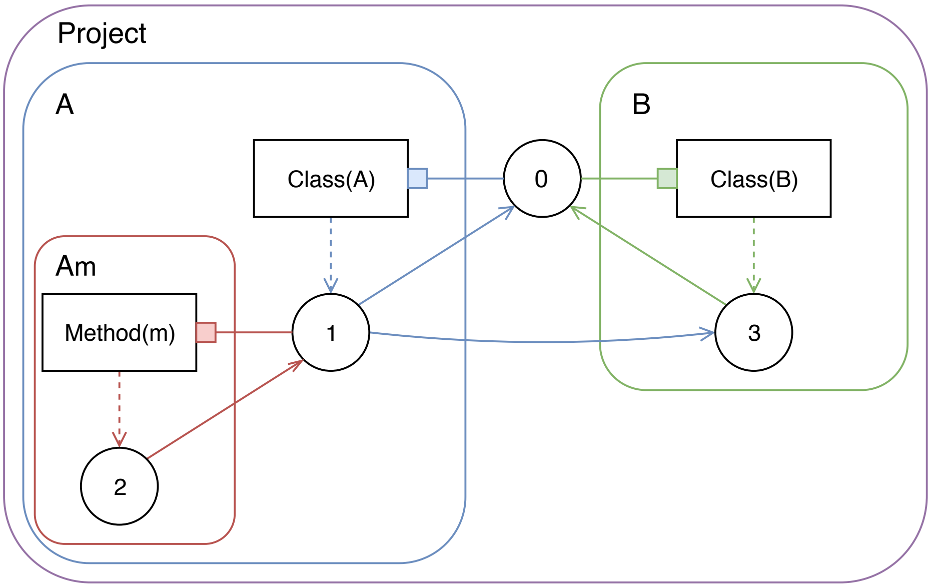


Figure 4.1: An example of a modularized scope graph. The colored boxes indicate the boundaries of the individual module scope graphs. Edges are colored to the module they belong to.

module that is a child of the project module. For example, in a language where files are the units of separate compilation, each file would get a module in the tree. If, in turn, these files are made up of other program constructs that are also modules, they are a child of their enclosing file module.

Children of a module are special in that they can view parts of the interface of a module which are not visible from completely unrelated modules. Modules can specify information which is visible only to them and their children to share information with children.

Definition 4.1.1. ***Child module***: *a module m is a* child module *of module m' if it was created by m'.*

The Interface of a Module Statix represents the scoping and interface of a program with a scope graph. As such, it would be only logical for a module to also have an interface represented by a scope graph. Effectively, we want to split the scope graph into multiple parts where each part represents the contributions of a particular module.

However, this split is not very clear cut. A module can contribute edges in the scope graph to scopes that are not in the scope graph of the module. This occurs for example for top level declarations. Top level declarations such as packages are defined in the global scope, which is owned by the project. However, the contribution of a package to the global scope is caused by a particular module, not by the project module itself.

Instead, it must be possible for module scope graphs to contain edges to scopes that are not their own. This gives us the following definition for a module scope graph.

Definition 4.1.2. ***Module scope graph***: *A subsection of a scope graph containing all the contributions made by a particular module. A module scope graph can contain outgoing edges to and incoming data edges from scopes which are not part of its graph.*

We will denote the scope graphs of individual modules by drawing boxes around them. Outgoing edges to scopes outside of the module scope graph are part of the module, while incoming edges are not. An example of a modularized scope graph is shown in Figure 4.1.

```
rules
  classOk : scope * Class
  modbound classOk(s, Class(name, content)) :-
    ...
```

Listing 4.1: Example of a simple module boundary in a Statix specification. Whenever the classOk rule is called, a new module will be created as a child of the calling module.

```
rules
  classOk : scope * Class
  modbound classOk(s, Class(name, content)) | $[class [name]] :-
    ...
```

Listing 4.2: Example of a module boundary with an associated name in a Statix specification. Whenever the classOk rule is called, a module with the name `class [name]` is created, where name is replaced with the name of the class.

4.1.2 Expressing Modules in Statix

In a Statix specification, rules are the predicates that match on parts of the AST. Since our modules correspond to program fragments in the source language, these fragments can already be matched by rules. It makes sense to be able to specify special rules which create a module. We achieve this by defining module boundaries. A module boundary is a specially annotated rule, which when called, will create a new module as a child of the calling module. We specify these module boundaries by introducing a new keyword `modbound`. Any rule can be prefixed with `modbound` to turn that rule into a module boundary. An example of a module boundary is shown in Listing 4.1.

We add some restrictions to the module boundaries to avoid the possibility of complex parent-child dependencies caused by variables. We say that all the values passed to a module boundary must be ground, i.e. must contain no free variables. This restriction simplifies the detection of dependencies later (see Section 4.3.4). If parent modules need to share information with children that is possibly not ground, they can still do so via the scope graph instead. As such, we deem this a sensible limitation.

Unique and Consistent Names Because we want to be able to refer to modules and store them in a logical way, it is important that we can refer to modules uniquely. To ensure that modules have unique names, we use a notation similar to file systems: paths. The fully qualified identifier of a module consists of the fully qualified identifier of its parent module followed by the name of the module. This moves the uniqueness problem to the children of a module. Given one parent, all its children need to have a unique name.

In many cases, the programming language we are specifying already requires unique names to some degree. For example, classes in java must be unique in their package. However, some cases are not necessarily unique by name, but rather by some other property. An example of this is method overloading in java. Methods in a class can have the same name, as long as they have different signatures.

Preferably, we would like modules to be referable by other meta-languages in Spoofax. To accommodate for this, we let the user specify the name of a module. We extend the syntax for module boundaries with a format string which defines the name of the module. The format string can contain variables from the signature of the module boundary, as long as they can have some textual representation (i.e. not scopes). An example is shown in Listing 4.2.

We require that the name of a module is known whenever it is created, i.e. before any of its constraints are solved. As such, only variables from the signature of the rule can be used in its name. This means that all information that is required for the name of a module must be provided by its parent or must be available in the AST directly. Care should be taken to ensure that module names are unique. An attempt to create multiple modules with the same name and parent will result in an error.

For proper incrementality it is important that the name is consistent. Consistent means that if the way one refers to the module from other parts of the program does not change, its name should also not change. Consider an extreme example where the name of our module is the textual representation of its corresponding AST. If any element in its AST changes, it will get a different name. It will then be considered to be a completely different module. Such a change would mean the deletion of one module and the creation of a new module, which can cause unnecessary reanalysis of its dependencies.

In some cases, the user will not be able to specify a unique name for a module boundary. In these cases, it would be useful to be able to assign a unique module name automatically. However, given that the order in which constraints are evaluated is not fixed, a simple counter for assigning names does not suffice. While we have not encountered scenarios which would require this, such scenarios might exist. Our current implementation does not yet support this scenario, but we do have a solution for it. Unique names can be assigned to modules based on a stack trace of rules. The trace of rules which were called and reached the module creation constraint is both unique and consistent, and can thus be used to assign names.

Module Extensions Some programming languages allow extensions to a module at a different location. Essentially, the module in the programming language is defined by multiple, separate locations in the program, i.e. in multiple files. We might want to specify that these different locations together create a single module. However, this makes it more difficult to determine what needs to be reanalyzed incrementally. In many cases, these different locations actually do not contribute to the same module, but rather one location extends the module specified in another location. Because it is already possible to encode such an extension model in Statix, we do not provide additional support through our module system.

4.1.3 Formally Specifying Modules

Figure 4.2 formally specifies modules and their corresponding scope graphs. We define functions to describe the module tree, i.e `parent` and `children`, as well as transitive variants of these functions, `parents` and `descendants`.

The `owner` function asserts the ownership of items in the scope graph and is overloaded to scopes, edges and data edges. A module scope graph is defined through the scopes, edges and data edges contributed (i.e. owned) by a module.

Finally, we also define an extends relation which relates scopes to all the modules which have permission to extend them.

Ownership of Edges Edges can only be added by the owner of the scope and its children, as described in Lemma 4.1.3. This follows from the fact that in order to add an edge to a scope, that scope needs to be passed directly to the rule. Since rules can be module boundaries, it is possible to pass a scope to a child module. Together, this means that only scopes passed by the parent of a module are extensible by that module.

Syntax

$$
\begin{array}{rlll}
\text{modules} & m \in M & := & \text{some finite set} \\
\text{module graph} & \mathcal{G}_m & ::= & \langle S_m, E_m, D_m \rangle \\
\text{module scopes} & s \in S_m & ::= & \{ s \mid s \in \text{scopes}(\mathcal{G}), owner(s) = m \} \\
\text{module edges} & e \in E_m & ::= & \{ e \mid e \in \text{edges}(\mathcal{G}), owner(e) = m \} \\
\text{module data} & d \in D_m & ::= & \{ d \mid d \in \text{data}(\mathcal{G}), owner(d) = m \}
\end{array}
$$

Formulae

$$
\begin{array}{rlll}
\text{parent} & M \nrightarrow M & & \\
\text{children} & M \rightarrow \overline{M} & & \\
\text{parents(m)} & M \rightarrow \overline{M} & ::= & \{\{m'\} \cup parents(m') \mid m' = parent(m)\} \\
\text{descendants(m)} & M \rightarrow \overline{M} & ::= & \{\{m'\} \cup descendants(m') \mid m' \in children(m)\} \\
\text{owner} & S \rightarrow M & & \\
\text{owner} & E \rightarrow M & & \\
\text{owner} & D \rightarrow M & & \\
\text{extends} & S \rightarrow \overline{M} & &
\end{array}
$$

Figure 4.2: Formal definition of modules and their scope graphs.

Lemma 4.1.3. *Each edge that appears in the scope graph of a module m must have as source a scope owned by m or owned by a parent of m.*

$$
\begin{aligned}
\forall (s \longrightarrow s') \in E_m \, . \, owner(s) = m \vee owner(s) \in parents(m) \\
\forall (s \mathrel{-\!\!\blacksquare} d) \in D_m \, . \, owner(s) = m \vee owner(s) \in parents(m)
\end{aligned}
$$

Reconstructing the Scope Graph We can obtain the complete scope graph by taking the disjoint union of all module scope graphs. In other words, there is no overlap between the scope graphs of modules. This is shown in Lemma 4.1.4.

Lemma 4.1.4. *The disjunct union of all module scope graphs yields the original scope graph.*

$$
\mathcal{G} = \biguplus_{m \in M} \mathcal{G}_m
$$

Proof. For each scope, edge and data edge in $\mathcal{G}_m$, it is defined that its owner is m. As such, there cannot be overlap between the different scope graphs of modules. Given that each scope, edge and data edge is given exactly one owning module and that M contains all modules, the union of the scope graphs of these modules must be our original scope graph, i.e. $\bigcup_{m \in M} \mathcal{G}_m$. □

The fact that this holds means that no information is lost in splitting the scope graph into modules. We will need this property later for our proofs that modules retain the soundness of Statix (see Section 4.4.4).

While reconstructed scope graphs are useful for formal specifications, they are not efficient for implementation. Section 4.3.3 specifies how our implementation provides the view of a complete scope graph without actually reconstructing it, which is necessary to be able to query information from other modules.

4.2 Libraries

Many programming languages have some support for librarys. A library is some package of implementations that can be used in a program, which has a well-defined interface for these implementations. Our incremental model adds support for libraries in the form of collections of modules as described below.

Definition 4.2.1. ***Library****: a collection of implementations which has a well-defined interface. A library consists of multiple smaller modules and presents some interface to use the implementations of those modules in other libraries or programs.*

4.2.1 Libraries in Statix

The main reason why library support is important for Statix is because many languages include a so-called standard library. A standard library is available to be used by any program in the language. It often includes common algorithms, collections, I/O operations and other utility functions. Analyzing programs in a particular language without the standard library is difficult. It either needs to be encoded in the Statix specification directly or, if it is written in the analyzed language itself, needs to be included and analyzed with the program every time. A much better approach would be if a library could be included in any project as is, without requiring either work from the language designer or additional analysis time.

Adding Libraries to Statix Our model adds support for libraries to Statix. A library in Statix is a collection of modules with a single scope graph. Adding a library to a project adds the modules of the library as so-called library modules. Library modules are treated in a special way in the solving process. They will never be reanalyzed and cannot contain errors. Essentially, they are fixed, unmodifiable modules which can be queried but cannot be changed.

Libraries can be created by exporting an analyzed project as a library. Libraries can only be created if the analysis of the library completes without errors. This is due to the special treatment of libraries, which would otherwise cause these errors to be lost.

Definition 4.2.2. ***Library module****: a special module that can be queried, but which will not participate in the solving process otherwise. Library modules have no constraints and their scope graphs and unifiers cannot be modified.*

4.2.2 Libraries in the Module Tree

Each project in Statix has a root node in its scope graph, called the global scope. If a library is added as is, it would introduce multiple root nodes to the scope graph. Recall that the semantics of querying a scope are to request the edges in said scope from its owner. The owner of the scope propagates this request to all children that have permission to extend said scope and collects the edges. Since libraries are not descendant from the same root, it would not be possible for a query to reach them.

Instead, the global scopes of the library and the project need to be represented as one scope. We considered two different approaches to this problem. The first approach is to change the semantics of queries to request edges from all global scopes. The second approach is to place the root modules of the libraries as children of the project module. The first approach has the advantage that the library and the project stay separated completely, but requires extensive modifications to the semantics. The second approach has the advantage that the semantics do not need to be changed, but does introduce libraries as if they are children of the project. We chose the second approach because of its simplicity and because a library is more a child of the project than a separate module.

With this definition, no additional changes need to be made. The modules of the library are simply added into the scope graph as is. Only the solving process needs to treat these library modules as different, to prevent them from being considered for reanalysis.

4.3 Splitting the Solving Process

This section focuses on the challenges created by splitting the solving process of Statix over multiple separate solvers. We change Statix from a monolithic global approach to a cooperative solving process. In addition, we reuse as much of the existing implementation as possible. First, Section 4.3.1 explains why the solving process needs to be split in the first place. Then, Section 4.3.2 shows how the split solving process is coordinated. Section 4.3.3 follows by describing how split scope graphs are implemented. Then, Section 4.3.4 explains how unification can be split across modules and the issues that arise from this separation. Finally, Section 4.3.5 addresses the changes that were made to the completeness mechanism of Statix to retain query answer stability.

4.3.1 Why Split the Solving Process?

The Statix Solver is a step based constraint solver. There is one global set of active constraints, and these constraints are solved one by one (one step at a time). Each constraint that is solved can introduce new constraints that will be added to the set. While it would be possible to keep this architecture even for modular solving, we want to achieve a stricter separation of modules and also accommodate for parallel solving of modules. This requires that there is a separate solver for each module, which also allows us to completely separate the interactions between modules in a logical way. This makes it easier to determine what is contributed by which module and to observe the interactions that occur between modules.

To achieve this, we need to modify all the monolithic components that comprise the Statix solver and turn them into components which collaborate to get to a solution. We mainly apply a technique of *redirecting*, i.e. a request for information which resides in some module m is redirected to that module regardless of where the request initially arrived.

The Module Solver Each module gets a solver that is responsible for satisfying the constraints of this module. This solver is for a large part identical to the original monolithic step solver that Statix had. The main difference is that it has an owning module, and all the different components it interacts with are modified versions which allow for cooperative solving.

4.3.2 Coordinating Multiple Solvers

With multiple solvers, the solving process needs to be coordinated. For this purpose, we introduce the *Solver Coordinator*. The solver coordinator is responsible for creating solvers for each module and to cooperatively reach a solution. It ensures that each solver that is able to make progress is allocated some time to actually make that progress. Whenever there are no solvers that can make progress, the coordinator gathers the results of each solver and combines them into a result. The coordinator can determine that solvers have finished successfully, that solving failed or that some solvers have become stuck.

We provide two implementations of the solver coordinator: a sequential, single-threaded version and a concurrent version. The sequential coordinator is described below and the concurrent coordinator is described in more detail in Section 5.4.

The sequential coordinator solves modules in a greedily. It makes as much progress as possible in a single module before moving on to the next module. This is achieved by having the step solver of the module do as many constraint reduction steps as possible until it cannot progress its state any further. Whenever a solver is done, the coordinator collects and stores the result of the solver and removes it from the queue of solvers to make progress in. After attempting to make progress in all solvers that are left in the queue, the coordinator checks if any progress was made. If no progress was made, solving is complete and the coordinator combines the results. If progress was made, the coordinator starts a new round in which it goes over the different solvers again to check if they have any work to do. If the queue of solvers ever becomes empty, the coordinator concludes that solving has finished and combines the results.

4.3.3 Split Scope Graphs

As explained in Section 4.1.1, we split the scope graph across the different modules. We want each scope and edge in the scope graph to have a *single* owner, so there can be no question about who introduces what. In addition, we still need to be able to reconstruct the complete scope graph to be able to do name resolution.

We change the identity of scopes from a single name to a pair, where the first part describes the module which created the scope and the second part describes its name. As such, any request for the edges of a particular scope can be redirected to its owner. By redirecting requests regardless of where they arrive, we effectively provide an interface for the complete scope graph, without having to reconstruct it.

However, all the descendants of a module can also contribute to its scopes if they have permission to extend them. As such, edges for a particular scope must be collected from some of the descending modules as well. We define a depth first collection operation, where each child module that has permission to extend the scope in question is asked to add its corresponding edges to the answer.

By using an interface like this, no changes need to be made to the implementations of each individual constraint. From the view of the constraint, the entire scope graph is accessible, but behind the scenes the queries are delegated and distributed. This model is called a *distributed scope graph*. Instead of a single module determining the answer to a question, the answer is collected by distributing the question among different modules.

4.3.4 Splitting Unification

A large part of Statix is unification. Unification of two terms is the process of finding a substitution which makes those terms equal. Rather than substituting variables eagerly, a mechanism of unification is used to associate values with variables. When a variable is encountered, its corresponding value can be obtained from the unifier. A variable is considered free if there is no value associated with it (yet). If a term contains free variables, it is called *non-ground*. In a monolithic approach, unification is simple as it can occur at any place and the place where values should be requested is singular. However, because we split the unifier over different modules, this becomes more complex.

We apply the same concept to the unifier as we did to the scope graph. Just like scopes, we add the information of which module created a variable in their identity. Whenever a unifier gets a request for the value of a variable from another module, we simply redirect the request there. In contrast to the redirections in the scope graph, the unifier needs to be modified in additional ways.

Variables can occur in arbitrarily complex terms. In fact, the values associated with variables can again contain variables. As such, a single level of redirection does not suffice. Instead, we need to recursively redirect to the correct unifier if we want to fully instantiate a term or if we want to determine if it is ground.

Cross-module Unification In Statix, it is possible to add free variables to the scope graph. This can occur because variables need not be instantiated when they are added as data in the scope graph. Either the variable will be instantiated later, or possibly it will never get any value. While this is not a problem by itself, assigning values to variables created in other modules is. Let us show how assigning values to variables from other modules is problematic for assigning blame (which module is responsible) and consistency.

Consider the following scenario. There is a module m with a single constraint $x = Int()$ and a module m' with a constraint $x = Bool()$. If m is evaluated first, then $Int()$ will be associated with x. Now, when module m' is evaluated, it will fail because $Bool()$ and $Int()$ are not unifiable. However, if m' would be evaluated first, the opposite would happen and module m would fail for the same reason. In addition, the actual value assigned to x depends on the order in which modules are evaluated. So, we now have a scenario in which the type of something in the program is dependent on the order in which we evaluate our modules. In addition, the module(s) which fail also depend on the order.

Instead, we would like a consistent result (x is always $Int()$, always $Bool()$, or always stays unbound) and consistent failure. To address this, we restrict the unification of variables. We restrict unification in such a way that a module can only assign values to variables created by it. The intuition behind this idea is that the creator of a variable should have full ownership of it and should be able to decide what it substitutes the variable with. Any other module however, should be able to request the value, but should not be able to influence it. This way, we avoid the described inconsistencies and get the same outcome regardless of module evaluation order. In addition, it means we do not have to consider the unifier for dependencies, which greatly simplifies how we determine dependencies between modules.

However, we do lose a bit of expressiveness. This is in line with our idea for modularization, i.e. each module is *separate* in some way from other modules. Having variables in one module which are given values in another module goes against the idea of this separation, and could indicate that incorrect module boundaries are used. Be aware that it is still possible to, for example, create a declaration in one module and assign its type in another. One module can create the declaration in the scope graph, and the other can query for it and create a relation which relates the type to that declaration. As long as one module does not preemptively create a variable which should be filled by the other module, we do not run into cross-module unification.

Delaying Constraints on Variables from other Modules It is possible for a constraint to be delayed on non-ground variables. For example, if the scope in which a query is performed is a free variable, we cannot perform the query yet until the variable is instantiated. Similarly, some of the pattern matching in rules can only be done if the terms are instantiated.

Originally, these constraints were tracked by the solver and reactivated whenever the variable was instantiated. However, in our split approach it is possible for constraints in one module to be delayed on variables of another module. Informing all solvers of each variable that is instantiated would incur quite a bit of messaging. Instead, we use an approach based on the observer pattern [18].

Whenever a constraint is delayed on a variable, a registration is made with the unifier of the module owning the variable. Whenever the variable is instantiated, the corresponding unifier notifies us to reactivate the constraint. In a single threaded fashion, such a mechanism is quite simple. However, in a concurrent fashion, extra care needs to be taken that the variable is not instantiated between the moment where it is determined that a constraint is delayed and the moment where the registration is made. Otherwise, the event would be missed. This is explained in further detail in Section 5.4.

4.3.5 Tracking Completeness over Multiple Modules

To guarantee answer stability, Statix uses the notion of critical edges (see Section 2.5.3). The (in)completeness of edges is tracked by the `completeness` component of the solver. The completeness is essentially a counter. Whenever the solver is instantiated, it determines critical edges from its initial constraints and the specification. Each (scope, label) combination where an edge could be added by a constraint later is identified and tracked.

As constraints get simplified, the completeness is updated. The critical edges appearing in the original constraint are deducted from the completeness, and the critical edges appearing in the constraints it reduced to are added to the completeness. In other words, whenever an edge creation constraint $s \xrightarrow{l} s'$ is reduced, the counter for (s, l) is lowered by one. An edge is considered complete if its counter is zero. However, determining this accross multiple solvers is slightly more difficult.

First, note that each scope can only be extended by a limited set of modules, i.e. modules which were passed that scope directly from their parents. These are the modules with permission to extend a scope. As such, only the modules with permission to extend a scope can affect the completeness of said scope. Furthermore, modules too are created from a reduction of a constraint. Whenever a module boundary is executed, the original solver of the constraint reduces it to no new constraints for itself, but does instantiate a new solver with the new constraints. This instantiation of the new solver will update the completeness.

We can apply our mechanism of redirections again, where updates to the completeness as a result of constraint reductions are sent to the completeness of the module owning the scope.

Delaying Constraints on Critical Edges Just like with variables, it is possible for constraints to be delayed on critical edges. We can apply a similar observer mechanism to solve the problem of delaying on critical edges from other modules. Whenever the counter of a particular edge hits zero, we activate all the observers of that edge.

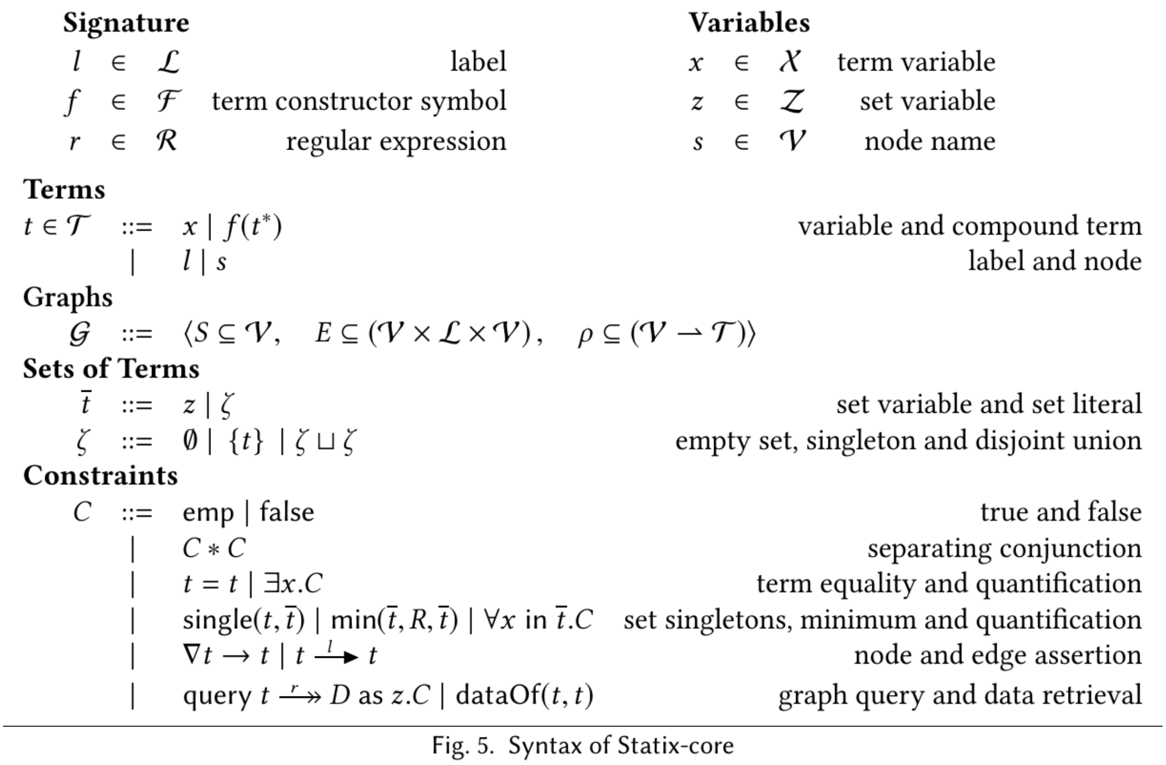
To prevent accidentally lowering the counter to zero and then increasing it again, we ensure that all increments of the counter happen before the decrements to the counter. For each reduction of a constraint c to a set of constraints C, we first add the critical edges of the constraints in C to the completeness, and then remove the critical edges of c from the completeness. In addition, we ensure that whenever a new module is created, it adds the critical edges from its initial constraints to the completeness, before the critical edges of the constraint reduction are removed by its parent. This way, we achieve atomic completeness tracking which also works out of the box with concurrency.

4.4 Formal Semantics of Modular Statix-core

In this section we provide the formal specification of an extension of Statix-core with modules. First, Section 4.4.1 provides the motivation for describing the semantics of Statix-core rather than Statix. Then, Section 4.4.2 specifies the syntax and updated declarative semantics in terms of constraint satisfiability. Section 4.4.3 follows by explaining the changes to the operational semantics that we make. Finally, Section 4.4.4 shows that modular Statix-core retains the soundness that Statix-core enjoys.

4.4.1 Motivation

Antwerpen et al. [3] presents an implementation that makes Statix rules executable. However, they do not prove that the operational semantics and name resolution algorithm of Statix are sound. This is mainly due to difficulty of characterizing the conditions under which it is safe to answer a query when guaranteeing query answer stability. Instead, Rouvoet et al.



Signature			**Variables**	
$l \in \mathcal{L}$	label		$x \in \mathcal{X}$	term variable
$f \in \mathcal{F}$	term constructor symbol		$z \in \mathcal{Z}$	set variable
$r \in \mathcal{R}$	regular expression		$s \in \mathcal{V}$	node name

Terms			
$t \in \mathcal{T}$	$::=$	$x \mid f(t^*)$	variable and compound term
	$\mid$	$l \mid s$	label and node
Graphs			
$\mathcal{G}$	$::=$	$\langle S \subseteq \mathcal{V},\quad E \subseteq (\mathcal{V} \times \mathcal{L} \times \mathcal{V}),\quad \rho \subseteq (\mathcal{V} \rightharpoonup \mathcal{T})\rangle$	
Sets of Terms			
$\bar{t}$	$::=$	$z \mid \zeta$	set variable and set literal
ζ	$::=$	$\emptyset \mid \{t\} \mid \zeta \sqcup \zeta$	empty set, singleton and disjoint union
Constraints			
C	$::=$	$\mathsf{emp} \mid \mathsf{false}$	true and false
	$\mid$	$C * C$	separating conjunction
	$\mid$	$t = t \mid \exists x.C$	term equality and quantification
	$\mid$	$\mathsf{single}(t, \bar{t}) \mid \mathsf{min}(\bar{t}, R, \bar{t}) \mid \forall x \text{ in } \bar{t}.C$	set singletons, minimum and quantification
	$\mid$	$\nabla t \rightarrow t \mid t \xrightarrow{l} t$	node and edge assertion
	$\mid$	$\mathsf{query}\ t \xrightarrow{r} D \text{ as } z.C \mid \mathsf{dataOf}(t, t)$	graph query and data retrieval

Fig. 5. Syntax of Statix-core

Figure 4.3: The syntax of Statix-core, as presented in figure 5 by Rouvoet et al. [38]

[38] introduce Statix-core which refines the concepts of Statix into a smaller core language which guarantees query stability. They also provide a proof that the operational semantics of Statix-core are correct with regards to the declarative semantics of Statix-core.

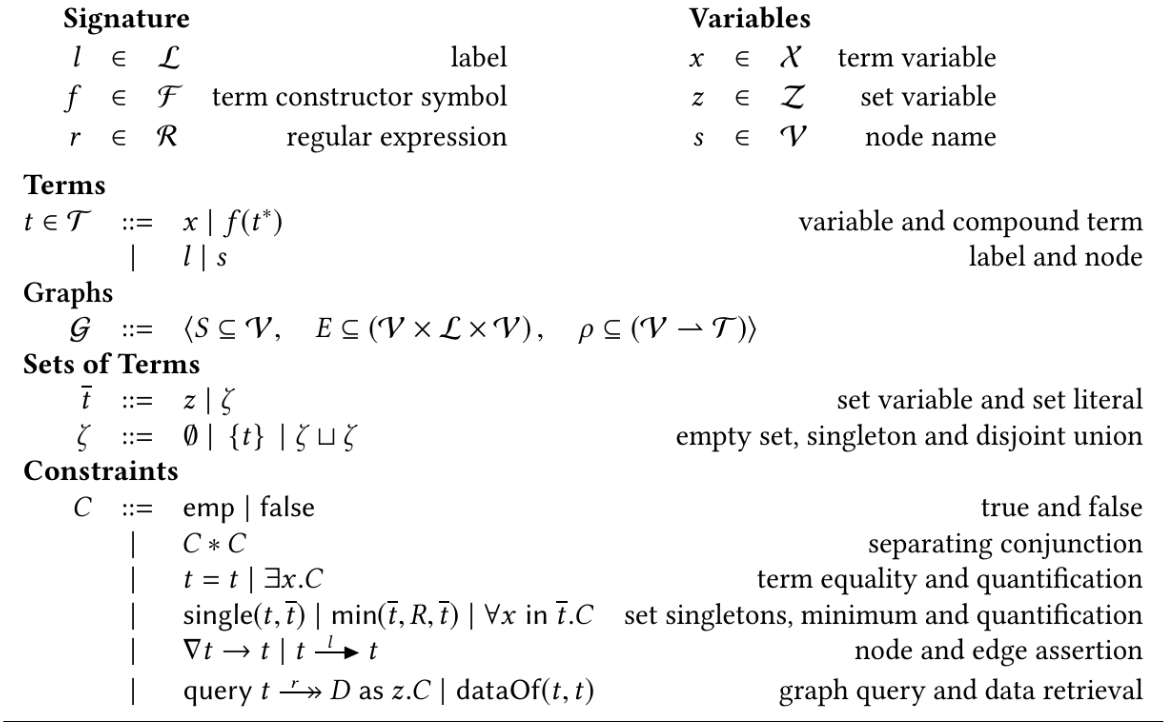
There are a few differences between Statix and Statix-core. First of all, queries in Statix-core do not have an order on paths. Second, queries bind their answer directly in the constraints they reduce to. Third, query results are represented as sets. In addition, they leave out predicates and guarded disjunction. However, they conjecture that their soundness and confluence results extend to full Statix as long as there are no overlapping guards.

We provide updated declarative and operational semantics for a modular version of Statix-core and provide proof sketches to show that it is still sound. We conjecture that this extends to the declarative semantics of modular full Statix as well.

4.4.2 Declarative Semantics

The syntax of Statix-core, as presented by Rouvoet et al. [38], is shown in Figure 4.3. We extend the syntax of Statix-core with a module creation constraint: mod C, where C is the initial constraint of the newly created child module.

Figure 4.4 shows the declarative semantics of Statix-core in terms of constraint satisfaction rules, as presented by Rouvoet et al. [38]. We do not need to redefine constraint satisfaction to include modules or modular graphs. Instead, only the operational semantics actually need to change to support modules. As such, we only need to make a small addition to the original declarative semantics of Statix-core. We add the constraint satisfiability rule for the module creation constraint in Figure 4.5. Because the declarative semantics do not consider modules, module creation constraints essentially become predicates, for which the unfolding is trivial.



Signature

l	$\in$	$\mathcal{L}$	label
f	$\in$	$\mathcal{F}$	term constructor symbol
r	$\in$	$\mathcal{R}$	regular expression

Variables

x	$\in$	$\mathcal{X}$	term variable
z	$\in$	$\mathcal{Z}$	set variable
s	$\in$	$\mathcal{V}$	node name

Terms

$t \in \mathcal{T} \;::=\; x \mid f(t^*)$ — variable and compound term

$\mid\; l \mid s$ — label and node

Graphs

$\mathcal{G} \;::=\; \langle S \subseteq \mathcal{V},\quad E \subseteq (\mathcal{V} \times \mathcal{L} \times \mathcal{V}),\quad \rho \subseteq (\mathcal{V} \rightharpoonup \mathcal{T})\rangle$

Sets of Terms

$\bar{t} \;::=\; z \mid \zeta$ — set variable and set literal

$\zeta \;::=\; \emptyset \mid \{t\} \mid \zeta \sqcup \zeta$ — empty set, singleton and disjoint union

Constraints

$C \;::=\; \mathsf{emp} \mid \mathsf{false}$ — true and false

$\mid\; C * C$ — separating conjunction

$\mid\; t = t \mid \exists x.C$ — term equality and quantification

$\mid\; \mathsf{single}(t, \bar{t}) \mid \mathsf{min}(\bar{t}, R, \bar{t}) \mid \forall x \text{ in } \bar{t}.C$ — set singletons, minimum and quantification

$\mid\; \nabla t \rightarrow t \mid t \xrightarrow{l} t$ — node and edge assertion

$\mid\; \mathsf{query}\ t \overset{r}{\twoheadrightarrow} D \text{ as } z.C \mid \mathsf{dataOf}(t, t)$ — graph query and data retrieval

Fig. 5. Syntax of Statix-core

Figure 4.4: The declarative semantics of Statix-core in terms of constraint satisfaction, as presented in figure 6 by Rouvoet et al. [38]

$$\textsc{Module-Fresh}\quad \frac{\mathcal{G} \vDash_\sigma C}{\mathcal{G} \vDash_\sigma \text{mod } C}$$

Figure 4.5: Constraint satisfiability for the module creation constraint

4.4.3 Operational Semantics

We present the operational semantics of modular Statix-core as small step reductions on state tuples. For completeness, we first provide all the rules for the original operational semantics of Statix-core in Figure 4.6. To avoid confusion between the original rules and the new rules, we will use the prefix `Op-M-*` to refer to the rules for modular Statix-core. Whenever a rule shown is not the final version of that rule, we suffix its name with a version number, i.e. `Op-Node-Fresh-1`.

$\kappa \to \kappa'$ State κ steps to κ'

Op-Emp

$$\langle \mathcal{G} \mid \mathsf{emp}; \overline{C} \rangle \to \langle \mathcal{G} \mid \overline{C} \rangle$$

Op-False

$$\langle \mathcal{G} \mid \mathsf{false}; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle$$

Op-Conj

$$\langle \mathcal{G} \mid (C_1 * C_2)\,; \overline{C} \rangle \to \langle \mathcal{G} \mid C_1; C_2; \overline{C} \rangle$$

Op-Eq-True

$$\frac{t_1\varphi = t_2\varphi \qquad \varphi \text{ is most general}}{\langle \mathcal{G} \mid (t_1 = t_2)\,; \overline{C} \rangle \to \langle \mathcal{G}\varphi \mid \overline{C}\varphi \rangle}$$

Op-Eq-False

$$\frac{\neg\exists\varphi. t_1\varphi = t_2\varphi}{\langle \mathcal{G} \mid (t_1 = t_2)\,; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

Op-Exists

$$\frac{y \text{ is fresh for } \mathcal{G} \text{ and } \overline{C}}{\langle \mathcal{G} \mid (\exists x.C)\,; \overline{C} \rangle \to \langle \mathcal{G} \mid C\,[y/x]\,; \overline{C} \rangle}$$

Op-Singleton-True

$$\langle \mathcal{G} \mid \mathsf{single}(t, \{t'\}); \overline{C} \rangle \to \langle \mathcal{G} \mid (t = t'); \overline{C} \rangle$$

Op-Singleton-False

$$\frac{\neg\exists t'. \bar{t} = \{t'\}}{\langle \mathcal{G} \mid \mathsf{single}(t, \bar{t}); \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

Op-Min

$$\frac{\zeta' = \mathsf{min}(\zeta, R)}{\langle \mathcal{G} \mid \mathsf{min}(\zeta, R, x); \overline{C} \rangle \to \langle \mathcal{G}\,[\zeta'/x] \mid \overline{C}\,[\zeta'/x] \rangle}$$

Op-Forall

$$\langle \mathcal{G} \mid (\forall x \text{ in } \zeta.C)\,; \overline{C} \rangle \to \langle \mathcal{G} \mid \{C\,[t/x] \mid t \in \zeta\} \cup \overline{C} \rangle$$

Op-Node-Fresh

$$\frac{s \notin S}{\langle \langle S, E, \rho \rangle \mid (\nabla x \to t); \overline{C} \rangle \to \langle \langle (s; S), E, \rho[s \to t]\,[s/x] \rangle \mid \overline{C}\,[s/x] \rangle}$$

Op-Node-Stale

$$\frac{t_2 \text{ is not a variable}}{\langle \mathcal{G} \mid (\nabla t_2 \to t_1)\,; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

Op-Data

$$\frac{\rho(s) = t_2}{\langle \mathcal{G} \mid \mathsf{dataOf}(s, t_1); \overline{C} \rangle \to \langle \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle}$$

Op-Edge

$$\langle\, \langle S, E, \rho \rangle \mid (s_1 \xrightarrow{l} s_2); \overline{C} \rangle \to \langle \langle S, (s_1, l, s_2)\,; E, \rho \rangle \mid \overline{C} \rangle$$

Op-Query-Guarded

$$\frac{\forall s_2, l. \Big(\mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2 \text{ and } \mathcal{L}(\delta_{wl} r) \neq \emptyset \text{ implies } (C; \overline{C}) \not\hookrightarrow (s_2, l)\Big)}{\langle \mathcal{G} \mid \mathsf{query}\ s_1 \xrightarrow{r}\!\!\!\!\rightarrow D \text{ as } z.C; \overline{C} \rangle \to \langle \mathcal{G} \mid C\Big[\mathsf{Ans}\Big(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\Big)/z\Big]\,; \overline{C} \rangle}$$

Figure 4.6: The original operational semantics of Statix-core as presented in Rouvoet et al. [38]

States To support the split scope graphs used by modules, some modifications to the operational semantics need to be made. Rouvoet et al. [38] give the operational semantics in small step semantics defined on state tuples $\langle \mathcal{G} \mid \overline{C} \rangle$, where $\mathcal{G}$ is the scope graph and $\overline{C}$ is a set of constraints that is repeatedly simplified. To stay close to the paper, we will do the same with a modified state.

Because we have split the scope graph into multiple modules, we also have to alter the state. Instead of one graph, we have partial graphs which together form the scope graph. In addition, modules form a tree hierarchy which needs to be available for query solving. We redefine the state as $\langle \overline{M} : m \rightharpoonup m' \mid \overline{\mathcal{G}} : m \rightharpoonup \mathcal{G} \mid \overline{(m, \overline{C})} \rangle$. $\overline{M}$ is a partial function from module identifiers to module identifiers which denotes the parent of a module. $\overline{\mathcal{G}}$ is a partial function from module identifiers to graphs and $\overline{(m, \overline{C})}$ is a set of tuples $(m, \overline{C})$ with a module identifier and the corresponding constraint set. We also define new accepting, rejecting and stuck states. The semantics accepts a final state $\langle \overline{M} \mid \overline{\mathcal{G}} \mid \overline{(m, \overline{C})} \rangle$ where $\forall (m, \overline{C}) \in \overline{(m, \overline{C})},\ \overline{C} = \varnothing$ and rejects a final state $\langle \overline{M} \mid \overline{\mathcal{G}} \mid \overline{(m, \overline{C})} \rangle$ where $\exists m \in dom(\overline{M}).\ (m, \{false\}) \in \overline{(m, \overline{C})}$. States in which no steps can be taken to reduce are said to be *stuck*.

To make the notation more readable, we define a rule which picks an arbitrary module and makes progress in that module, `Op-M-SingleModule`. We can then define rules using the simplified relation if these rules do not need information from other modules. This means that we can almost reuse most of the operational semantics from the original small step semantics given in Figure 7 in the paper.

Op-M-SingleModule

$$\frac{\overline{\mathcal{G}}(m) = \mathcal{G}_m \qquad \langle m \mid \mathcal{G}_m \mid \overline{C} \rangle \to \langle m \mid \mathcal{G}'_m \mid \overline{C}' \rangle}{\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, \overline{C}); \overline{(m, \overline{C})} \rangle \to \langle \overline{M} \mid \overline{\mathcal{G}'}[m \to \mathcal{G}'_m] \mid (m, \overline{C}'); \overline{(m, \overline{C})} \rangle}$$

Figure 4.7: Small step semantics of performing a step in a single module.

Introducing Module Creation Constraints First of all, we introduce a new constraint mod C. This constraint creates a child module with the constraint C as initial constraint. The small step semantics for the module constraint are shown in Figure 4.8. This is the only rule that is able to create other modules.

Op-M-Module-Fresh

$$\frac{\mathrm{fv}(C) = \varnothing \qquad m' \text{ is fresh}}{\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, (\mathrm{mod}\ C); \overline{C}); \overline{(m, \overline{C})} \rangle \to \langle \overline{M}[m' \to m] \mid \overline{\mathcal{G}}[m' \to \epsilon] \mid (m, \overline{C}); (m', \{C\}); \overline{(m, \overline{C})} \rangle}$$

Figure 4.8: Small step semantics of the module creation constraint

Redefining Scopes To access scope graphs of other modules, we need to know the module owning a particular scope to redirect queries and dataOf requests to. To accommodate for this, we add the owner of a scope to its identity. This means that a scope becomes a tuple (m, i) with the module owning the scope m and the original identity of the scope i. This affects the rule for creating a scope slightly. In addition, the rule for requesting data of a scope needs to be modified to request correct scope graph. These rules are shown in Figure 4.9.

Non-local Substitutions Most rules can only have a local effect. However, variables can 'leak' from one module into another module if they are exposed as data. This data can then

$$\textsc{Op-M-Node-Fresh-1}\quad \frac{s = (m,i) \qquad s \notin S}{\langle m \mid \langle S, E, \rho\rangle \mid (\nabla x \rightarrow t); \overline{C}\rangle \rightarrow \langle m \mid \langle s; S, E, \rho[s \rightarrow t][s/x]\rangle \mid \overline{C}[s/x]\rangle}$$

$$\textsc{Op-M-Data}\quad \frac{s = (m', i) \qquad \overline{\mathcal{G}}(m') = \langle S_{m'}, E_{m'}, \rho_{m'}\rangle \qquad \rho_{m'}(s) = t_2}{\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, \text{dataOf}(s, t_1); \overline{C}); \overline{(m, \overline{C})}\rangle \rightarrow \langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, (t_1 = t_2); \overline{C}); \overline{(m, \overline{C})}\rangle}$$

Figure 4.9: The updated rules for creating scopes and retrieving data when using the new scope notation

be queried by other modules and can end up in other constraints and other scope graphs. As such, substitution of variables has to happen globally rather than locally. To this end, we define the rules `Op-M-Eq-True` and `Op-M-Node-Fresh` to apply their substitutions globally in Figure 4.10. We do not need to consider the rule for min, as set variables cannot be added to the scope graph since they are not terms. As such, min only needs local substitution.

$$\textsc{Op-M-Eq-True}\quad \frac{t_1\varphi = t_2\varphi \qquad \varphi \text{ is most general}}{\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, (t_1 = t_2); \overline{C}); \overline{(m, \overline{C})}\rangle \rightarrow \langle \overline{M} \mid \overline{\mathcal{G}}\varphi \mid (m, \overline{C}\varphi); \overline{(m, \overline{C})\varphi}\rangle}$$

$$\textsc{Op-M-Node-Fresh}\quad \frac{\overline{\mathcal{G}}(m) = \langle S, E, \rho\rangle \qquad s = (m,i) \qquad s \notin S}{\begin{array}{c}\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, (\nabla x \rightarrow t); \overline{C}); \overline{(m, \overline{C})}\rangle \rightarrow \\ \langle m \mid \overline{\mathcal{G}}[m \rightarrow \langle s; S, E, \rho[s \rightarrow t]\rangle][s/x] \mid (m, \overline{C}[s/x]); \overline{(m, \overline{C})}[s/x]\rangle\end{array}}$$

Figure 4.10: Operational semantics for rules that need to apply global substitutions, i.e. `Op-M-Eq-True` and `Op-M-Node-Fresh`.

Queries The rule for queries is arguably the most interesting one. Queries need to be allowed to find paths throughout the scope graph. However, queries are guarded by $(C; \overline{C}) \nrightarrow (s_2, l)$. Because the constraint set is split over multiple modules, this does no longer guarantee query answer stability. Instead, this fact needs to be asserted for all the constraint sets to adhere to the original rule. Note that unlike full Statix, Statix-core does not have the notion of *permission to extend*, since it does not support predicates.

We first define a helper function `join` which joins all the partial graphs together into one scope graph as follows.

$$\textsc{Join}\quad \frac{\langle S, E, \rho\rangle = \biguplus_{m \in dom(\overline{\mathcal{G}})} \overline{\mathcal{G}}(m) \qquad \mathcal{G} = \langle S, E, \rho\rangle[\forall (m', s) \in S.\, s/(m', s)]}{join(\overline{\mathcal{G}}) = \mathcal{G}}$$

Considering all of this, we end up with the rule as presented in Figure 4.11.

Simple Rules All the remaining rules can use the simplified relation for stepping in a single module. These simple rules are mostly equivalent to their definition in Statix-core (besides the altered state tuples). These rules are presented in Figure 4.12.

Op-M-Query-Guarded

$$\frac{\mathcal{G} = \text{join}(\overline{\mathcal{G}}) \qquad \overline{C'} = \{\overline{C_{m'}} \mid (m', \overline{C_{m'}}) \in \overline{(m, \overline{C})} \wedge m' \neq m\} \\ \forall s_2, l.\, (\mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2 \text{ and } \mathcal{L}(\delta_{wl} r) \neq \varnothing \text{ implies } (C; \overline{C} \cup \overline{C'}) \not\hookrightarrow (s_2, l))}{\langle \overline{M} \mid \overline{\mathcal{G}} \mid (m, (\mathsf{query}\, s_1 \xrightarrow{r}\!\!\!\!\twoheadrightarrow D\, \mathsf{as}\, z.\, C; \overline{C}); \overline{(m, \overline{C})} \rangle \to \\ \langle m \mid \overline{\mathcal{G}} \mid (m, C[\text{Ans}(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\twoheadrightarrow D)/z]; \overline{C}); \overline{(m, \overline{C})} \rangle}$$

Figure 4.11: Operational semantics of queries for modular Statix-core

$$\textsc{Op-M-Emp}\quad \frac{}{\langle m \mid \mathcal{G} \mid emp; \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \overline{C} \rangle}$$

$$\textsc{Op-M-False}\quad \frac{}{\langle m \mid \mathcal{G} \mid \text{false}; \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \{\text{false}\} \rangle}$$

$$\textsc{Op-M-Conj}\quad \frac{}{\langle m \mid \mathcal{G} \mid (C_1 * C_2); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid C_1; C_2; \overline{C} \rangle}$$

$$\textsc{Op-M-Eq-False}\quad \frac{\neg \exists \varphi . t_1 \varphi = t_2 \varphi}{\langle m \mid \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \{false\} \rangle}$$

$$\textsc{Op-M-Exists}\quad \frac{y \text{ is fresh for } \mathcal{G} \text{ and } \overline{C}}{\langle m \mid \mathcal{G} \mid (\exists x . C); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid C[y/x]; \overline{C} \rangle}$$

$$\textsc{Op-M-Singleton-True}\quad \frac{}{\langle m \mid \mathcal{G} \mid \text{single}(t, \{t'\}); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid (t = t'); \overline{C} \rangle}$$

$$\textsc{Op-M-Singleton-False}\quad \frac{\neg \exists t' . \bar{t} = \{t'\}}{\langle m \mid \mathcal{G} \mid \text{single}(t, \{t'\}); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \{false\} \rangle}$$

$$\textsc{Op-M-Min}\quad \frac{\zeta' = \text{min}(\zeta, R)}{\langle m \mid \mathcal{G} \mid \text{min}(\zeta, R, z); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \overline{C}[\zeta'/z] \rangle}$$

$$\textsc{Op-M-Forall}\quad \frac{}{\langle m \mid \mathcal{G} \mid (\forall x \text{ in } \zeta . C); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \{C[t/x] \mid t \in \zeta\} \cup \overline{C} \rangle}$$

$$\textsc{Op-M-Node-Stale}\quad \frac{t_2 \text{ is not a variable}}{\langle m \mid \mathcal{G} \mid (\nabla t_2 \to t_1); \overline{C} \rangle \to \langle m \mid \mathcal{G} \mid \{false\} \rangle}$$

$$\textsc{Op-M-Edge}\quad \frac{}{\langle m \mid \langle S, E, \rho \rangle \mid (s_1 \xrightarrow{l} s_2); \overline{C} \rangle \to \langle m \mid \langle S, (s_1, l, s_2); E, \rho \rangle \mid \overline{C} \rangle}$$

Figure 4.12: The operational semantics of modular Statix-core that do not need information from other modules. These rules use the simplified modular state, but are otherwise equivalent to their original specification in Rouvoet et al. [38].

Maintaining Well-formedness In terms of well-formedness, we do not need to modify any of the original well-formedness rules. We only need to add a rule for our new module creation constraint, which is shown in Figure 4.13. Likewise, we only need to add one rule to the syntactical extends predicate, as shown in Figure 4.14. Both of these rules are quite straightforward, since the module creation constraint does not actually impose any interesting behavior.

$$\frac{\textsc{Wf-Module-Fresh}}{} \quad \frac{\triangle^{\downarrow}, \triangle^{\uparrow} \vdash C}{\triangle^{\downarrow}, \triangle^{\uparrow} \vdash \text{mod}\ C}$$

Figure 4.13: Well-formedness rule for the module creation constraint. A module creation constraint is well-formed if the constraint it reduces to is.

$$\frac{\textsc{Ext-Module-Fresh}}{} \quad \frac{C \hookrightarrow (s, l)}{(\text{mod}\ C) \hookrightarrow (s, l)}$$

Figure 4.14: Syntax directed edge support predicate for the module creation constraint. The critical edges of a module creation constraint are equal to the critical edges of the constraint it reduces to.

4.4.4 Proving Correctness

The intuition for our approach is that the changes that we make only move constraints from one set to multiple sets. The non-deterministic nature of the operational semantics does not change, since the module to make progress in is also non-deterministically selected. Constraints with global effect (i.e. substitution) are still applied to all constraints and queries still execute on the complete graph. Thus, constraints are only being divided over multiple (disjoint) sets while scopes and edges are divided over multiple (disjoint) graphs. Any state of modular Statix-core can be converted to a state for Statix-core by taking the union of all the module scope graphs and by taking the union of all constraint sets (removing module creation constraints) and by converting variables and scopes to their original, non-modular forms. As such, nothing in the behavior actually changes as a result of modules.

To describe that any state for modular Statix-core can be converted to a state for Statix-core, we define the helper function `original`. `original` constructs an original constraint set from modular constraint sets as shown in Figure 4.15.

State transitions of modular Statix-core can now be expressed in terms of state transitions in Statix-core, as shown in Theorem 4.4.1.

Original-False

$$\frac{\exists m.\,(m, \{\text{false}\}) \in \overline{(m, \overline{C})}}{\text{original}(\overline{(m, \overline{C})}) = \{\text{false}\}}$$

Original-True

$$\frac{\nexists m.\,(m, \{false\}) \in \overline{(m, \overline{C})} \qquad \overline{C} = \bigcup_{(m, \overline{C_m}) \in \overline{(m, \overline{C})}} \overline{C_m} \qquad \overline{C}' = \overline{C}[\forall (\text{mod } C) \in \overline{C}.\, C/(\text{mod } C)]}{\text{original}(\overline{(m, \overline{C})}) = \overline{C}'}$$

Figure 4.15: Definition of the original helper function which transforms multiple modular constraint sets into the equivalent original constraint set.

Theorem 4.4.1. *Any state transition in modular Statix-core can be expressed in terms of state transitions in Statix-core.*

$$\left\langle \overline{M} \mid \overline{\mathcal{G}} \mid \overline{(m, \overline{C})} \right\rangle \to \left\langle \overline{M}' \mid \overline{\mathcal{G}}' \mid \overline{(m, \overline{C})}' \right\rangle$$

$$\textit{implies}$$

$$\left(\textit{there exists a step } \left\langle \textit{join}(\overline{\mathcal{G}}) \mid \textit{original}\left(\overline{(m, \overline{C})}\right) \right\rangle \to \left\langle \textit{join}(\overline{\mathcal{G}}') \mid \textit{original}\left(\overline{(m, \overline{C})}'\right) \right\rangle \right)$$
$$\vee \left(\left\langle \textit{join}(\overline{\mathcal{G}}) \mid \textit{original}\left(\overline{(m, \overline{C})}\right) \right\rangle = \left\langle \textit{join}(\overline{\mathcal{G}}') \mid \textit{original}\left(\overline{(m, \overline{C})}'\right) \right\rangle \right)$$

Proof Sketch. The proof is by induction on the possible reductions. The case where the original states are equal occurs only whenever a module creation constraint is reduced. The reduction for query constraints already operate on the joined scope graph and all the remaining constraints, which makes it quite trivial. For all the constraints that can be reduced with the `Op-M-SingleModule` relation (i.e. Figure 4.12), the equivalence follows trivially because the reductions are the same and the effects are local, except for `Op-M-False`. In Statix-core, all constraints are immediately reduced to false whenever false is encountered. However, in the modular version, other modules are immediately reduced to false in such a case. The equivalence follows from the definition of original, which reflects that a failure in any module reduces to a rejecting state. The rules for `Op-M-Eq-True` and `Op-M-Node-Fresh` apply substitutions on all scope graphs and all constraints, which is equivalent to the original reductions. □

As such, we can conclude that the operational semantics of modular Statix-core is sound with regards to the declarative semantics of Statix-core. Given that the module creation constraint functions as a simple predicate for constraint satisfaction, constraint well-formedness and the syntax directed edge support mechanism, it is trivial to extend the existing proofs by Rouvoet et al. [38] to support them. We conclude that modular Statix-core is sound with respect to its declarative semantics and enjoys the same soundness as Statix-core.

Chapter 5

Incremental Analysis

Building on the separation into modules introduced by Chapter 4, this chapter describes our model for incremental analysis for Statix. First, Section 5.1 introduces our model and necessary concepts for the rest of the chapter. Then, Section 5.2 describes how precise dependencies between modules can automatically be identified from queries. Section 5.3 follows by explaining how a change to the project can be mapped to all the modules which are affected by that change. Then, Section 5.4 describes how we add concurrency to our model in the form of parallel execution of the solvers of multiple modules. Finally, Section 5.5 concludes with the answer to our second research question.

5.1 Incremental model

In this section we introduce our model for incremental analysis. First we introduce the terminology for incremental analysis and clean and incremental runs. Then, we will discuss the multi-phase solving approach and how control over module interactions allows for different incremental strategies. Finally, we give an overview of the functionalities of our model, which will be

5.1.1 Incremental Analysis with Strategies

We define an incremental model for executing incremental analyses. The incremental model essentially describes the required coordination and organization of the analysis process, as well as the features that need to be available to solve incrementally. Our model is set up in a flexible way to allow for different incremental strategys. Incremental strategies are algorithms which use the features of our model to determine the result of an analysis from the results of the previous analysis and some changeset. These strategies can sometimes avoid reanalyzing modules whenever their reanalysis is not necessary.

Definition 5.1.1. ***Incremental analysis****: the analysis of a program which potentially uses information from a previous analysis to determine its result.*

Definition 5.1.2. ***Clean run****: an analysis run which analyzes all modules without using information from previous analyses.*

Definition 5.1.3. ***Incremental run****: an analysis run which has access to information from a previous analysis.*

Definition 5.1.4. ***Incremental strategy****: an algorithm for incremental analysis which takes a changeset and the previous analysis result as input and produces the result of the new analysis as output.*

Our model is strategy directed, that is, the strategy completely directs the solving process by using functionality of the model.

5.1.2 Multi-phase Solving

Our model is designed with multi-phase solving in mind. Multi-phase solving means the strategy can decide to separate the solving process into multiple different phases. For example, a strategy might reanalyze some set of modules in one phase and then conclude it needs another phase to reanalyze other modules. In addition, our model offers extensive control over the solving process, and allows for partial solving of individual modules.

Partial Solving and Control over Module Interactions Partial solving is the concept of solving only a subset of all constraints of one or more modules. Sometimes it makes sense for a strategy to limit the constraints that are solved. For example, a pessimistic strategy might not allow using potentially outdated information from other modules. Such a strategy can define that every time a module requests information from another module, that request must be denied. All denied requests cause the constraint making the request to get delayed. Delays

It is also possible to use partial solving to determine all the statically available information in scope graphs. If module interactions are completely disallowed, the only constraints which will be solved are those that are satisfiable in isolation. While our optimistic name-based strategy does not use this functionality, it can be used for other strategies as they see fit.

5.1.3 Features of our Model

In addition to the support for multi-phase solving, our model offers the following features.

- **Dependencies:** Automatic detection of dependencies between modules from queries. Detailed information is added to dependencies to allow for precise incremental strategies which only reanalyze a module if it could be affected by a change, rather than just because a module it depends on changed in some way (Section 5.2).

- **Scope Graph Comparisons:** Scope graphs can be compared with our model to obtain a so-called scope graph diff, consisting of added and removed edges. Strategies can extend these diffs with additional information as well (Section 5.3).

- **Affected Modules**: Affected modules can be determined by combining a diff with recorded dependencies. The detailed information in the dependencies is used for precise detection if a module is affected or not. For removals of edges, the affected modules are determined with maximum precision. Only if the removed edge is traversed, will the module be redone. For additions of edges, all additions which could affect a query cause reanalysis (Section 5.3).

5.2 Dependencies between Modules

For incremental analysis, it is important to know which modules can be affected by a change. We want to know the answer to the question *'If one module is changes, what are the modules that could be affected by this change?'* One part of the answer to this question is the dependencies that modules have on each other. The challenge is to determine dependencies between modules with as much details as possible, but in an efficient way. With more detailed dependencies, we can determine more precisely whenever modules are affected and whenever they are not, improving our incremental analysis.

First, Section 5.2.1 describes how modules can depend on each other and introduces the syntax we will use for dependencies. Then, Section 5.2.2 define sa naive dependency relation which is sound but not precise. Section 5.2.3 follows by refining this definition to be more precise by removing overapproximated dependencies. Finally, Section 5.2.4 improves precision again by adding additional detailed information to the dependencies.

5.2.1 How are Dependencies introduced?

A module can only depend on another module if it requests or uses some kind of information from modules. There is thus an obvious dependency relation of a child module on its parent module. If the parent module changes, the child module might not even exist any more. We do not need to do any tracking of these dependencies, since they are already present in the module tree.

Then, there are two other ways in which information can be obtained from other modules. First of all there are queries which request information from the scope graph. Queries request edges and data edges from scopes in the scope graph, and can reach other modules.

Second, it is possible to determine the value of a variable from the unifier of another module. However, we do not actually need to consider these unifier dependencies. The only way to obtain a variable from another module is if that module exposes it in its scope graph. We can obtain such a variable by querying the module in question, in which case we already introduce a dependency. It is also possible for another module to obtain this variable via a query, and to put it in its scope graph. If we obtain a variable this way, there will still be a transitive dependency on the module creating the variable.

In addition, it is not possible to assign values to variables from other modules, as explained in Section 4.3.4. This means that there cannot be a reverse dependency of the module owning the variable on the value that another module assigns to it. Since queries are the only constraint able to obtain information from other modules, only queries need to be considered for the introduction of dependencies.

Syntax for Dependencies We write $m_1 \hookrightarrow m_2$ when module m_1 depends on module m_2. We define the transitive closure of dependencies as $m_1 \hookrightarrow^* m_2$, i.e. m_1 transitively depends on m_2. We distinguish two specific kinds of dependencies, *incorrect* and *irrelevant* dependencies. An incorrect dependency is a dependency that cannot be relevant for incrementality. A dependency $m_1 \hookrightarrow m_2$ is irrelevant for a specific change c to m_2 whenever m_1 is not affected by c.

Definition 5.2.1. ***Dependency****: a dependency of module m_1 on module m_2, written $m_1 \hookrightarrow m_2$, means that m_1 directly uses information from m_2 or made available by m_2.*

$$m_1 \hookrightarrow^* m_2 \implies m_1 \hookrightarrow m_2 \vee \exists m_3.\, (m_1 \hookrightarrow m_3 \wedge m_3 \hookrightarrow^* m_2) \tag{5.1}$$

Definition 5.2.2. ***Incorrect dependency****: whenever there is a dependency $m_1 \hookrightarrow m_2$, but there does not exist a change to m_2 such that m_1 is affected.*

Definition 5.2.3. ***Irrelevant dependency****: a dependency $m_1 \hookrightarrow m_2$ is irrelevant for a change c to module m_2, whenever m_1 is not affected by the change c.*

5.2.2 Naive Dependencies

Multiple mechanisms for determining dependencies are possible. For example, one could identify dependencies from the queries directly. All the paths that could be matched by the query which are available in the scope graph could be classified as the dependency paths. The dependencies would then be the owners of the scopes and declarations encountered on the path. Figure 5.1 shows an example of such a naive dependency relation. The query is represented by the B in the bottom left corner. The query follows paths of the form P* and looks for a declaration with the name B. The resolution path is $S_{A_M} \cdot \mathsf{P} \cdot S_A \cdot \mathsf{P} \cdot S_P \cdot \mathsf{P} \cdot S_G \cdot \textsc{decl}(B)$. This would mean that A_M depends on all other modules, which is correct.

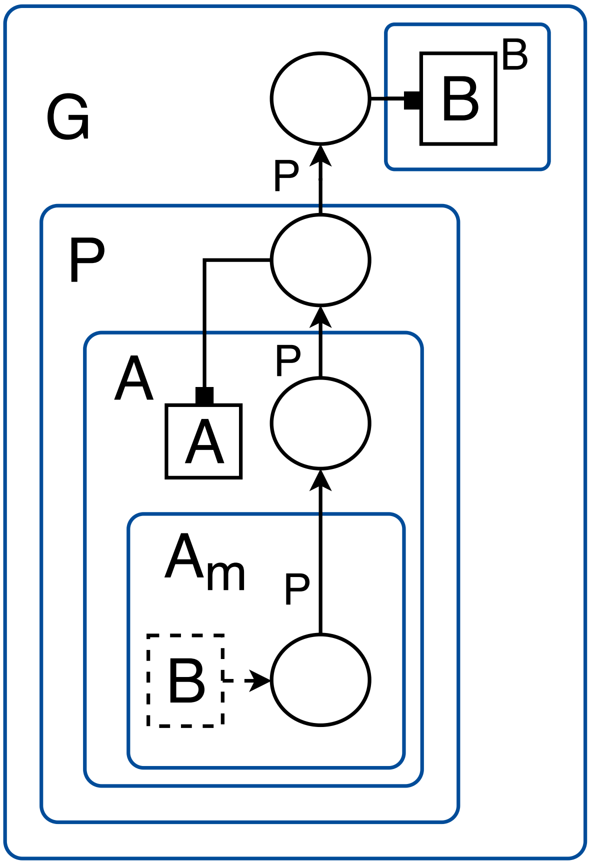


Figure 5.1: A scope graph showing query resolution with modules. In the bottom left a query for a name B, which resolves to the declaration of B in the top right, traversing all other modules in the process.

Precision The naive dependency relation is not very precise. A lot of dependencies are introduced that are not actually relevant. This is due to the fact that the naive dependency relation is using reachability and not visibility. In essence, the naive dependency relation also shows *potential* dependencies. Usually, declarations are shadowed by other declarations. Whenever a query resolves to a declaration via a path p, we do not need to consider any path p' that is less *specific*. That is, we do not need to add dependencies for declarations and paths that are not reached because they are shadowed by other declarations. We make this refinement in Section 5.2.3.

Definition 5.2.4. *A path p is more **specific** than a path p' for some query q, if q will chose p' over p if it is available. Formally, if $<_l \vdash p <_p p'$.*

Definition 5.2.5. ***Precision**: the precision of a dependency relation depends on the number of incorrect and irrelevant dependencies that it finds. A dependency relation becomes more precise as it finds less incorrect dependencies and as it provides more information with which irrelevant dependencies can be avoided.*

Soundness The naive dependency relation ensures that all the *current* dependencies are discovered. However, care needs to be taken when considering child modules. A child module m has permission to extend all the scopes that are passed to it from its parent. That is, m can have permission to extend a scope s where $owner(s) = m_p$ and $m \neq m_p, m_p \in parents(m)$. This means that a change to m can affect m_p. With the naive dependencies, this dependency is not captured explicitly.

We can address this in a few different ways. First, it is possible to say that a module m depends on all its children. However, this would introduce a lot of additional dependencies that are often not necessary. Another option is to add a dependency on the owners of the traversed edges in addition to the owners of the scopes. While this would make the dependency on child modules explicit, it is not enough by itself. It is possible for a module m which previously contributed no edges nor scopes, to introduce a new *child edge* in a scope of one of its parents. Because it previously did not contribute anything to the scope graph, there cannot have been a direct dependency on this module. However, the newly added edge can influence the resolution of existing queries. This would imply that we missed a dependency.

Definition 5.2.6. ***Child edge***: *an edge* $s \xrightarrow{l} s'$ *or* $s \xrightarrow{r} d$ *where the module* m *introducing the edge is not the owner of* s*, but rather a descendant of the owner of* s*, i.e.* $m \in descendants(owner(s))$*.*

Instead, it is possible to consider a change to m which modifies the edges in a scope s owned by a parent module m_p as a change to m_p and not to m. Intuitively, it makes sense that a change to a scope s owned by module m is a change to m. By moving the change to a module to the module which it affects, we essentially hide the details of child modules from the dependencies. Assuming that we can determine the changes to the scope graph of a module in terms of added and removed edges in particular scopes, this is quite easy to apply. As such, we will use this definition of changes for all incremental strategies that can do so (see Chapter 6). It follows that a dependency relation can only be considered 'correct' in the context of how changes are handled. This gives us the following definition of soundness for dependency relations.

Definition 5.2.7. ***Effective change***: *an effective change* c *to module* m *is a change* c' *to module* m' *with* $m' = m \vee m' \in descendants(m)$ *which modifies the edges of a scope owned by* m*.*

Definition 5.2.8. ***Soundness***: *A dependency relation is considered* ***sound*** *whenever an* effective change *c to module m which affects the result of a query in m', implies that there is a dependency* $m' \hookrightarrow^* m$*.*

5.2.3 Refined Dependencies

While the naive dependency relation is sound when using *effective changes*, it is not very precise. Usually, queries resolve to something in the scope graph and then do not visit the remainder of the scope graph. For the most part, only queries that do not have any results actually depend on all the paths that they can reach. Figure 5.2 shows an example of this. The query is represented as the x in the bottom left. The query is defined to look for declarations with the name x with the path regex P* (query $s \xmapsto{\mathsf{P*:}} \textsc{decl}(x_j)$ as x_k).

With the naive dependency relation as defined above, we would end up with the dependency of A_M on A, P and G. In reality it only depends on A. We refine the dependencies to be the owners of the scopes and declarations that appear on the paths of the query results. If the query does not resolve to anything, we use our definition for the naive algorithm. We call this the *Refined Dependency Relation*.

However, using *only* the path of the result to determine dependencies is not always correct. Consider the scope graphs in Figure 5.3. In the left scope graph there is a dependency of A on B because the query resolves to the x declared in B. Now, C is modified and also gets a declaration x, resulting in the scope graph on the right. The query in A would now resolve to the x in C instead. However, since we did not record a dependency of A on C, the incorrect previous result would be used. Clearly a dependency of A on C was missed here.

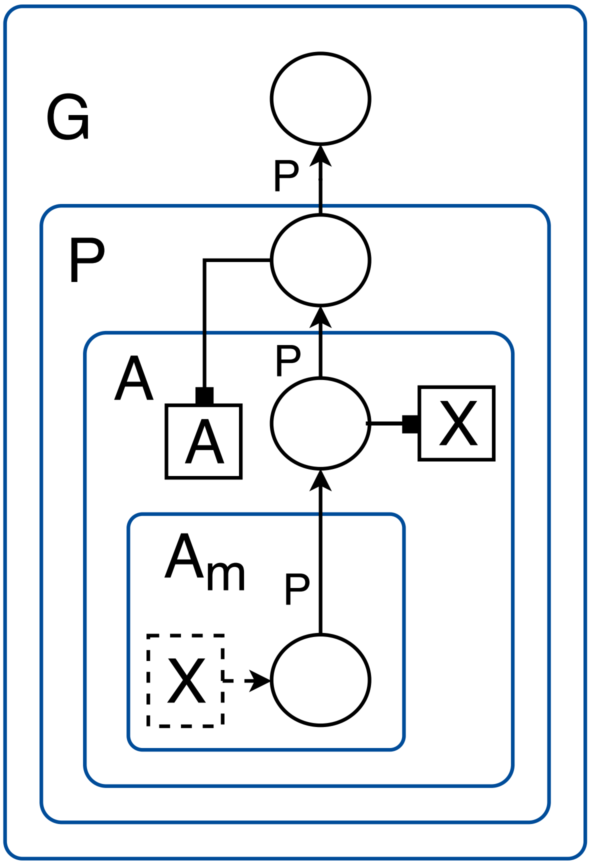
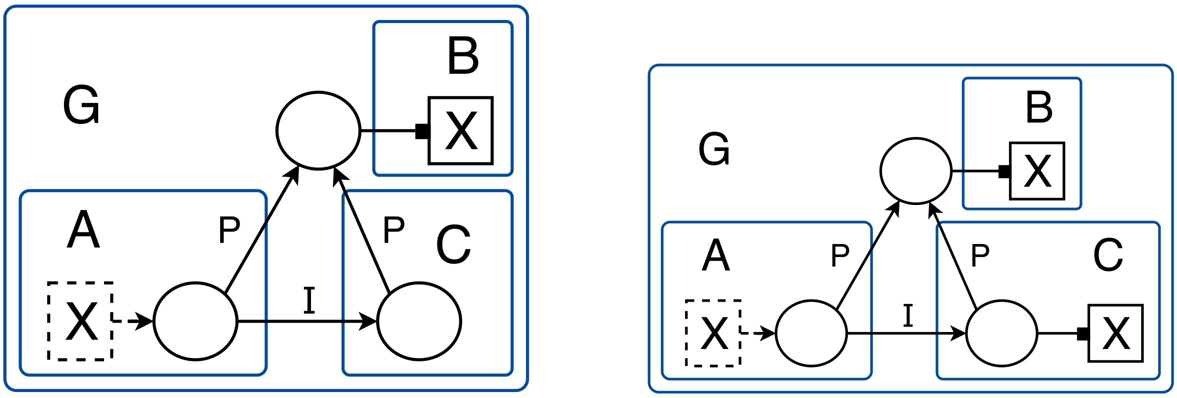


Figure 5.2: A scope graph showing that not all dependencies are relevant.



(a) Original scope graph

(b) Changed scope graph

Figure 5.3: Scope graphs showing the problem with the simple dependencies.

To counter this problem, we refine our dependency relation as follows. If a query does not resolve to a declaration, we depend on the owners of the scopes on all the paths that can be reached by the query. If a query resolves to some declaration d via some path p, we depend on all the owners of the scopes on the path p as well as on all the owners of scopes on any path p' that is more specific than p.

The added part describes that we need to consider all the scopes where a declaration or an edge *could* be added which would potentially shadow our current resolution. If such a declaration or edge is added, the answer to our query can change. With this definition, we are now both sound and precise. The dependencies we consider for a query are based on *all* the scopes and *only* those scopes where the introduction of new edges or declarations could alter the result of our query.

We cannot obtain a higher precision in terms of simple dependencies of the form m depends on m' without losing soundness. However, it is possible to add more details to our dependencies to make them more useful in determining when these dependencies are actually relevant, which we will discuss next.

5.2.4 Adding Details

While the refined dependency relation cannot be made more precise, we would still like to reduce the number of dependencies that need to be considered for a particular change. To achieve this, we add additional information to the dependencies which can be used to determine if a particular change can affect a particular dependency (Section 5.3).

Since our dependencies are created on the basis of all the scopes and edges that could affect the result of the query, it makes sense to add this information to the dependency. We record the scope and the edge label of each path considered for the dependency. For example, if module A performs a query and requests edges with the label l in a scope of module B, i.e. there is a path segment $\ldots \cdot S_B \cdot \mathsf{l} \cdot \ldots$, then there is a dependency $A \hookrightarrow B \mid \langle S_B, \mathsf{l} \rangle$. In other words, A depends on B because A requested l edges in scope S_B, a scope owned by B. If we record this information with all the paths of our refined dependency relation, we know exactly why we have dependencies on each module. In fact, we already have to determine these scopes and edges to determine the dependencies in the first place, which means that adding such information does not incur a significant extra cost.

In addition, we add the information of all the paths that were resolved to the query. This is also information we already have. With these paths, we can determine if the removal of an edge in the scope graph affects the result of the query more precisely than with just the considered paths. If a removed edge is not on the resolved paths, we know for sure that it cannot affect the result of our query. Finally, instead of recording each scope and label combination individually, we write $A \hookrightarrow B \mid \langle V_e, V_d, R \rangle$, where V_e is the set of visited edges, V_d is the set of visited data edges and R is the set of edges on the paths to the resolutions. We specify this further in Section 5.2.5.

The Cost of Tracking Precise Dependencies in Statix Recording such precise dependencies might seem quite expensive, but with the way the name resolution algorithm is implemented in Statix, it is actually quite trivial. The name resolution algorithm requests paths in the order of increasing specificality and stops as soon as it finds a most specific target. As such, it will *only* and *exclusively* visit those paths that are relevant for our dependency tracking. All the edges that could potentially influence the answer to the query are exactly those edges requested by the name resolution algorithm. We only add a small tracking method through which the name resolution algorithm requests edges and data edges. These accesses are then stored in a map, incurring only a very small cost.

5.2.5 Formally Specifying Dependencies

In this section we provide the formal specification of our dependency relation as a judgement. First, we specify the naive dependency relation to show the basis of our approach. Then, we will extend this specification to the refined dependency relation with added details. We do not separate this into a relation with details and one without details, since the introduction of the dependency happens in the same way.

Notation of Queries and Name Resolution The rules for name resolution in the scope graph are shown in Figure 2.4 and explained in Section 2.3.2. We will use a different notation for queries in the form of $\mathsf{query}\ \mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l\ \mathsf{in}\ s \overset{r}{\longmapsto} \overline{(p, d)}$, where each pair (p, d)

describes a path p and a data term d that was found by the query. For brevity, we will sometimes omit the well-formedness predicates and the orderings when they are not relevant. We use the letter q to refer to queries. We project out the *WFD*, *WFL*, $\leqslant_d$ and $<_l$ from a query q by denoting them as WFD_q, WFL_q, $\leqslant_{d_q}$ and $<_{l_q}$ respectively.

Dep-Naive

$$\frac{s'' \in \mathsf{scopes}(p) \qquad \begin{array}{c}(\mathcal{G} \vdash p : s \twoheadrightarrow s') \vee (\exists d.\, (\mathit{WFD}_q, \mathit{WFL}_q, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d)) \\ \mathit{WFL}_q \vdash p \text{ OK}\end{array} \qquad m_1 = \mathit{owner}(q) \qquad m_2 = \mathit{owner}(s'') \qquad m_1 \neq m_2}{\mathcal{G}, q := \left(\mathsf{query\ in}\ s \overset{r}{\longmapsto} \overline{(p, d)}\right) \vdash m_1 \hookrightarrow m_2}$$

Figure 5.4: Naive dependency relation which creates dependencies on all scopes and declarations that can be reached by a query.

Naive Dependency Relation The first part of the relation, $\mathcal{G} \vdash p : s \twoheadrightarrow s'$, states that there is a path p from our source scope s to some other scope s'. This part represents all the paths that can be made in the scope graph, but is restricted by $\mathit{WFL}_q \vdash p$ OK to only paths accepted by the query.

The second part, $\exists d.\, (\mathit{WFD}_q, \mathit{WFL}_q, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d)$, states that there is some declaration that can be reached from s via the path p under relation r. This part represents the full resolution paths.

$\mathit{WFL}_q \vdash p$ OK specifies that we only consider valid paths for our relation. This restricts the resolution to only paths that are accepted by the query. Finally, we consider all the scopes that appear in our path with $s'' \in \mathsf{scopes}(p)$ and specify that m_1 is the owner of the query, m_2 is the owner of a scope on the path and that $m_1 \neq m_2$.

Considered Scopes and Reached Scopes To be able define the refined dependency relation, we need to first describe how we determine the scopes we consider and the scopes that we reach. We add an auxilary relation, `considers` which specifies that a query $\mathsf{query\ in}\ s \overset{r}{\longmapsto} \overline{(p, d)}$ *considers* an edge $s' \xrightarrow{l} s''$ or data edge $s' \overset{r'}{\longrightarrow}\blacksquare\ d'$. This means that either a path using the aforementioned edges could be more specific than resolution paths found by the query, or that the query did not find any resolution path and that paths using the aforementioned edges would be valid resolution paths.

For ease of reading and notation, we define the *considers* relation on a pair (s, r) rather than specifying the full query $\mathsf{query\ in}\ s \overset{r}{\longmapsto} \overline{(p, d)}$. Furthermore, we make a distinction between $\texttt{considers}_e$ and $\texttt{considers}_d$ to distinguish between normal edges and data edges respectively.

The rules for the `considers` relation are shown in Figure 5.5. We explain these rules in more detail.

For resolutions that succeed, i.e. $\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\longmapsto} d$, we are interested in all the paths that satisfy the query which are more specific than the path p, but which currently do not yield a resolution. With more specific we mean paths p' where $<_l \vdash p' <_p p$. On a scope graph $\mathcal{G}'$ where $\mathcal{G} \subset \mathcal{G}'$, i.e. $\mathcal{G}'$ has additional edges, the result of the query could be one of these paths p' instead of the current resolution. In essence, we want to know the edges $s \xrightarrow{l} s'$ and data edges $s \overset{r}{\longrightarrow}\blacksquare\ d$ that could be added to $\mathcal{G}$ which can affect the result of our query. To this end, we define the relations *considers*$_e$ and *considers*$_d$. The relation $\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_e\ (s', l)$ holds for all (s', l) which can affect the result of the visibility query $\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\longmapsto} d$. Likewise, the relation $\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_d\ (s', r)$ holds for all (s', r) which can affect the result.

The rules for edges need to find not only the paths that do match, but also those that could be matched. First, it selects a path in the graph to a scope s' reachable from s. Such a path is relevant if it is more specific (in the case a resolution exists) and could be extended to some well-formed path. This is achieved by taking a label l and some suffix x and checking if adding the suffix $l\ x$ to a path would be valid. The labels on the path are determined with $w = word(p')$. Only if $(w\ l\ x)$ is in the regular language which the query accepts, we know that the query could be affected by additions for label l in scope s'.

The rules for data are much the same, but are simpler, since they only need to check if the path to s' is well-formed. This is because declarations are requested only on well-formed paths. This property is denoted by $\mathit{WFL} \vdash p'\ \textsc{ok}$.

NR-ConsidersEdge-Result

$$\frac{\begin{array}{c}\exists pd.\,(\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \xmapsto{r} d)\\ \mathcal{G} \vdash p' : s \twoheadrightarrow s' \qquad <_l \vdash p' <_l p \qquad w = word(p') \qquad l \in \mathcal{L} \qquad x \in \mathcal{L}^* \qquad (w\,l\,x) \in \mathit{WFL}\end{array}}{\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_e\ (s', l)}$$

NR-ConsidersEdge-NoResult

$$\frac{\begin{array}{c}\nexists pd.\,(\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \xmapsto{r} d)\\ \mathcal{G} \vdash p' : s \twoheadrightarrow s' \qquad w = word(p') \qquad l \in \mathcal{L} \qquad x \in \mathcal{L}^* \qquad (w\,l\,x) \in \mathit{WFL}\end{array}}{\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_e\ (s', l)}$$

NR-ConsidersData-Result

$$\frac{\begin{array}{c}\exists pd.\,(\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \xmapsto{r} d)\\ \mathcal{G} \vdash p' : s \twoheadrightarrow s' \qquad <_l \vdash p' <_l p \qquad \mathit{WFL} \vdash p'\ \textsc{ok}\end{array}}{\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_d\ (s', r)}$$

NR-ConsidersData-NoResult

$$\frac{\nexists pd.\,(\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash p : s \xmapsto{r} d) \qquad \mathcal{G} \vdash p' : s \twoheadrightarrow s' \qquad \mathit{WFL} \vdash p'\ \textsc{ok}}{\mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r)\ \mathsf{considers}_d\ (s', r)}$$

Figure 5.5: Formal definition of the `considers` relations, i.e. all the (scope, label) and (scope, relation) combinations that are considered by a query.

Stuck Queries Require Dependencies In addition, dependencies also need to be recorded if a query is stuck. However, stuckness of queries is not captured by the declarative semantics. Name resolution either finds a path or doesn't find a path. However, it is possible for a query to get stuck. In those cases, our model should still provide the necessary dependencies to recover from such a scenario. For a stuck query, all the edges which can be considered up to the point where the query is stuck need to be recorded. That is, all the scopes and edges the query would consider if it would, instead of getting stuck at location (s, l) or (s, r), conclude that there is a matching declaration at some path following (s, l) or of the form (s, r). The query then follows the rules for `NR-ConsidersEdge-Result` and `NR-ConsidersData-Result` up to the point where it is stuck.

Added Details For the added details, we introduce a few new sets. First, we define V_e and V_d to be the sets of all the scope label combinations (s, l) and the scope relation combinations (s, r) that were considered by a query respectively. We define R to be a set of all the paths that were resolved by a query. We extend our dependency relation with details, which we write as $m_1 \hookrightarrow m_2 \mid \langle V_e, V_d, R\rangle$ to denote that module m_1 depends on module m_2 with detailed information $\langle V_e, V_d, R\rangle$.

Refined Dependency Relation with Details We then define our dependency relation in terms of the `considers` relations. Figure 5.6 shows the dependency relation. This relation works as follows. We assert that we depend on all m_2 for which V_e or V_d is not empty. V_e and V_d are computed as the union of all the (scope, label) and (scope, relation) combinations that are considered by the query, for which the owner of the scope is m_2. Finally, R contains all the paths to declarations that can be resolved by the query.

$$\textsc{Dep-Refined-Detailed}\quad \frac{\begin{array}{c} m_1 = \text{owner}(q) \qquad \exists m_2 \in M .\, m_1 \neq m_2 \\ V_e = \left(\bigcup_{s' \in \text{scopes}(\mathcal{G}), \text{owner}(s') = m_2, l \in \mathcal{L}} \mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r) \text{ considers}_e\ (s', l) \right) \\ V_d = \left(\bigcup_{s' \in \text{scopes}(\mathcal{G}), \text{owner}(s') = m_2} \mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l, \mathcal{G} \vdash (s, r) \text{ considers}_d\ (s', r) \right) \\ V_e \neq \varnothing \vee V_d \neq \varnothing \qquad R = \left\{ p \mid \mathit{WFD}_q, \mathit{WFL}_q, \leqslant_{d_q}, <_{l_q} \vdash p : s \overset{r}{\rightarrowtail} d \right\} \end{array}}{\mathcal{G}, q := \left(\mathsf{query\ in}\ s \xhookrightarrow{r} \overline{(p, d)} \right) \vdash m_1 \hookrightarrow m_2 \mid \langle V_e, V_d, R \rangle}$$

Figure 5.6: Refined dependency relation with added details

5.3 Determining the Impact of a Change

One of the key parts of incrementality is to determine the impact of a change. Whenever a change is made, we need to determine which modules need to be recomputed and which modules are unaffected based on the dependencies recorded. This section describes how edits to files can ultimately be mapped to affected modules. First, Section 5.3.1 describes how edits to files are mapped to modules. Then, Section 5.3.2 shows how changes to modules can be represented as changes to their scope graphs after reanalysis. Section 5.3.3 explains how scope graphs can be compared to create a diff of added and removed edges, followed by Section 5.3.4 which shows how irrelevant information in the scope graph can be ignored for these diffs. Finally, Section 5.3.5 combines dependencies and diffs to determine affected modules.

5.3.1 Mapping Edits to Modules

A *changeset* maps the edits that have occurred to files in the project to the modules they occur in. The granularity of changesets can range from file level to the level of individual Statix modules, depending on the granularity that is used by the IDE. In addition, changesets can have different levels of details, but they contain at least sets for added modules, removed modules, modules for which the AST changed and modules for which the AST did not change.

Definition 5.3.1. ***Changeset****: a tuple which maps the edits that were made to the project to modules. The tuple has to contain at least sets for*

- *modules which were added*
- *modules which were removed*
- *modules for which the AST changed*
- *modules for which the AST did not change*

An incremental run has as input the output of some previous run and the changes that were made to the project since that run. From these, we first build a changeset. An example of building a changeset from changed files is shown in Listing 5.1. Note that while we are using files in this example, Statix only needs some *unit* which has a unique name for the project and has a corresponding AST. For Spoofax, these units are files, but it is possible to offer different units instead if necessary or wanted.

The map from files to modules in this example is extracted from the output of the previous run. It is created on the initial run by the createModule instructions executed.

```
procedure changeSet($F_a$, $F_r$, $F_c$: Files, $M$: Map(File, Module)):
    $M_a$ := $\{$createModule$(m, f) \mid f \in F_a\}$
    $M_r$ := $\{m \mid f \in F_r, m = M(f)\}$
    $M_c$ := $\{m \mid f \in F_c, m = M(f)\}$
    $M_u$ := codomain$(M)$ \ $M_r$ \ $M_c$
    return $(M_a, M_r, M_c, M_u)$
```

Listing 5.1: An example implementation for building a changeset from changed files

5.3.2 From Changed Modules to Changed Scope Graphs

In true separate compilation, we would be able to determine the new interface of a module from its definition. In Statix however, the construction of the scope graph is intertwined with the full analysis of the module. The scope graph follows as a solution of our constraint set, but determining this solution often requires information from other scope graphs in the project. It depends on the source language and the Statix specification if the interface of a module can be determined in isolation or not. To support as many different specifications and languages as possible, we do not assume that this is the case.

Instead, we introduce optimistic strategies in Section 6.1 which reanalyze a module in the context of the previous analysis, and then determine the changes to its interface. We then restore consistency by also reanalyzing modules that are dependant on those changes, until a fixpoint is reached.

Changes to the Input and Output First, let us see what changes in the input. The input of a module is everything that is passed to its module boundary. For top level modules, this consists of the global scope and the AST corresponding to the module. The AST is effectively converted to some set of constraints via the Statix specification. So, it is this set of constraints, which include the AST in some way, that changes as a result of an edit to a part of the project. The effect of these changes is then some possibly modified scope graph, unifier, set of failed constraints and set of stuck constraints.

It is possible for the constraint set to be unsatisfiable or for the solver to get stuck. Formally speaking, the semantics of Statix do not require that the solver gives any solution in the case of failure. In practice however, the solver will do a best effort attempt to construct a partial solution. Because we would like the incremental analysis to work in an IDE while the user is typing, it is likely that some of the analyses fail. It is thus important to consider how these non-accepting states are treated.

We can map all of these changes to changes in the scope graph. Any constraint that fails or gets stuck, is effectively ignored and does not have its effect, which will express itself as changes to the scope graph and/or changes to the unifier. The changes to the unifier are only relevant to other modules if they affect the values of variables that are in the scope graph. We can thus express these changes to the unifier as changes in the scope graph by using fully substituted scope graphs, or by using the new and old unifiers when comparing terms in the scope graph.

This leaves us with only changes to the scope graph which we need to consider. We can express the transformation of one scope graph $\mathcal{G}_a$ into another scope graph $\mathcal{G}_b$ as a sequence of one or more individual *solution changes*, expressed as changes to the state. The different possible *solution changes* are shown in Figure 5.7.

Solution changes	
m creates a child module m'	$\langle \overline{M} \mid \overline{\mathcal{G}} \rangle \to \langle \overline{M}[m' \to m] \mid \overline{\mathcal{G}}[m \to \epsilon] \rangle$
m deletes a child module m'	$(\langle \overline{M} \mid \overline{\mathcal{G}} \rangle \text{ where } \overline{\mathcal{G}}(m') = \epsilon) \to \langle \overline{M}[/m'] \mid \overline{\mathcal{G}}[/m'] \rangle$
m creates a scope s	$\langle S_m, E_m, D_m \rangle \to \langle s; S_m, E_m, D_m \rangle$
m deletes a scope s	$\langle s; S_m, E_m, D_m \rangle \to \langle S_m, E_m, D_m \rangle$
m introduces an edge $s \xrightarrow{l} s'$	$\langle S_m, E_m, D_m \rangle \to \langle S_m, (s \xrightarrow{l} s'); E_m, D_m \rangle$
m deletes an edge $s \xrightarrow{l} s'$	$\langle S_m, (s \xrightarrow{l} s'); E_m, D_m \rangle \to \langle S_m, E_m, D_m \rangle$
m introduces a data edge $s \xrightarrow{r}\!\!\blacksquare\; d$	$\langle S_m, E_m, D_m \rangle \to \langle S_m, E_m, (s \xrightarrow{r}\!\!\blacksquare\; d); D_m \rangle$
m deletes a data edge $s \xrightarrow{r}\!\!\blacksquare\; d$	$\langle S_m, E_m, (s \xrightarrow{r}\!\!\blacksquare\; d); D_m \rangle \to \langle S_m, E_m, D_m \rangle$

Figure 5.7: The definition of the different *solution changes*.

The removal of a module can be represented as the consecutive removal of all of its scopes, edges and data edges: $\langle S_m, E_m, D_m \rangle \to^* \langle \epsilon \rangle$.

In this way, we can translate a change to the AST of a module to a sequence of solution changes to its scope graph. However, there are some technical details that need to be solved to actually create this comparison between scope graphs and make it useful.

5.3.3 Comparing Scope Graphs

In our model, it is possible to determine the difference between two scope graphs. By comparing the scope graph of a module in the previous analysis with its scope graph in the current analysis, we can determine from the dependencies which other modules are affected by these changes. Optimistic strategies can use these so-called diffs in a multi-phase approach to only reanalyze modules whenever they are affected by the changes. We use the term *scope graph diff* to refer to the difference between two scope graphs.

Definition 5.3.2. ***Scope graph diff**: the difference between two scopes graphs in terms of added and removed scopes, added and removed edges, and added and removed data. Depending on the strategy, additional information can be present in a diff.*

There are some challenges in creating these scope graph diffs. In general, computing the difference between two graphs is an NP-complete problem (computing the maximum common subgraph) [24]. However, scope graphs have a very structured shape and have some properties that make it possible to compute the difference. A few different techniques can be applied to constrain the graph enough to be able to compute the difference between graphs. It is not necessary to determine the maximum common subgraph. In fact, we only want to know two things. First, we want to know if a scope is still the same scope. Second, we want to know if the results of queries have changed, which is split into the paths that are followed and the data in the results of queries. These questions are answered in the following subsections.

Scope Identity: Keeping Scopes the Same The original implementation of Statix uses a simple counter for scope identity. Whenever a scope is created, it gets the value of the counter as its name and the counter is incremented. However, combined with the fact that evaluation order of constraints does not have to be the same, we cannot tell if the scope with identity 1 is the same scope when we see it in different analyses.

We address the problem of 'is a scope still the same scope' with *scope identity*. We want to have some kind of identity that uniquely identifies a scope. This identity has to be consistent between multiple analyses and must not be bound to the edges or data in the scope graph. The semantics of Statix specify that when a scope is created, the variable describing the scope is fresh, but does not guarantee that scope identity is the same across runs. As such, we had to change this for our incremental model. We create scope identities based on a stack trace of the predicates that have been visited by the solver. Because each module uses a separate solver, this means that the stack trace consists of all the predicates since the last module boundary. A scope is considered to be the same scope if it has the same owner, the same name and has the same predicate trace.

Because the scope identity fixes the scopes, we can compare the edges and the data in scopes directly. This is a relatively easy operation, since edges and data are stored per scope already. We can then directly compare this information to determine added and removed edges and data.

Converting Changes into Effective Changes Recall that for our dependencies it was relevant that a change to a scope owned by m is expressed as a change to that module, even if that change occurs in a descendant of m. We call this an *effective change*. As such, we also need to create an *effective diff*. By simply using the owner of each scope that is changed as the module in which the change occurs, we can create effective diffs. If our comparison encounters an edge $s \xrightarrow{l} s'$ in the scope graph of module m, then it records that edge with the owner of s rather than m. As such, all contributions of child modules to scopes of their parents are moved to their parents easily.

Adding more Details: Names Because data can be any term in Statix, it becomes more difficult to track this information. If we look at the queries that occur in Statix, then we see that most queries only request a single name. With *name* we refer to some name in the source code of a program. Names are represented as a combination of a namespace and some number of terms in Statix. In most cases, the latter is just a single string. For example, a variable `x` could have the name `Var{"x"}` and a class `MyClass` could have the name `Class{"MyClass"}`. In a language where one can refer to the nth method with a particular name, it might make sense to use names of the form `Method{1}`. We can identify these names in the scope graph and compute the difference in terms of names.

Definition 5.3.3. ***Name****: an occurrence of some name in the source code of a program, represented in Statix by a namespace and some number of terms.*

Queries that only match declarations with one specific name are identified. The corresponding name is then stored with any dependencies created as a result of that query. This name can be used by the strategy to treat it separately. When making comparisons of scope graphs, all data edges which are associated with a specific name are identified. Changes to these names are then stored in the diff. If the data corresponding to the name changes, e.g. the type of some declaration has changed, it will show up in the diff with its corresponding name. As such, we can determine which queries are affected by the changes more precisely.

5.3.4 Sufficient Information for a Diff

From the perspective of a module itself, it is important that all information in the scope graph is available. However, from the perspective of a different module, certain parts of the scope graph are effectively inaccessible. As such, we define two interfaces, the internal interface and the external interface.

The internal interface contains all the information in the scope graph of the module. Any descendant of a module is able to access the internal interface of that module. Because descendants can be passed any scope in the internal interface of a module, they could potentially access any part of the module and even introduce edges to it.

The external interface contains only those scopes, edges and declarations that are visible to other modules. Isolated parts of the scope graph, a subsection of the scope graph that has no incoming edges from other subsections, are not in the interface unless they are exposed via a declaration. A scenario in which this occurs is exposing scopes through types. To model object oriented languages in Statix, classes are usually modeled as having a type that contains the scope of the class. The class declaration exposes its scope through its type. Whenever fields or methods are requested on an object of the corresponding class, its scope is determined from its type and the corresponding field or method can be looked up.

Methods are an example of a declaration with a scope that is not exposed to the outside. While the signature of a method needs to be known, the contents of a method are irrelevant for callers. As such, the scope of a method is not accessible to anyone but the method itself, and forms an isolated part of the scope graph. As such, method scopes will not be part of the external interface of a module.

For our scope graph diffs, we only need to consider the external interface of a module, since the other parts cannot be reached by other modules, and can thus not affect them. Of note however are child modules, which have access to the internal interface of a module. However, if a module is reanalyzed, all its child modules are also reanalyzed. As such, changes to the internal interface will have already been propagated to child modules, and need not be considered by our diffs.

External Interface The external scope graph is a graph that contains all scopes that are reachable from other modules. If a scope is referenced by a declaration or edge that resides in a different module (which is not a descendant of said module), it needs to be in the external interface. If any such scopes exist, then all reachable scopes (via edges or data with associated scopes) also need to be included in the set. We formally define the external interface in Figure 5.8 and provide an algorithm to compute it in Listing 5.2.

Our formal definition is as follows. P_m contains all the scopes from parents that m can extend. Any scope not reachable from a parent scope is not in the interface. Then, R_m describes the reachable scopes, which are all the parent scopes and all the scopes that are exposed via declarations, e.g. through types. Note that these declarations are also dependent on R_m. As more scopes are 'discovered', more declarations are reachable. Finally, we define $\hat{S}_m$, $\hat{E}_m$ and $\hat{D}_m$ in terms of R_m. Only reachable scopes are in $\hat{S}_m$, only edges from reachable scopes are in $\hat{E}_m$ and only data edges from reachable scopes are in $\hat{D}_m$, filtered to only scopes and edges owned by m (in the scope graph of m.

External interface $\hat{\mathcal{G}}_m = \langle \hat{S}_m, \hat{E}_m, \hat{D}_m \rangle$ with

$$
\begin{aligned}
P_m &:= \{s \in \mathit{Scopes} \mid (m \text{ has permission to extend } s) \wedge \mathit{owner}(s) \neq m\} \\
R_m &:= P_m \cup \{s \in \text{scopes}(d) \mid d \in \mathit{Decls} \wedge \exists s', r.(s' \overset{r}{\longrightarrow\!\!\blacksquare} d) \in \hat{D}_m\} \\
\hat{S}_m &:= \{s \in R_m \mid \text{owner}(s) = m\} \\
\hat{E}_m &:= \{s \xrightarrow{l} s' \mid s \in R_m \wedge l \in \mathit{Labels} \wedge s' \in \mathit{Scopes}.(s \xrightarrow{l} s') \in E_m\} \\
\hat{D}_m &:= \{s \overset{r}{\longrightarrow\!\!\blacksquare} d \mid s \in R_m \wedge r \in \mathit{Rels} \wedge d \in \mathit{Decls}.(s \overset{r}{\longrightarrow\!\!\blacksquare} d) \in D_m\}
\end{aligned}
$$

Figure 5.8: Description of the external interface of a module scope graph . Only the subgraph of $\mathcal{G}_m$ that is reachable from parents is in the external interface.

```
procedure ExternalInterface($m$):
    Q := ParentScopes
    while (Q != $\varnothing$):
        s := pop(Q)
        if (owner(s) = m):
            $\hat{S}_m$ += s

        $\forall(s \xrightarrow{l} s_2) \in E_m$:
            $\hat{E}_m$ += $s \xrightarrow{l} s_2$

        $\forall(s \overset{r}{-\!\!\blacksquare}\, d) \in D_m$:
            $\hat{D}_m$ += $s \overset{r}{-\!\!\blacksquare}\, d$
            Q += $(\texttt{scopes}(d) \backslash \hat{S}_m)$
```

Listing 5.2: Algorithm for computing the external interface of a module.

5.3.5 Combining Dependencies with Diffs

The last step in the process is to combine the dependencies with the scope graph diffs to conclude a set of modules which could be affected and need to be reconsidered. We provide an algorithm in pseudocode which determines exactly that in Listing 5.3. Recall that each dependency is represented as $m_1 \hookrightarrow m_2 \mid \langle V_e, V_d, R \rangle$, with V_e and V_d sets of visited edges and R the set of reached edges (see Section 5.2.4). Dependencies are stored as an index from scopes and labels to dependencies. Each edge that appears in V_e and V_d is in the index and maps to the corresponding dependency. Our algorithm then works as follows. First, it retrieves the dependency index from the context. It determines which dependencies are relevant for additions, which is any dependency where (s, l) appears in V_e or (s, r) in V_d, i.e. any dependency in the index at (s, l) or (s, r). For removals, this is much the same, but we additionally check if the edge appears in the R set of the dependency. This improves our precision for removals to the maximum possible. If the removed edge is not followed by a query, it cannot be affected. If it is followed, it will be affected. For each dependency found to be relevant, the module depending on the change is added to a set of affected modules, which is returned by the algorithm.

5.4 Concurrent Solving of Modules

Besides the goal of achieving incremental solving, we would like to make the solving process concurrent. Because the solving process is already split up into one solver per module, parallel execution of these solvers seems like a good fit to enable concurrency. In this section we discuss the main techniques and challenges to achieving safe parallel execution of multiple solvers.

5.4.1 Coordinating Solvers

As mentioned in Section 4.3.2, all the solvers are managed by the coordinator. To achieve parallel execution of solvers we introduce a concurrent version of the coordinator as well. The concurrent coordinator is slightly more complex than the non-concurrent version. Instead of a loop where solvers are executed sequentially, the concurrent coordinator has a pool of worker threads available for scheduling solvers. All solvers that have pending work and are waiting to get scheduled are in a queue. Whenever a worker is not already executing a solver,

```
procedure determineAffectedNames($D$: Diff, $Ctx$: Context):
    $Deps$ := getDependencies($Ctx$)
    $(E_a, E_r, D_a, D_r)$ := $D$

    $M$ := $\varnothing$
    // Compute dependencies affected by edge additions
    for each $(s, l) \in E_a$:
        for each dependency $d \in Deps(s, l)$:
            $M$ += dependant($d$)
    for each $(s, r) \in D_a$:
        for each dependency $d \in Deps(s, r)$:
            $M$ += dependant($d$)

    // Compute dependencies affected by edge removals
    for each $(s, l) \in E_r$:
        for each dependency $d \in Deps(s, l)$:
            if $(s, l) \in d.R$:
                $M$ += dependant($d$)
    for each $(s, r) \in D_r$:
        for each dependency $d \in Deps(s, r)$:
            if $(s, r) \in d.R$:
                $M$ += dependant($d$)

    return $M$
```

Listing 5.3: The definition of the algorithm for determining affected modules from a diff and dependencies

it removes a solver from the queue and starts executing it, making as much progress in the solver as possible. Then there are two possibilities.

First, it is possible for a solver to be finished, in which case the worker collects the results for the solver and stores them in the context. The worker is then free to move on to the next solver. Second, it is possible for a solver to be delayed on other solvers. In this case the solver is *not* put back in the queue. Recall that our model uses an observer pattern for delays on critical edges and variables. As soon as a delay is resolved, all solvers waiting on that delay are notified. Upon receiving such a notification, the delayed constraint is reactivated and, if the solver is not already active or in the queue, is put back into the queue. With this mechanism, waiting solvers do not end up in the queue at all, reducing resources required for scheduling. The workers are expected to be working on solving almost all of their time, and no time is actually spent in waiting or blocking on other solvers.

However, it has to be ensured that solvers are not put into the queue whenever they are still being executed. In addition, having the same solver in the queue multiple times should also not be possible, as this could cause multiple workers to try and work on the same solver. We achieve this by using a lightweight compare-and-swap (CAS) mechanism provided by the JVM through volatile variables and atomic classes [20]. An example of a CAS operation is shown in Listing 5.4. We use these operations to ensure that a notification is never missed but is also never ignored whenever it is relevant, without introducing significant locking overhead.

```
$x$ := fetch value from memory
if $x$ = expectedValue then:
    store newValue in memory
    return newValue
else:
    return $x$
```

Listing 5.4: A simple example of a compare-and-swap operation. The entire operation shown in this listing is executed as one atomic operation by the Java Virtual Machine. In other words, no other operations can interleave it.

5.4.2 Safety of Parallel Execution of Solvers

In order to execute multiple solvers in parallel, it must be ensured that the actions of the different constraints cannot interfere with each other. We achieve this by leveraging a property of Statix. Statix constraints can be solved non-deterministically. Regardless of the ordering in which they executed, the solution that we find will be the same. This is achieved by guaranteeing query answer stability through the critical edge mechanism of Statix. A query constraint can only be reduced whenever it is ensured that no edges will be added which can change the result of the query. This mechanism is further explained in Section 2.5.3. As long as it is guaranteed that a change to the scope graph happens before the removal of its critical edge, a query trying to access that edge will either be delayed, or will see the added edge. In other words, either the constraint adding the edge happens before the query, or happens after the query. Queries are then atomic with respect to the scope graph extensions which could change their answer.

Effectively, this means that a query does not need to use locking to access information from other modules. It can use any combination of versions of the scope graphs of other modules as long as the above property is ensured. Every time a query requests the edges with a particular label in a particular scope, it either gets all the edges with that label that will ever exist, or it will be delayed. As such, queries can use any version or even multiple versions of the scope graphs of other modules, while query answer stability is still guaranteed, without requiring any locking.

Unification Variables Unification is another point where care must be taken to ensure correctness. In addition to critical edges, it is also possible to delay on variables of other modules. For example, when the variable of a different module is used in an equality constraint, its value must be known in order to determine if terms are equal. Here, we achieve safety through the observer registration. The changes to variables are already atomic, since there is no separate completeness that needs to be updated as well. The request for the value of a variable either happens before a constraint which assigns that value or after such a constraint. We only need to ensure that whenever a module registers itself as observer of a particular variable, that it is immediately notified if the variable was assigned a value already.

For all the other constraints, their effects are local to their own module. These cannot be observed by other modules and can thus be safely executed in parallel to other modules.

5.4.3 Detecting Stuckness

One of the problems for parallel execution of solvers, is that solving in Statix can get stuck. With a sequential coordinator, detecting stuckness is just a matter of concluding that no progress can be made any more. However, in a concurrent setting this becomes a lot less clear. Let us first define the notion of stuckness.

Definition 5.4.1. *Stuckness An analysis is stuck if there are solvers which only have delayed constraints left, which are waiting on critical edges or unification variables that will never be resolved.*

Since the waiting on critical edges and variables occurs between modules, we have a problem similar to deadlock detection. In our case, stuckness can be detected with a simple counter. Whenever a module is notified of work and is not already working, the counter is incremented. Once the module goes back into waiting, finishes or fails, the counter is decremented. Note that the counter is incremented by a notified module *before* the notifying module decrements the counter. If the counter hits zero, that means that all modules are either done or waiting. We can then collect results and determine if we are stuck or done.

5.5 Conclusion

We now answer **Research Question 2: Based on the conclusions drawn from the literature, how can we create a model for incremental analysis for Statix?**

From literature we have concluded that the techniques of separate compilation and dynamic dependency detection can be used to achieve incrementality for Statix. We contribute an incremental model for Statix based on these techniques. The goal of our model is to facilitate different strategies for incremental analysis, to fit the needs of different source languages. We achieve this goal through the following features of our model.

- **Modules**: We add support for modules, which correspond to modules in the language we are analyzing. Rather than constructing a complete scope graph for the project, we split the scope graph over the different modules. The interface of each module is its part of the scope graph, which contains all information that can be used by other modules. A main challenge in the implementation of our model was the splitting of the solving process over these modules, an approach very different from the original monolithic architecture of Statix. We stayed close to the original architecture by having each component redirect requests to the module which can answer that request. As such, we still provide a view of the system as a whole, without incurring a significant extra cost.

- **Dependencies**: Dependencies between modules are automatically identified from queries. Queries are the only constraints that can retrieve information from other modules. As such, queries are responsible for introducing dependencies. Our incremental Statix solver automatically determines dependencies from queries by tracking the parts of the scope graph they visit. These locations are exactly the locations where something could be added which would affect the result of the query. This detailed location information is stored with the dependency, and can be used to prevent reanalysis whenever a dependency cannot actually be affected by a change.

- **Scope Graphs Comparisons**: We provide a technique for comparing the scope graphs of modules, which is necessary to determine the impact of a change. A big challenge was to do this efficiently, given that comparing graphs is generally an NP-complete problem [24]. However, the strict structure of scope graphs allows us to compare them in a much better way. The key point is that we need to know if a scope is still the same scope. We achieve this by using the trace of all the rules that led up to the creation of the scope in its identity. If the trace of a scope is the same between in two analyses, we know it is still the same scope. With scope identity, nodes in our graph are effectively fixed, and we can easily determine all the scopes, edges and data edges that are different. By combining these diffs with our dependencies, it is possible to determine all modules which are affected by a change.

- **Control over Solving**: We give extensive control over the solving process to the strategy. The strategy can limit interactions between modules to prevent usage of outdated information if necessary. In addition, solving can be split over multiple different phases, which is used by our optimistic name-based strategy.

Our model facilitates high-precision incremental analysis for Statix. We show its effectiveness with our optimistic name-based strategy implemented in our model, which is able to reduce analysis time significantly (Chapter 7).

Chapter 6

Strategies

Our model for incremental Statix gives us all the tools that we need to be able to do incremental analysis. A strategy then, is an algorithm which uses the different features of our incremental model to actually do incremental analysis. In this chapter we describe different families of strategies and show the advantages and disadvantages of each family. We describe our optimistic name-based incremental strategy which we have implemented with our model and provide proof sketches why it is sound.

Definition 6.0.1. *A **strategy** is an algorithm which uses the features of our incremental model to perform incremental analysis. That is, given the previous analysis result and a changeset of changed files in the project, a strategy has to determine the new analysis result.*

6.1 Optimistic Strategies

We distinguish two families of strategies, optimistic strategies and pessimistic strategies. The main difference is in the expected impact of changes. Optimistic strategies expect a small impact, while pessimistic strategies expect a large impact.

Definition 6.1.1. ***Optimistic strategy**: an optimistic strategy uses information from the previous analysis, even if that information is potentially outdated. It is based on the assumption that the changes made will not have a large impact on modules other than the changed modules.*

Definition 6.1.2. ***Pessimistic strategy**: a pessimistic strategy only uses information from a previous analysis which is known to be up to date. It is based on the assumption that the changes made will have a large impact on modules other than the changed modules.*

Optimistic strategies assume that the changes that were made are small and have a small impact, i.e. only a subsection of all modules needs to be recomputed. Optimistic strategies reanalyze all changed modules completely, even if such an analysis requires information from other modules which is potentially outdated. The effect of the changes can then be determined, and affected modules are reanalyzed. This cycle is repeated until no modules are affected.

The advantage of optimistic strategies is that they are less complex than pessimistic strategies. Pessimistic strategies have to determine which old information can safely be used and which information can not. They then have to try to work around the missing information to determine the new analysis result. If a new result cannot be reached from this limited set of information, they either need to resort to reanalysis of all involved modules, an optimistic fallback or a more complex strategy of doing slightly more work to make progress towards being able to determine this result (a finer granularity than module level).

The disadvantage of an optimistic strategy is that it is possible for an incremental analysis to be slower than a clean run. In particular, there are certain Statix specifications that lend themselves better towards optimistic strategies than others. Depending on the specification, an optimistic strategy might have to reanalyze the same module multiple times. This can occur if there are cyclic dependencies between modules, specifically dependencies between the external interfaces of modules.

6.1.1 Relevant Cyclic Dependencies

However, not all cyclic dependencies will actually cause a module to be reanalyzed multiple times. The external interfaces of the modules involved in the cycle must change depending on the items available in the other modules. An example of this is a language with conditional definitions. I.e. if module A defines some field x, then B defines some field y. If there are cycles in these conditions, A and B might need to be recomputed multiple times.

While these conditional declarations can be specified in Statix, the solver cannot always find a solution whenever there are cycles. Consider the example in Figure 6.1. In this case, there exists a valid solution consisting of $\{A.x, A.y, B.e, B.f\}$. However, the Statix solver instead gets stuck because of the overapproximation of critical edges.

```
class A {
    if (B.e is defined) then define x
    define y
}

class B {
    if (A.y is defined) then define e
    if (A.x is defined) then define f
}
```

$B.e \rightarrow A.x$
$A.y$
$A.y \rightarrow B.e$
$A.x \rightarrow B.f$

Figure 6.1: On the left an example of a program with field definitions that depend on the existence of other fields. On the right the corresponding logic formulae.

Whenever such a scenario does not initially exist, but one is created through some change, an optimistic strategy does not necessarily give the same result as a clean run. Consider the code in Figure 6.2, consisting of two separate modules A and B. First, the program on the left is analyzed in a clean run. The conclusion is that x, y, e and f are all defined. Now, a change is made to A as shown on the right side. An optimistic strategy would now first reanalyze A using the existing information. This analysis would conclude that still, x, y, e and f are defined. A clean run on the new program however, would get stuck. In this case, the optimistic strategy actually finds a solution where the solver with all the information does not.

However, less fortunate examples are also possible. Consider Table 6.3. It shows logic formulae corresponding to the definitions of fields in A and B. It shows the differences between the results of an incremental run compared to a clean run. In the first example, the optimistic strategy finds a solution for the cyclic definition. Depending on the language, this might be an unwanted solution, but it is nonetheless a valid solution for the constraint problem. In the second example however, the optimistic strategy diverges as it keeps reanalyzing first A and then B. This is because it determines that B is affected by the changes to the interface in A and vice versa.

As such, whenever specifications allow such conditional definitions, optimistic strategies can get into an infinite loop. It is possible to detect the scenarios in which this happens by linking the relevant queries to the changes they cause in the scope graph. From this we could detect that a cycle can occur, and analyze modules in the cycle together instead. In addition, it is possible to statically analyze the Statix specification to determine if there is

```
class A {
    define x
    define y
}

class B {
    if (A.y is defined) then define e
    if (A.x is defined) then define f
}
```

```
class A {
    if (B.e is defined) define x
    define y
}

class B {
    if (A.y is defined) then define e
    if (A.x is defined) then define f
}
```

Figure 6.2: The program on the left is changed into the program on the right. If the program on the right is analyzed with a clean run, the solver gets stuck, while an incremental run with an optimistic strategy would instead find the correct result.

Table 6.3: Difference in results between incremental and clean runs for optimistic strategies, written as logic formulae.

version	module A	module B	incremental result	clean result
original	$A.x$	$A.x \rightarrow B.y$	$\{A.x,\ B.y\}$	$\{A.x,\ B.y\}$
changed	$B.y \rightarrow A.x$	$A.x \rightarrow B.y$	$\{A.x,\ B.y\}$	*stuck*
original	$A.x$	$\neg A.x \rightarrow B.y$	$\{A.x\}$	$\{A.x\}$
changed	$B.y \rightarrow A.x$	$\neg A.x \rightarrow B.y$	*diverges*	*stuck*

even a possibility of cyclic definitions. We do not implement such a detection mechanism in our solution, and instead cut off reanalysis if the same modules are repeatedly analyzed for a set number of times.

Examples in Actual Languages In C, C# and C++ it is possible to create scenarios similar to the one presented in Figure 6.1 with preprocessor directives [16, 9]. With the preprocessor it is possible to include or exclude certain lines of code based on (global) meta variables. This is often used for including different definitions based on environment variables like the operating system or to include additional functionality if some optional library is also present. We estimate that the cyclic examples like in Figure 6.1 do not happen a lot in practice, due to the confusion that this can cause for programmers. In a language where this does occur a lot, switching to a pessimistic strategy might provide better results.

While Rust supports a technique called conditional compilation [29] which is similar to preprocessor directives, it is not possible to control the values used in the condition from the code directly. These can only be controlled from the configuration files or through environment variables passed to the compiler. As such, creating a scenario as in Figure 6.1 is not possible in Rust.

6.1.2 The Algorithm of Optimistic Strategies

All optimistic strategies follow the same general structure, which can be described with the sequence below. We give the corresponding algorithm in pseudocode in Listing 6.1.

1. Remove all the removed modules from the context
2. Compute an effective diff of the removed modules
3. Determine modules affected by the removal
4. Take the union of all changed modules and the modules affected by removal
5. If this set is empty, collect the result and return it. Otherwise, reanalyze all modules in the set together

6. Compute an effective diff of the reanalyzed modules
7. Determine affected modules from the diff and dependencies. Go to step 5

```
procedure OptimisticAnalyze($M_r$, $M_c$, $C$):
    // Remove all the removed modules from the context, determine affected modules
    $C'$ := removeModules($M_r$, $C$)
    $D$ := computeDiff($C$, $C'$, $M_r$)
    $M_a$ := determineAffected($D$, $C'$)

    // As long as modules are affected, continue to do analysis phases
    $M'$ := $M_c \cup M_a$
    while $(M' \neq \varnothing)$ do:
        $(C', M')$ := AnalysisPhase($C'$, $M'$)

    return gatherResult($C''$)

procedure AnalysisPhase($C$: Context, $M$: Modules):
    // Reanalyze the modules
    $C'$ := analyze($C$, $M$)

    // Compute a diff of the reanalyzed modules
    $D$ := computeDiff($C$, $C'$, $M$)

    // Determine affected modules
    $M_a$ := determineAffected($D$, $C$)

    return ($C'$, $M_a$)
```

Listing 6.1: Algorithm for incremental analysis with optimistic strategies

6.1.3 Correctness of Optimistic Strategies

We can only show the correctness of an optimistic strategy under the following assumptions:

1. Whenever the answer of a query in m' would be affected by the addition or removal of some edge e from scope s owned by module m, then there is a dependency of m' on m.
2. The diff that is created of the scope graph records all added and removed edges
3. All added and removed edges as determined by the diff are correctly combined with the dependencies to determine affected modules.

These assumptions will need to be proven separately for each strategy to make it sound. In addition, if there are relevant cyclic dependencies, an optimistic strategy need not give the same result as a clean run. Instead, it will either give a solution which is a possible valid answer to the constraint set, or it will diverge.

The Statix solver is tasked with finding an ordering in which to evaluate constraints. Before certain query constraints can be answered, constraints which make contributions to the scope graph relevant to the query need to be answered first. Whenever there are no relevant cyclic dependencies between modules, we know that an ordering exists and that it can be found by the Statix solver. The idea of our proof is to show that a new valid ordering will eventually be found by an optimistic strategy.

Proof Sketch. Let us consider a set of constraints C_1 with answer $\mathcal{G}_1$ and stuck constraints C_s. We now remove some subset of constraints C_r from C, and introduce some set of new constraints C_a. $\mathcal{G}_2$ is $\mathcal{G}_1$ with all contributions made in C_r removed. We try to satisfy the constraints C_a in the context of $\mathcal{G}_2$. We will find some answer $\mathcal{G}_3$ with some set of failed constraints C'_f and some set of stuck constraints C'_s.

All the stuck and failed constraints will not make their contributions to the scope graph, but dependencies will be recorded. We compare $\mathcal{G}_1$ with $\mathcal{G}_3$ and note all the added edges E_a and removed edges E_r. Any constraint in C or C_s which is affected by these changes will have some dependency $m' \hookrightarrow m'' \mid \langle V, R \rangle$ where $\exists (s, l) \in E_a \cup E_r. \, ((s, l) \in V)$ (by our assumptions on dependencies and diffs). As such, we will identify all modules m' as affected.

The strategy will now reanalyze all the affected modules. This process repeats itself, continuously altering the ordering of constraints until some valid ordering is found. Because of the absence of cyclic dependencies, we will eventually find some solution which is identical to the solution of the whole new constraint set. □

Pessimistic Strategies Pessimistic strategies are based on the assumption that the changes made will have a large impact on the project as a whole. Pessimistic strategies try to determine which information is still consistent and which information is not. They reanalyze modules partially, using only information that is known to be consistent and restore consistency by also analyzing other modules.

The main problem of pessimistic strategies is that it is quite difficult to determine which information is inconsistent without overapproximating. If we overapproximate however, we will possibly have to reanalyze a much larger set of modules than would be necessary. Essentially, this is because our modules do not offer enough granularity to reanalyze smaller chunks.

One technique to obtain this information would be to split each module in a static part and a dynamic part. Given that (at least large parts) of the interfaces of modules in many languages are static, it would be possible to quickly determine if modules need to be recomputed. We leave this to future work.

6.2 Optimistic Name-based Strategy

We have defined and implemented one strategy in our model: the Optimistic Name-based Strategy. As its name suggests, it is an optimistic strategy which uses names to refine dependencies. Examples of names are 'class String' or 'variable x', represented in Statix as 'Class{"String"}' and 'Var{"x"}'.

The strategy works as follows. Whenever a query queries for a specific name, i.e. 'Class{"String"}' or 'Var{"x"}', we record it with the dependencies it creates. Whenever we compute a diff of our program, we also determine when names are added to or removed from the scope graph. A dependency will only be considered if the name it looks for is introduced or removed. It is also possible for the information associated with a name to change, in which case the query is only considered if it requests that specific information and matches the name.

This technique significantly limits the dependencies that need to be considered. A strategy which does not use names would recompute a module A depending on a field x in B even if another field y is added in B, because all fields are added with the same relation. Our strategy avoids reanalysis in these scenarios.

However, Statix allows queries to use an arbitrary predicate to determine if a declaration in the scope graph matches. Whenever this predicate is more complex than a specific, individual name, our strategy does not associate any name with the query. These more complex queries will be considered every time an edge they query is changed.

```
procedure computeDiffWithNames($C$: Context, $C'$: Context, $M$: Modules):
    $(E_a, E_r, D_a, D_r)$ := computeDiff($C$, $C'$, $M$)
    $N_a$ := $\varnothing$
    $N_r$ := $\varnothing$
    for each data edge $s \xrightarrow{r} d \in D_a$:
        if $d$ describes a name:
            $N_a$ += $(s, r, name(d))$

    for each data edge $s \xrightarrow{r} d \in D_r$:
        if $d$ describes a name:
            $N_r$ += $(s, r, name(d))$

    return $(E_a, E_r, D_a, D_r, N_a, N_r)$
```

Listing 6.2: The definition of the diff algorithm for the Optimistic Name-based Strategy

6.2.1 Algorithm

The main procedure for the Optimistic Name-based Strategy is the same as the algorithm for any optimistic strategy. The difference is that it uses different functions for computing the diff and for determining affected modules from dependencies, which are described in Listing 6.2 and Listing 6.3. In addition, the mechanism for recording dependencies from queries is extended to also include a name if the predicate matching data of the query describes a name. We detect such predicates by checking if the predicate consists of a single equality constraint containing an occurrence of the form `X{y}` where `y` is a bound variable.

The diff mechanism is extended to include information about names, as presented in Listing 6.2. Whenever a data edge appears in the added data regards a particular name, it is stored in a separate set.

Listing 6.3 shows the algorithm for determining affected names from diffs extended with names. The algorithm is very similar to Listing 5.3, but with one small change and one addition. First, dependencies which have additional name information are ignored for additions of data edges, since these will be handled by our mechanism for names instead. However, for edge removals, our algorithm already determines if a query is affected with optimal precision, as only edges actually traversed by the query to reach its answer are in R. As such, names only need to be checked for additions.

Possible Improvements Our strategy could be further improved if the meaning of more complex predicates can also be determined. For example, a query for all classes only needs to be considered if a class changes. However, the predicates describing this are much more complex, and multiple ways to write such a predicate exist. Instead, it would be useful if predicates are normalized to simpler forms to ease the static detection of such predicates. In addition, there are cases in languages where a query needs to look for two names at once. For example, in Java, a reference to a variable in a method can refer to either a field or a variable. Currently, we address this problem by recording fields and variables in the same namespace, but it would be nice if queries requesting multiple specific names could also be identified.

6.2.2 Correctness

Our approach for modularization and incrementality guarantees that if any information used for these checks changes, that the checks are reevaluated. Our collection of dependencies for queries guarantees that we record all the places where information could be added which

```
procedure determineAffectedNames($D$: NameDiff, $C$: Context):
    $Deps$ := getDependencies($C$)
    $(E_a, E_r, D_a, D_r, N_a, N_r)$ := $D$

    $M$ := $\varnothing$
    // Compute dependencies affected by edge additions, ignoring names for data edges
    for each $(s, l) \in E_a$:
        for each dependency $d \in Deps(s, l)$:
            $M$ += dependant($d$)
    for each $(s, r) \in D_a$:
        for each dependency $d \in Deps(s, r)$:
            if $d$ does not describe a name:
                $M$ += dependant($d$)

    // Compute dependencies affected by edge removals
    for each $(s, l) \in E_r$:
        for each dependency $d \in Deps(s, l)$:
            if $(s, l) \in d.R$:
                $M$ += dependant($d$)
    for each $(s, l) \in D_r$:
        for each dependency $d \in Deps(s, r)$:
            if $(s, r) \in d.R$:
                $M$ += dependant($d$)

    // Compute dependencies for data edge additions regarding names
    for each $(s, r, n) \in N_a$:
        for each dependency $d \in Deps(s, r)$:
            if $d$ describes a name $n'$ and $n = n'$:
                $M$ += dependant($d$)

    return $M$
```

Listing 6.3: The definition of the algorithm for determining affected modules from a diff and dependencies for the Optimistic Name-based Strategy

would affect our result. If any new information is added which could affect the result, the module encompassing the query is reanalyzed. If the original query gets stuck (there is not enough information to answer it), we still record the paths that would be more specific, which would prevent it from getting stuck. In addition, we also consider the edge on which the query is stuck, since that could also affect the dependencies.

The proof sketch that we give in Section 6.1.3 also applies to our name based strategy. The modifications we make to the diff and the dependency detection only apply if a particular name is included in the query, and only for additions of data edges. Additions of data edges which do not mention names are treated as normal, additions which do mention a name are only considered if that name matches the name of the dependency. It is trivial to see that a query for some name n will not be affected by a change to some name n' where $n \neq n'$. As such, we conclude that our optimistic name-based strategy is sound.

Chapter 7

Evaluation

In this chapter we evaluate the performance of our optimistic name-based strategy. First, we evaluate the baseline performance by comparing clean runs in the original solver with clean runs with our incremental solver. Then, we compare incremental performance with clean run performance on a variety of randomly applied changes.

7.1 Setup

All benchmarks are run on Ubuntu 19.04 with an Intel Core i7-4700MQ CPU @ 2.40GHz (4 cores, 8 threads) and 16 GB of RAM, with Spoofax 2.5.6 on OpenJDK Java 1.8.0_212, 64-bit. We allocate 13.5 GB of RAM to Java for the baseline performance measurements, and 8 GB of RAM for the tests for incremental performance.

We evaluate the performance on differently sized Java projects. We selected Java for multiple different reasons. First of all, the syntax specification for Java is already fully available in Spoofax. While there are syntax specifications for other languages in Spoofax, these are either not complete, or are for toy languages for which no substantial code bases exist. Second, Java has interesting name resolution in the form of type-dependent name resolution (see Listing 7.1), giving rise to a lot of dependencies between classes. Finally, there are many open-source projects available for Java which we can use to evaluate our performance, allowing us to evaluate the impact of the size of a project on incremental performance.

Statix Specification for Java We use a simplified Statix specification for Java which focusses only on type-dependent name resolution. Examples of what we mean by type-dependent name resolution are shown in Listing 7.1. Our specification does not do type checking, but does use types to do name resolutions. Type checking itself is not interesting for our incremental performance, since all relevant accesses to the interfaces of other files happen through name resolution.

At the file level, the specification includes support for packages, imports and all class-like structures (classes, interfaces and enums). Inside of classes and interfaces, we support methods and fields. At the statement level, we support blocks, declarations and expression statements.

We use a desugaring on our parsed Java files to remove the functionality from statements, only retaining blocks, declarations and expression statements. We retain the scoping structure inside of methods by converting all block-like statements, e.g. `if`, `for`, `while`, `switch` and `try`, to corresponding blocks, variable declarations and expression statements. Additionally, we retain expression statements only it they do some form of type-dependent name resolution. This leaves us with an AST with only blocks, declarations and expressions which are interesting for incrementality.

We include the entire specification in Appendix B.

```
public class A {
    public static class B {
        public static B b;
        public static int x;
    }

    public B m() {
        Class c;
        int x;
        c = A.class;
        c = A.B.class;
        x = 10;
        x = A.B.x;
        x = B.x;
        x = B.b.b.x;
        x = new A().m().b.x;
    }
}
```

Listing 7.1: An example showing different type-dependent name resolutions in Java of various complexity. All of these examples in the method m are supported by our Statix specification for Java.

Reproducability of Benchmarks All the scripts and code required to repeat these benchmarks are available at `https://github.com/MetaBorgCube/statix-incremental-artifact`. While the changes we apply to projects are random, they are seeded random, which makes the results reproducible. For these benchmarks, we always use a seed of 1.

7.2 Baseline Performance

We evaluate the baseline performance of our approach by comparing clean runs in the original solver with clean runs when using our Optimistic Name-based Strategy. We evaluate this performance on an analysis of a reduced, stripped version of the source code for the Java Development Kit (JDK), version 8. We can only run an effective comparison with the original Statix solver for the JDK itself, since it does not support libraries. Fully analyzing a project in combination to the JDK is too large a task for (either version of) Statix to handle, requiring more RAM than we have available on our system. However, we believe that given the size of the JDK, it still offers a good comparison.

Reduced JDK First, we create a stripped and reduced version of the source code of the JDK. We remove the bodies of all methods since we are only interested in the interfaces of the files. Then, we remove all the classes except for those in `java.io`, `java.lang`, `java.math`, `java.net`, `java.nio` and `java.util`, which are the most used parts of the JDK. We follow by running our reduction script, to replace all references to the removed files with `Object` or `Serializable` for classes and interfaces respectively. Both of these reside in the `java.lang` package, and are always available. This way, we prevent errors in the signatures of methods as a result of the missing parts of the JDK. The reduced JDK still contains 879 classes, which we can only barely handle with our simplified Statix specification for Java.

We use the aforementioned system and Java version, and allocate 13.5 GB of RAM to Java. Any lower amount of RAM either causes significantly increase analysis times because of aggressive garbage collection, or analysis does not complete due to insufficient memory.

7.2.1 Single Threaded Performance

Figure 7.1 shows the absolute clean analysis times of the original solver and our incremental solver for different concurrency levels. We show two modules granularities, class and file level. In Figure 7.2, we show the relative analysis time of our incremental solver compared to the original solver. Positive values indicate a slower analysis time than the original solver, while negative values indicate a faster analysis time. Both figures use the average analysis time as measured over 10 different runs.

From Figure 7.2, we see that the incremental solver performs about 5 to 10% worse than the original solver, depending on the module granularity. This is in line with what we would expect given that the incremental solver needs to keep track of dependencies. In addition, there is some overhead in the split solving and the continuous redirections in our components.

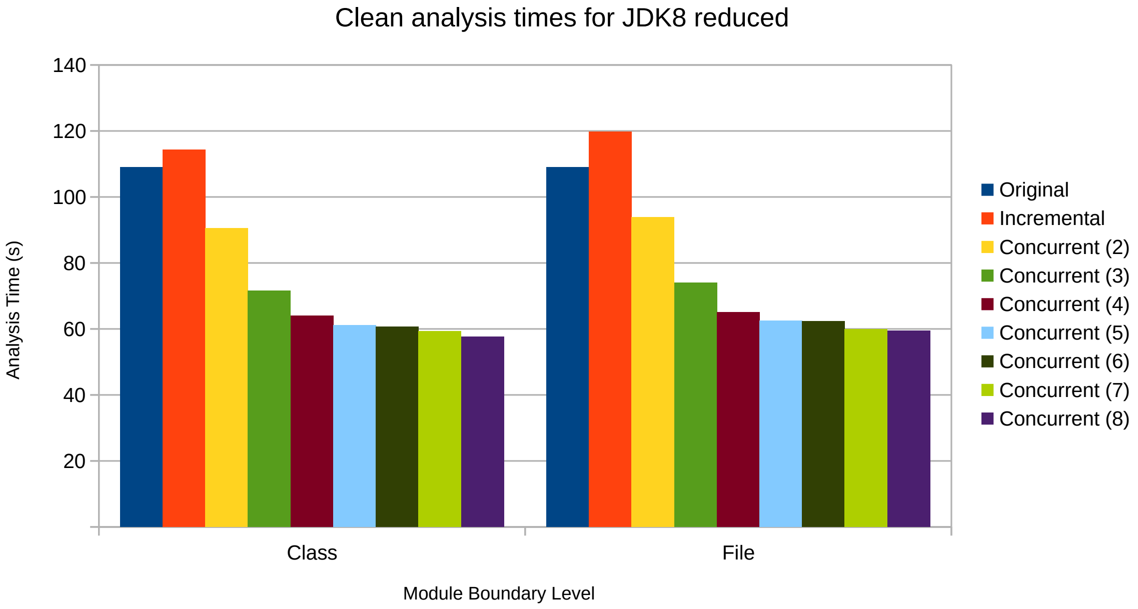


Figure 7.1: The performance of the incremental solver for clean runs compared to the original solver. Total analysis time for a reduced and stripped version of the source code for the Java Development Kit 8. On the y-axis, the absolute analysis time is plotted in seconds. On the x-axis, the module boundary level. We see that while the incremental solver is slower than the original solver, running it concurrently negates this completely, even achieving marginal speedup.

7.2.2 Concurrency

In Figure 7.2, we show the relative analysis time of our incremental solver compared to the original solver. Positive values indicate a slower analysis time than the original solver, while negative values indicate a faster analysis time. In terms of concurrency, we see that we achieve quite significant speedup, even when there are only two cores available. With two threads, we already achieve a 10 to 15 percent reduction in analysis time, improving further to a 30 percent reduction with three threads and a 40 percent reduction with four threads. With eight threads, we almost halve the solving time compared to the original solver. In fact,

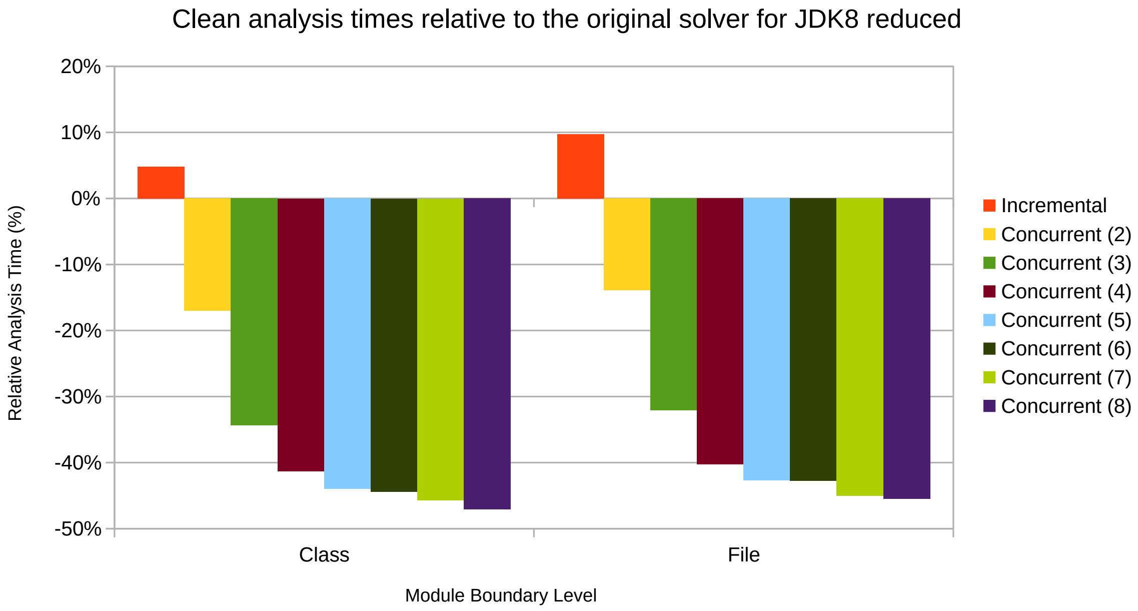


Figure 7.2: The performance of the incremental solver for clean runs relative to the original solver. Each bar represents the relative time increase (positive) or decrease (negative) compared to the original solver. For example, the positive value of 5% for the incremental solver at class level means that it is 5% slower than the original solver at that granularity. We see that the non-concurrent version is 5-10% slower than the original solver. The concurrent version achieves significant speedups compared to the original version, depending on the amount of threads.

we halve the solving time of our single threaded incremental analysis when running with eight threads.

We also see that the module granularity does not influence the impact of concurrency much, with performance gains about equal. However, we see some interesting results when going from file granularity to class granularity, with analysis time actually reducing when we split classes from files. This is due to the improved constraint ordering that occurs as a result.

We do see that we get diminishing returns from increasing the amount of threads, especially after four threads. This can be attributed to Amdahls Law, which states that any parallel process which cannot be fully parallelized will have diminishing returns from an increased amount of cores. If we enter our results into the formula for Amdahls Law, we see that our system has about 40% single threaded execution [44]. We clearly see these diminishing returns across all of our steps, where we see that the benefit of each additional thread is smaller than the previous addition.

7.2.3 Impact of Module Granularity

We evaluated the JDK on two module granularities, classes and files. We were unable to evaluate the impact of smaller module granularities, i.e. methods, because we did not have not enough RAM to complete this analysis. The additional overhead of all the extra modules that method boundaries create on a large project like the JDK is too much for our system to handle.

Interestingly, we see that class boundaries actually perform slightly better than file boundaries. We believe that this is due to an improved ordering in which constraints are evaluated caused by separating classes from the packages, which reduces the amount of delays required. The way that packages are modelled in our Statix specification means that each file contributing to the package must first contribute their package scope before anyone can do type resolution. By splitting class modules from file modules, the solver will first complete all the packages from the file modules and will only then start evaluating all the class modules. This means that type resolutions are not as likely to be delayed, improving the performance.

7.2.4 Storage Costs

In terms of storage costs, we see a significant increase compared to the original solver as presented in Table 7.3. Almost all of this additional storage cost is attributed to the dependencies, which we can see when we convert a context into a library. The library removes the ASTs and the dependencies from the context and uses the external scope graph of each module, explaining the reduction in size compared to the original solver.

Table 7.3: The storage costs for storing the analysis result of the reduced JDK8 for the different solvers. On the left the original solver, in the middle the incremental solver with the optimistic name-based strategy and on the right when stored as a library.

Project	Original	Incremental	Library
JDK 8 Reduced	241 MiB	1320 MiB	178 MiB

7.2.5 Conclusion

We answer **Research Question 3: How can we apply concurrency to Statix?** We support concurrency in our model for incremental analysis by solving modules in parallel. We provide this functionality with minimal changes to the design by leveraging existing safety guarantees of Statix. Our concurrent implementation achieves significant speedups of over 40% compared to the original solver, almost cutting analysis times in half.

7.3 Incremental Performance

We evaluate the incremental performance of our strategy by applying different types of changes to differently sized project. For each change, we do both an incremental run and a clean run on the changed project. We compare the performance between the incremental and non incremental runs and evaluate how much of the project is actually reanalyzed.

We cannot compare the incremental performance to the performance of the original solver directly, since it does not offer support for libraries. Analyzing projects without the JDK as a library would not be a fair comparison, since a lot of the constraints would fail and prevent others from being evaluated. Analyzing the JDK in combination with differently sized projects would not allow us to measure how the size of a project influences the incremental performance. As such, we compare the performance of clean runs to the performance of incremental runs both with our optimistic name-based strategy.

If one does want to estimate how to compare this with the original solver, it is possible to get a rough estimate from the fact that the incremental solver is about 5 to 10 percent slower, as shown in the baseline performance.

7.3.1 Types of Changes

For benchmarking the incremental changes, we follow a similar procedure to Szabó et al. [40]. We randomly introduce small changes into the files to emulate a user of an IDE making changes to the project. We look at small changes because analysis can run in the background every time the file becomes parseable while the user is typing, keeping the changes between versions small. The changes we consider are listed in Figure 7.4. We have chosen these changes specifically based on the granularity of the analysis and because they have impact on different scopings. The changes that affect classes are common changes to the global structure of the project. The changes that affect methods will have a limited impact on the class itself and some impact on extending/calling classes. Then, the changes to statements represent the common scenario where a programmer is changing statements inside a method, the effect of which is rarely visible outside the method.

- Remove a class
- Remove a method
- Add a method
- Remove a statement
- Duplicate a statement

Figure 7.4: The different categories of changes that we evaluate.

We apply changes as follows. First, a file is selected randomly from all the available files. We then try to apply our change to that file. For the removal of a file and the addition of a method, this always succeeds. For the removal of a method, this will only succeed if there is a method in the file. If such an application fails, we remove the file from the options and randomly select a new file until we find one where the change can be applied.

For the removal and duplication of statements, we randomly select a method from all the methods that have at least one statement. We then randomly select a statement from the body and remove it or duplicate it. If the application of our removal or duplication somehow still fails (i.e. the statement in the method is an empty statement), we try again in the same file 15 more times. If that fails, we move on to a different file in the project, until an application succeeds.

For each category, we measure the analysis time of 100 changes, each starting from a clean context. The changes that are applied are randomly determined, but are seeded. This means that with a given seed, the same set of changes will be evaluated. In our evaluation, we always use a starting seed of 1.

Projects For our evaluation, we use differently sized open source projects written in Java. We show the projects we use for evaluation and the number of classes in Table 7.5. We evaluate these projects with the reduced version of JDK8 loaded as a library.

Table 7.5: The different projects used for our incremental analysis with their number of files. Both projects are open source and are available for download at their respective sites [41, 42]

Project	Files
Apache Commons CSV	13
Apache Commons IO	119

Setup We use the same setup as before, but we can now reduce the amount of memory allocated to Java to 8 GB. We note that it is possible to analyze these projects with any amount of RAM larger than 6 GB, but that the aggressive additional garbage collecting that sometimes has to happen as a result can significantly lengthen the analysis time.

7.3.2 Reduced Analysis Time and Reanalyzed Files

Figure 7.6 and Figure 7.7 show a comparison of the average analysis time between clean and incremental runs for different changes. First of all, we see that bigger changes such as removing entire files, have a significantly larger impact on reanalysis time than smaller changes. This is in line with what we would expect from a larger change.

However, we also see that there is little difference in analysis time between removing methods and duplicating/removing statements. We would expect that the removal of a method has more impact than changing statements inside a method. If we look at the number of files that are redone per change, as presented in Figure 7.8 and Figure 7.9, we see that the number of reanalyzed files for removing methods is on average slightly above one. Through further investigation, we have found why this is the case.

For Commons IO, the low number of reanalyzed modules due to the removal of methods is mainly because it is a library where almost all classes implement some interface. All the other implementation then is against the interfaces, not the concrete implementations. This means that there are little to no dependencies on the methods of concrete classes. Given that there is a significantly higher number of classes than interfaces, the methods removed are skewed towards removals in classes rather than in the interfaces. In addition, the classes with an actual concrete implementation often have private helper methods which can only be used within the file. The removal of such methods will thus never cause reanalysis of other files.

For Commons CSV, there are only 13 files in the first place, only 10 of which actual contain methods. Each method is usually only used by one or two classes. Since Commons CSV is also a library, there are quite a few methods that are never used, i.e. a large part of the public API. This explains the low number of reanalyzed files that we see when methods are removed.

In addition, we notice that sometimes, the removal or addition of a statement triggers reanalysis of additional files. If a change introduces errors, there can be cascading failures in Statix. Especially constraints that get stuck can cause a lot of variables to remain free, in turn making other constraints get stuck as well. The duplication and removal of some statements can cause this effect as well. Because of the stuck constraints, larger parts of the scope graph disappear or change to contain variables instead of values. This in turn leads to our incremental solution having to reanalyze more files. However, we see that this happens in a minority of cases.

In conclusion, we see that our strategy skips reanalysis of unaffected modules as we would expect. There are cases where the number of reanalyzed modules is higher due to constraints getting stuck. However, we see that the average number of reanalyzed modules is still very low whenever the impact of changes is small. As such, we believe our incremental approach is effective.

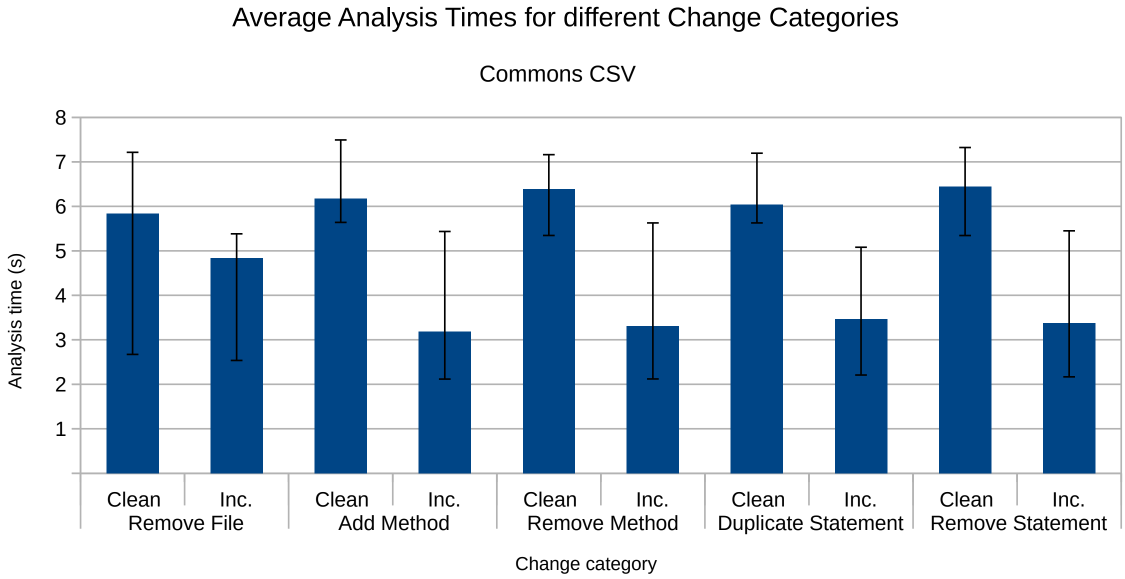


Figure 7.6: A comparison of the average analysis time per change category for the Commons CSV project (small). Each bar represents the average analysis time of 100 changes of the corresponding category. The error bars show the minimum and the maximum analysis time.

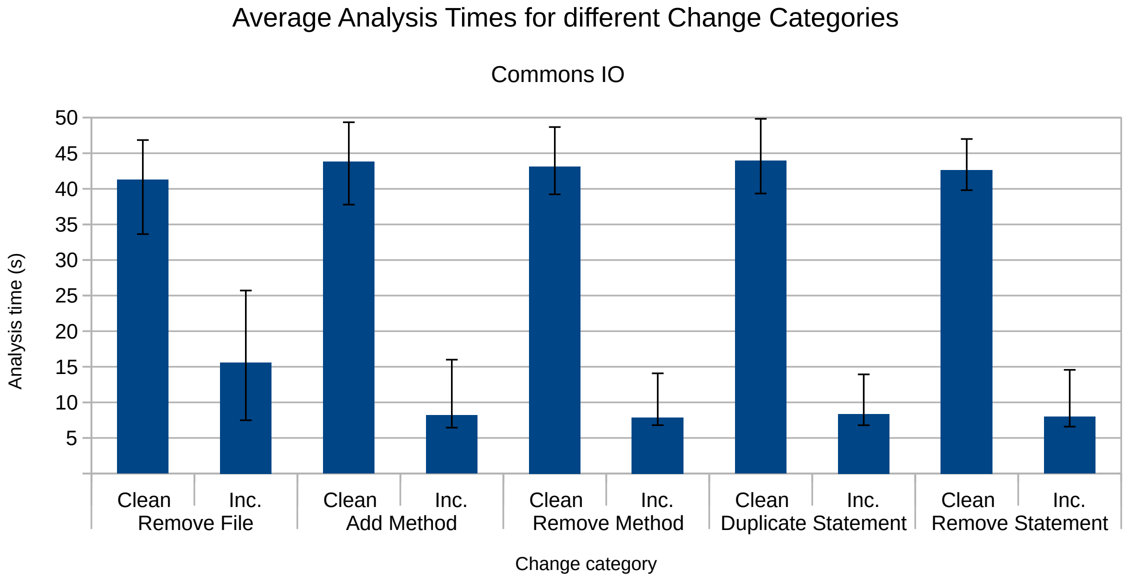


Figure 7.7: A comparison of the average analysis time per change category for the Commons IO project (large). Each bar represents the average analysis time of 100 changes of the corresponding category. The error bars show the minimum and the maximum analysis time.

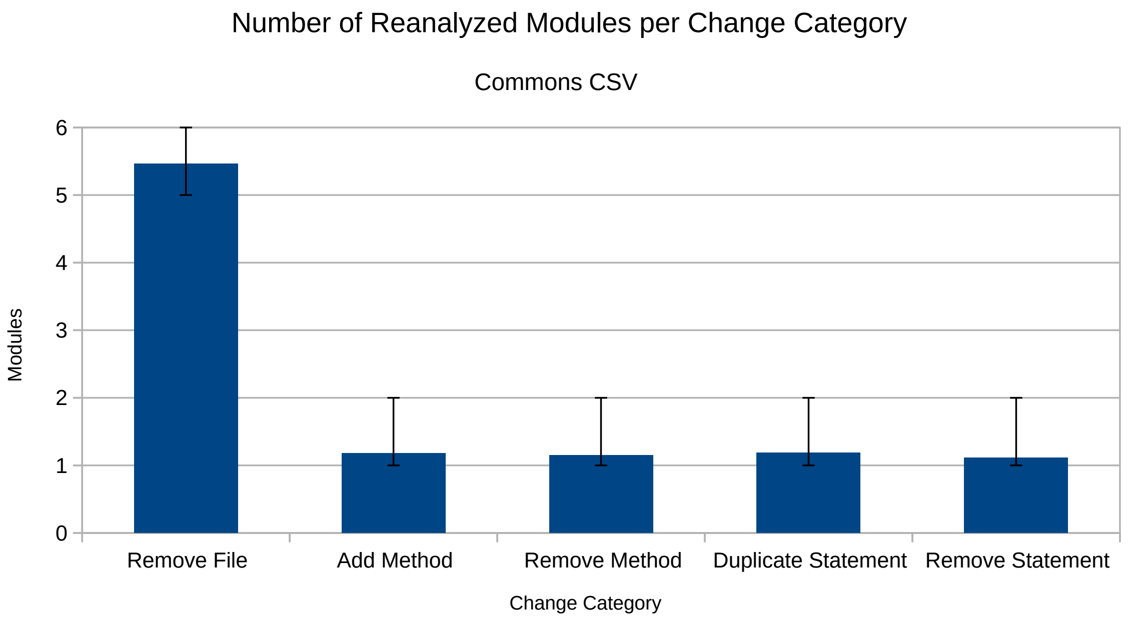


Figure 7.8: The average number of reanalyzed modules per change category for the Commons CSV project (small). Each bar shows the average as measured over 100 changes in the corresponding category. The error bars show the minimum and maximum reanalysis counts.

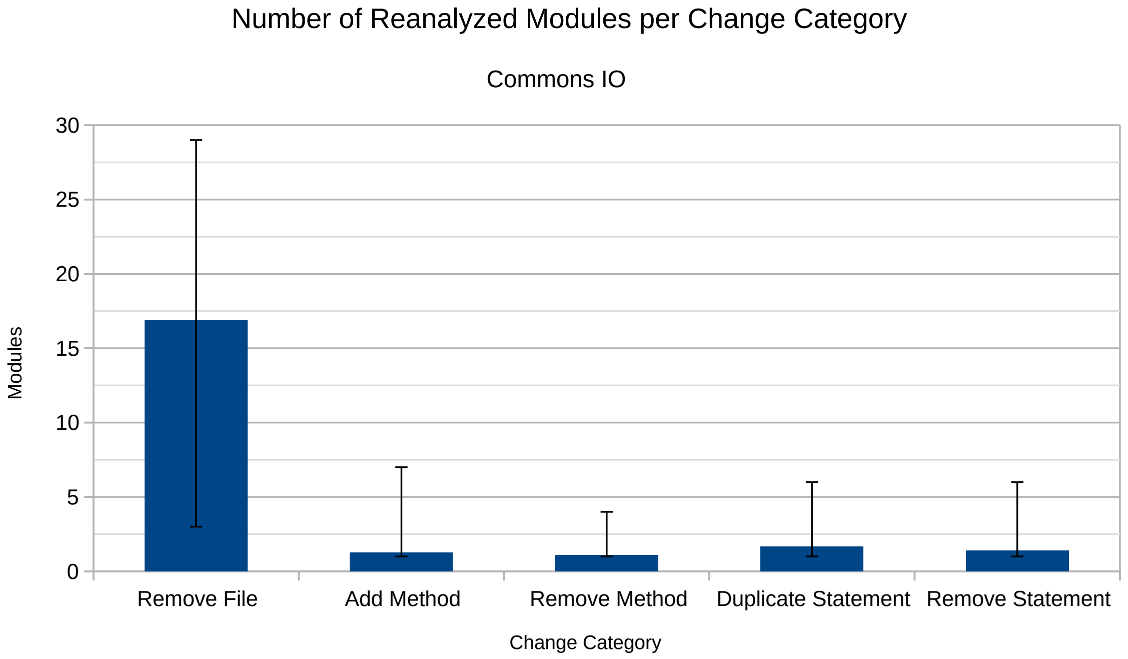


Figure 7.9: The average number of reanalyzed modules per change category for the Commons IO project (large). Each bar shows the average as measured over 100 changes in the corresponding category. The error bars show the minimum and maximum reanalysis counts.

7.3.3 Worst Case Performance

From Figure 7.6 and Figure 7.7 we also see that on a smaller project, the incremental performance is much closer to the clean analysis time than for a large project. This is also what we would expect, given that even if only a single file needs to be redone is already quite significant for the project. In addition, files in Commons CSV are much more coupled than files in Commons IO, leading to larger effects from changes such as the removal of files as well.

One of the properties that we want to achieve is that the incremental analysis is never significantly slower than a clean analysis would be. From Figure 7.6 and Figure 7.7 we see that for both our small and our large project, the *maximum* analysis time of incremental runs is always smaller than the *average* analysis time of clean runs. However, we should also look at each individual change to see how times compare there.

Figure 7.10 shows a comparison of analysis time grouped by file size after a file has been removed. First of all, it is clear that there is one file in the project which contains a majority of the code, i.e. the largest file. We also see that the incremental analysis time is at worst about equal to the clean analysis time.

In Figure 7.11 we show a comparison of the analysis time for the duplication of statements. We see more instances where incremental analysis time is a bit longer than clean analysis time. At worst, incremental performance is within 10% of the clean run performance, which occurs for about 10% of our measurements. We find these numbers to be relatively high given that we only make a very small change. Unfortunately, we were unable to identify exactly why these cases perform badly. From the figure we can see that these cases still only reanalyze a single file, i.e. the changed file, as expected. The time spent on computing scope graph differences and on dependency checking is only a few milliseconds for all cases and it is not different here. Instead, something in the solving process itself seems to just be quite a bit slower for these cases.

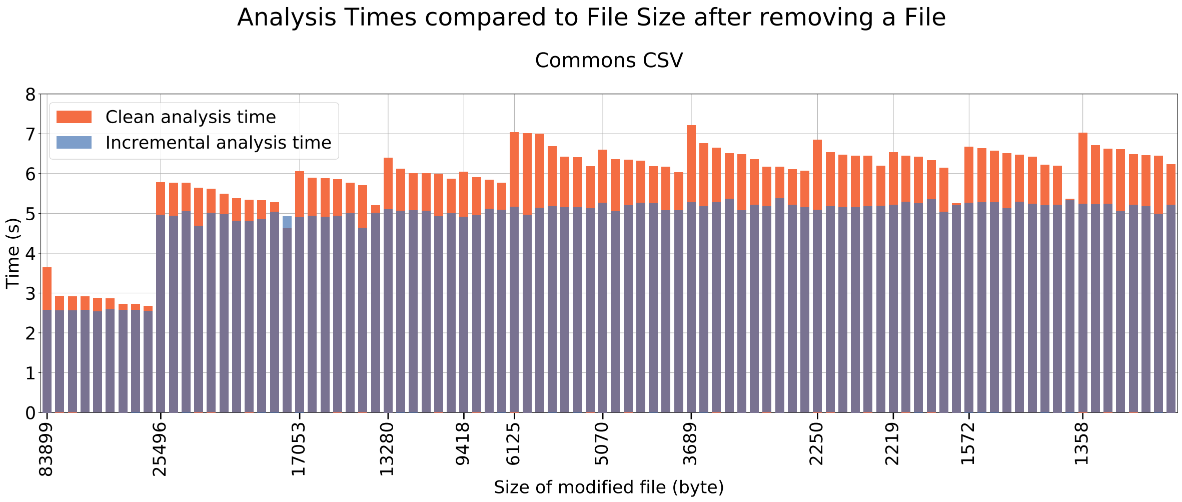


Figure 7.10: Comparison of clean analysis time (orange) and incremental analysis time (blue) for 100 file removals for Commons CSV (small). On the x-axis the size of the removed file and on the y-axis the analysis time.

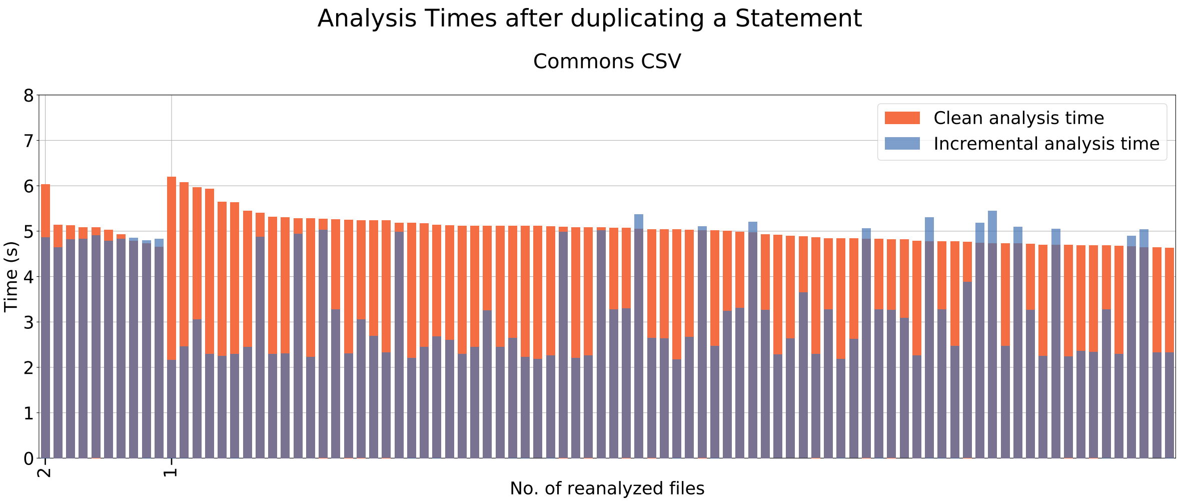


Figure 7.11: Comparison of clean analysis time (orange) and incremental analysis time (blue) for 100 statement duplications for Commons CSV (small). On the x-axis the number of reanalyzed files and on the y-axis the analysis time.

Of course, the small changes that we evaluated are not the worst case for our strategy. Java does not have conditional declarations (Section 6.1.1), i.e. declarations which are only there whenever other modules have some declarations. The worst-case we can achieve with our Statix specification for Java consists of only two analysis phases, where in the first phase one module is reanalyzed and in the second phase all remaining modules are reanalyzed. There is no way for changes to propagate transitively further than one step because Java is strictly typed. If we look at the number of phases for each type of change, we see exactly this behavior. All analyses consist of at most 2 phase, i.e. first analyzing the changed module and then optionally doing one additional phase.

So, on average, our incremental strategy is noticeably faster for small projects (up to two times) and significantly faster for large projects (up to five times). We see that the number of reanalyzed files is small when changes are small. However, on small projects there are cases where our strategy is slower than a clean run, even when changes have a small impact. Given that the worst-case scenario for our strategy cannot occur in Java, we expect that there are languages where this effect will be worse. An analysis on languages other than Java is required to determine if this impact is indeed larger for languages with more complex module systems.

Impact of Failures Stuck constraints can affect the number of files that need to be reanalyzed. If there are many constraints that get stuck, our model sometimes has to overapproximate the impact of changes and reanalyze additional modules for safety. This is also what we see for some duplications and removals of statements. The failures these introduce cause an overapproximation of the impact of changes, triggering reanalysis of more modules than necessary. However, this is a side effect of Statix and the way our Statix specification for Java is written. We believe that our specification could be adjusted to avoid these cascading failures to achieve better incremental results.

7.3.4 Conclusion

We have manually checked the equivalence of our incremental runs with clean runs and found no cases where they do not align. This, combined with our proof sketch for the correctness of optimistic strategies as provided in Section 6.1.3 lets us believe that our optimistic name-based strategy is correct.

We now answer **Research Question 4: How well is our optimistic name-based strategy suited for incremental analysis?** Our optimistic name-based strategy is both correct (Section 6.1.3) and effective. Our results show that it achieves significant speedups whenever small changes are made to a project. Even on small projects and with changes with a large impact, our strategy does not perform significantly worse than a clean analysis, with analysis times staying within 10% of the clean analysis times.

Chapter 8

Conclusion and Future Work

IDEs support developers by giving real-time feedback on their code. This feedback is generated from static analyses such as type checking, name binding and flow analysis. However, creating such static analyses for domain specific languages and integrating them into IDEs is a challenging and time consuming task. Statix addresses this problem by generating a type checker from a declarative specification of typing and name binding rules. This generated type checker is integrated into an IDE directly, to allow for fast language development. However, Statix is not fast enough to give such feedback in real-time. If a project grows to a size of more than a couple of files, even small changes can require a lengthy reanalysis. In this thesis, we aim to address this problem by answering the question *"How we can improve the analysis time of scope graph based type checkers such as Statix?"*

Existing Techniques in Literature First, we looked at literature to find existing techniques used by type checkers, compilers, IDEs and build systems to improve analysis time, to determine if and how we could apply them to Statix. A common trend throughout many of these techniques is the use of incrementality. An incremental analysis uses the result of the previous analysis and a set of changes, to reanalyze only the changed units and those units depending on them. If the effect of the changes is small, the amount of work required is also small. From the literature, we have come to the following conclusions:

- Separate compilation is a technique of compiling program fragments separate from each other, using only limited information about other available program fragments [7]. It has been applied to compilers for many different languages to achieve incremental compilation [10, 6, 26, 4, 28]. We found that the techniques used for separate compilation can be applied to Statix. The separate program fragments of the source language can be mapped to modules in Statix. We can then analyze these modules separately from each other in the context of the interfaces of other modules, with the scope graph as a representation of the interface of a module.

- Incremental rebuilding of attribute grammar trees is another example where incrementality is used. Attribute grammars associate attributes to nodes in the abstract syntax tree of a program, allowing the definition of various static analyses. Whenever parts of the tree change, it is possible to invalidate only some of the attributes and recompute them, without having to recompute them for the entire tree [37, 36, 23, 31]. However, extending this to Statix would be difficult, since the ordering of constraints is complicated and not as clearly defined as for attributes.

- Hedin et al. [22, 39, 21] extend attribute grammars with support for objects and references. They effectively build a graph on top of the syntax tree, describing the scoping and name resolution. These reference attribute grammars require a different incremental approach, because dependencies are dynamic and are not known before hand. They present a technique for recording these dynamic dependencies automatically as their graph is built. The graph they build on the syntax tree bears a lot of similarities to the scope graphs used by Statix. Their way of tracking dynamic dependencies can be used in Statix as well, given that queries are similar to their name resolution rules.

- There are type checkers which take a completely different approach, gaining incrementality directly from their design [14]. These techniques cannot be applied to Statix without significantly changing the architecture, which we want to avoid.

We use separate compilation as the basis of our incremental model, and use the ideas of Hedin et al. for detailed tracking of dynamic dependencies between modules. With these techniques, we can reanalyze individual modules and propagate the changes to only those modules that depend on the changes. Because modules which are unaffected are skipped, this leads to a reduction of analysis time.

Building an Incremental Model for Statix From literature we have concluded that the techniques of separate compilation and dynamic dependency detection can be used to achieve incrementality for Statix. We contribute an incremental model for Statix based on these techniques. The goal of our model is to facilitate different strategies for incremental analysis, to fit the needs of different source languages. We achieve this goal through the following features of our model.

- **Modules**: We add support for modules, which correspond to modules in the language we are analyzing. Rather than constructing a complete scope graph for the project, we split the scope graph over the different modules. The interface of each module is its part of the scope graph, which contains all information that can be used by other modules. A main challenge in the implementation of our model was the splitting of the solving process over these modules, an approach very different from the original monolithic architecture of Statix. We stayed close to the original architecture by having each component redirect requests to the module which can answer that request. As such, we still provide a view of the system as a whole, without incurring a significant extra cost.

- **Dependencies**: Dependencies between modules are automatically identified from queries. Queries are the only constraints that can retrieve information from other modules. As such, queries are responsible for introducing dependencies. Our incremental Statix solver automatically determines dependencies from queries by tracking the parts of the scope graph they visit. These locations are exactly the locations where something could be added which would affect the result of the query. This detailed location information is stored with the dependency, and can be used to prevent reanalysis whenever a dependency cannot actually be affected by a change.

- **Scope Graphs Comparisons**: We provide a technique for comparing the scope graphs of modules, which is necessary to determine the impact of a change. A big challenge was to do this efficiently, given that comparing graphs is generally an NP-complete problem [24]. However, the strict structure of scope graphs allows us to compare them in a much better way. The key point is that we need to know if a scope is still the same scope. We achieve this by using the trace of all the rules that led up to the creation of the scope in its identity. If the trace of a scope is the same between in two analyses, we know it is still the same scope. With scope identity, nodes in our graph are effectively fixed, and we can easily determine all the scopes, edges and data edges that are different. By combining these diffs with our dependencies, it is possible to determine all modules which are affected by a change.

- **Control over Solving**: We give extensive control over the solving process to the strategy. The strategy can limit interactions between modules to prevent usage of outdated information if necessary. In addition, solving can be split over multiple different phases, which is used by our optimistic name-based strategy.

Our model facilitates high-precision incremental analysis for Statix. We show its effectiveness with our optimistic name-based strategy implemented in our model, which is able to reduce analysis time significantly.

Adding Concurrency to our Model The incremental model we created facilitates incremental analysis based on modules. We split the solving process over the different modules and added support for solving these modules in parallel. We provide this functionality with minimal changes to the design by leveraging existing safety guarantees of Statix. Our concurrent implementation achieves significant speedups of over 40% compared to the original solver, almost cutting analysis times in half.

The Effectiveness of our Model With support for concurrency in our model, we finished by implementing our optimistic name-based strategy with our model. Our results show that it is effective and achieves significant speedups whenever small changes are made to a project. Even on small projects and with changes with a large impact, our strategy does not perform significantly worse than a clean analysis.

Improving Analysis Time Our model for incremental analysis is based on separate compilation and automatic detection of dynamic dependencies. Our optimistic name-based strategy uses our model to achieve parallel and incremental analysis for Statix. We show that it is correct and effective, significantly reducing analysis time in good cases and not increasing analysis time in bad cases. We believe that our model is also suited for other kinds of strategies, through the various functionalities it offers. With the extensive control our model offers over the solving process, we believe that pessimistic strategies, which only use up-to-date information to determine affected modules, can also be expressed effectively in our model.

Our optimistic name-based strategy is effective on small and on large projects. While we only analyzed its effectiveness for Java, we predict it is also effective for other languages with different modules. We predict this holds even when the module system of the language does not support full separate compilation, such as the module systems of Standard ML and Haskell.

8.1 Future Work

IDE Integration Our approach yields significant speedups, especially when the impact of changes is small. We believe it to be a good fit for an IDE, where the analysis can be kept up to date by continuously running in the background. However, even with our changes, Statix still needs to be made faster to achieve sub-second analysis times. One of the main limiting factors is the fact that we always reanalyze the entire file in which the change occurs. While it is already possible to use chunks smaller than files in our model, there needs to be better integration between Statix and the IDE to achieve this.

Splitting Modules In many languages, most, if not all, parts of the interface of a module are not actually dependent on other modules. We believe it would be possible to split each module into two parts. One part which represents everything of the module that is fixed, and one part which represents everything of the module which depends on other modules. This would achieve multiple benefits. First, this would directly give incremental analysis time reductions. The fixed part of a module only has to be reanalyzed if the module is directly changed. For all other modules, only the dynamic part possibly needs to be reanalyzed. Second, it would pave the way for pessimistic strategies. Currently, pessimistic strategies have to avoid all information from other modules until they know that it is still up to date. With certain parts of the scope graph fixed and known to be up-to-date, this would become easier.

Some of the challenges we encountered is the fact that cross-module unification occurs between the fixed module and the dynamic part of the module. If we put all variables in the dynamic part, then no constraints actually end up in the fixed part of the module.

Interesting Module Systems Our approach should be evaluated on other languages with interesting module systems. Especially interesting would be a language such as Haskell, where true separate compilation is already quite a challenge for the official compiler. In addition, there are large implementations written in Haskell which could be used for evaluation.

Cross-module Unification We do not support cross-module unification. However, we do believe it would be possible to support it. Whenever there is unification between two variables from different modules, a cyclic dependency could be recorded with the variables as detailed information. Likewise, whenever a module unifies a non-local variable with a value, we can also track this and introduce a dependency in the same way. The incremental strategies would need to be adjusted to also handle unification variables. The impact of these unification dependencies whenever there is cross-module unification would need to be investigated. As such, we believe this to be a good opportunity for future work.

Updating Module Names Updating the names of modules is currently not supported by our model. Whenever a file is renamed to a different file, this is currently considered to be the removal of the old file and the addition of the new file. This means that valuable information from the previous analysis is lost. A change to the name of any module changes the identity of all its scopes, triggering reanalysis of all dependencies. Tools like Git utilize techniques such as file similarity to determine when it is likely that a file has been renamed. Of course, these techniques are not limited to files alone, and can also be applied to changes within the file, e.g. renaming a method.

From Modules to Constraints Given the precision with which we can record which changes would affect the result of a query, the next step would be to use this information for something smaller than the modules. We believe it is possible to reanalyze individual constraints instead of entire modules. To achieve this, it would be necessary to track where the information from queries is used. Changes to the result of a query could then be mapped to the affected constraints, propagating changes in a similar way as we do for modules.

Bibliography

[1] Davide Ancona, Giovanni Lagorio, and Elena Zucca. "True separate compilation of Java classes". In: *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, October 6-8, 2002, Pittsburgh, PA, USA (Affiliated with PLI 2002)*. ACM, 2002, pp. 189–200. DOI: `10.1145/571157.571177`.

[2] Hendrik van Antwerpen et al. "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, 2016, pp. 49–60. ISBN: 978-1-4503-4097-7. DOI: `10.1145/2847538.2847543`.

[3] Hendrik van Antwerpen et al. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: `10.1145/3276484`.

[4] Andrew W. Appel and David B. MacQueen. "Separate Compilation for Standard ML". In: *PLDI*. 1994, pp. 13–23.

[5] Franz Baader and Wayne Snyder. "Unification Theory". In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 445–532. ISBN: 0-444-50813-9.

[6] James S. Briggs. "Separate compilation and the ADA programming language". British Library, EThOS. PhD thesis. University of York, UK, 1984.

[7] Luca Cardelli. "Program Fragments, Linking, and Modularization". In: *POPL*. 1997, pp. 266–277. DOI: `10.1145/263699.263735`.

[8] Magnus Carlsson. "Monads for incremental computing". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*. 2002, pp. 26–35. DOI: `10.1145/581478.581482`.

[9] Microsoft Corporation. *C# preprocessor directives*. 2015. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/` (visited on 08/28/2019).

[10] Manfred Dausmann et al. "A Separate Compilation System for Ada". In: *Werkzeuge der Programmiertechnik, GI-Arbeitstagung, Karlsruhe, 16.-17. März 1981, Proceedings*. Ed. by Gerhard Goos. Vol. 43. Informatik-Fachberichte. Springer, 1981, pp. 197–213. ISBN: 3-540-10725-8.

[11] Torbjörn Ekman and Görel Hedin. "Modular Name Analysis for Java Using JastAdd". In: *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2006, pp. 422–436. ISBN: 3-540-45778-X. DOI: `10.1007/11877028_18`.

[12] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18. isbn: 978-1-59593-786-5. doi: 10.1145/1297027.1297029.

[13] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. "A sound and optimal incremental build system with dynamic dependencies". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 89–106. isbn: 978-1-4503-3689-5. doi: 10.1145/2814270.2814316.

[14] Sebastian Erdweg et al. "A co-contextual formulation of type rules and its application to incremental type checking". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 880–897. isbn: 978-1-4503-3689-5. doi: 10.1145/2814270.2814277.

[15] Georgios Fourtounis and Nikolaos S. Papaspyrou. "Supporting Separate Compilation in a Defunctionalizing Compiler". In: *2nd Symposium on Languages, Applications and Technologies, SLATE 2013, June 20-21, 2013 - Porto, Portugal*. Ed. by José Paulo Leal, Ricardo Rocha, and Alberto Simões. Vol. 29. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 39–49. isbn: 978-3-939897-52-1. doi: 10.4230/OASIcs.SLATE.2013.39.

[16] Free Software Foundation, Inc. *The C Preprocessor*. 1987. url: https://gcc.gnu.org/onlinedocs/cpp/ (visited on 08/28/2019).

[17] Ben Gamari. *Deterministic builds*. url: https://gitlab.haskell.org/ghc/ghc/wikis/deterministic-builds (visited on 09/14/2019).

[18] E. Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[19] Zheng Gao, Christian Bird, and Earl T. Barr. "To type or not to type: quantifying detectable bugs in JavaScript". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE / ACM, 2017, pp. 758–769. isbn: 978-1-5386-3868-2. doi: http://dl.acm.org/citation.cfm?id=3097459.

[20] Brian Goetz. *Java Theory and Practice: Going Atomic*. url: https://www.ibm.com/developerworks/library/j-jtp11234/index.html (visited on 08/28/2019).

[21] Görel Hedin. "Incremental Semantic Analysis". PhD thesis. 1992.

[22] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000).

[23] Martin Jourdan et al. "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System". In: *PLDI*. 1990, pp. 209–222.

[24] Viggo Kann. "On the Approximability of the Maximum Common Subgraph Problem". In: *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*. Ed. by Alain Finkel and Matthias Jantzen. Vol. 577. Lecture Notes in Computer Science. Springer, 1992, pp. 377–388. isbn: 3-540-55210-3.

[25] Lennart C. L. Kats and Eelco Visser. “The Spoofax language workbench: rules for declarative specification of languages and IDEs”. In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/-Tahoe, Nevada: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869497`.

[26] Jan van Katwijk. “The Ada - compiler: On the design and implementation of an Ada compiler”. PhD thesis. Delft University of Technology, 1987.

[27] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. ISBN: 0-13-110163-3.

[28] Scott Kilpatrick et al. “Backpack: retrofitting Haskell with interfaces”. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 19–32. ISBN: 978-1-4503-2544-8. DOI: `10.1145/2535838.2535884`.

[29] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018. URL: `https://doc.rust-lang.org/1.30.0/book/first-edition/conditional-compilation.html`.

[30] Donald E. Knuth. “Semantics of Context-Free Languages”. In: *Theory Comput. Syst.* 2.2 (1968), pp. 127–145.

[31] Matthijs F. Kuiper and João Saraiva. “Lrc - A Generator for Incremental Language-Oriented Tools”. In: *Compiler Construction, 7th International Conference, CC 98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Kai Koskimies. Vol. 1383. Lecture Notes in Computer Science. Springer, 1998, pp. 298–301. ISBN: 3-540-64304-4.

[32] Xavier Leroy. “A modular module system”. In: *Journal of Functional Programming* 10.3 (2000), pp. 269–303.

[33] Sean McDirmid, Wilson C. Hsieh, and Matthew Flatt. “A Framework for Modular Linking in OO Languages”. In: *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*. Ed. by David E. Lightfoot and Clemens A. Szyperski. Vol. 4228. Lecture Notes in Computer Science. Springer, 2006, pp. 116–135. ISBN: 3-540-40927-0. DOI: `10.1007/11860990_9`.

[34] Pierre Néron et al. “A Theory of Name Resolution”. In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: `10.1007/978-3-662-46669-8_9`.

[35] David Lorge Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

[36] Thomas W. Reps. *Generating language-based environments*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1984. ISBN: 0-262-18115-0.

[37] Thomas W. Reps. “Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors”. In: *POPL*. 1982, pp. 169–176. DOI: `10.1145/582153.582172`.

[38] Arjen J. Rouvoet et al. “Knowing when to Ask”. unpublished. N.D.

[39] Emma Söderberg and Görel Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Tech. rep. 98. Department of Computer Science, Lund University, 2012.

[40] Tamás Szabó et al. "Incrementalizing lattice-based program analyses in Datalog". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). doi: 10.1145/3276509.

[41] The Apache Software Foundation. *Apache Commons CSV*. url: https://commons.apache.org/proper/commons-csv/ (visited on 09/10/2019).

[42] The Apache Software Foundation. *Apache Commons IO*. url: https://commons.apache.org/proper/commons-io/ (visited on 09/10/2019).

[43] Guido Wachsmuth et al. "A Language Independent Task Engine for Incremental Name and Type Analysis". In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 260–280. isbn: 978-3-319-02653-4. doi: 10.1007/978-3-319-02654-1_15.

[44] Leonid Yavits, Amir Morad, and Ran Ginosar. "The effect of communication and synchronization on Amdahl's law in multicore systems". In: *Parallel Computing* 40.1 (2014), pp. 1–16. doi: 10.1016/j.parco.2013.11.001.

Glossary

AST Abstract Syntax Tree. 5

clean run A single analysis of a project which analyzes all modules without using information from previous analyses. 45, 66, 68

critical edge An extension to the scope graph that will be added by a constraint which is yet to be evaluated. 12

free A variable is free if it does not have an associated value (yet). 32

global scope The scope where globally visible declarations are stored. There is a single global scope, owned by the root module. 26, 30

ground A term is considered ground if it contains no free variables. 32

incorrect dependency Whenever a dependency of m_1 on m_2 is recorded, but there is no change possible to m_2 such that m_1 is affected. 47

incremental run A single analysis of a project which has access to information from previous analyses, possibly allowing incremental computation of the new analysis result. 45, 66

incremental strategy An algorithm which uses the features of our incremental model to perform incremental analysis. That is, given the previous analysis result and a changeset of changed files in the project, a strategy has to determine the new analysis result. 45, 94

incremental analysis The analysis of a program which potentially uses information from a previous analysis to determine its result. An incremental analysis might skip reanalyzing certain modules if their reanalysis is not necessary to compute the new result. 15, 45

irrelevant dependency A dependency of m_1 on m_2 is irrelevant for a specific change if applying that change to m_2 does not affect m_1. 47

library A collection of implementations which has a well-defined interface. A library consists of multiple smaller modules and presents some interface to use the implementations of those modules in other libraries or programs. 29

library module A special module that can be queried, but which will not participate in the solving process otherwise. Library modules have no constraints and their scope graphs and unifiers cannot be modified.. 30

modularization The concept of breaking a program into smaller fragments called *modules*. 19, 25, 33

module A program fragment which represents some logically separate section of a program. 15, 30, 93

module A mapping of a module in the source language to a corresponding module in Statix. 25

module boundary A rule in a Statix specification can be marked as a module boundary. If this rule is called, a new module will be created as a child of the calling module. 26

non-ground A term is considered non-ground if it contains free variables. Likewise, a term is ground if it has a value without free variables. 32

permission to extend A predicate with *permission to extend* a scope s is allowed to introduce new edges and data edges in s. A predicate has permission to extend a scope s if it was created in that predicate, or if it was passed to it by a predicate with permission to extend s. 12

query answer stability Queries in Statix are only executed whenever it is known that their answer will not change as a result of edges which will be added to the scope graph after the query was executed. 31

scope graph A directed graph representing the lexical scoping of a program. Scope graphs consist of scopes, edges, declarations and data edges. 5, 6

separate compilation The technique of typechecking and compiling separate program fragments. These program fragments cannot be compiled in isolation, but can be compiled in an environment with type information about the missing fragments. 2, 18, 19

source language A Statix specification describes the name binding and typing rules of a particular language, which is called the source language of the specification. 25, 26, 94

standard library A library which is available to be used by any program, included by default in a particular programming language. 30

Statix specification A specification of the typing and name binding rules for a particular source language for Statix. 5

strategy *See* incremental strategy. 2

Appendix A

Mathematic Symbols

This appendix list all the mathematic symbols used throughout this thesis and their meaning.

A.1 Scope Graphs

$$
\begin{aligned}
d \in \mathit{Decls} &:= \text{terms denoting declarations} \\
s \in \mathit{Scopes} &:= \text{terms denoting scopes} \\
l \in \mathcal{L} &:= \text{strings denoting label names} \\
r \in \mathcal{R} &:= \text{strings denoting relation names} \\
p \in P &:= \text{paths describing a sequence of scopes connected by labels}
\end{aligned}
$$

A.1.1 Scope Graph Elements

$$
\begin{aligned}
s \in S &:= \text{set of scopes} \\
(s \xrightarrow{l} s') \in E &:= \text{set of edges} \\
(s \xrightarrow{r}\!\blacksquare\; d) \in D &:= \text{set of data edges} \\
\mathcal{G} &:= \text{scope graph of the form } \langle S, E, D \rangle \\
\text{scopes}(\langle S, E, D \rangle) &:= S \\
\text{edges}(\langle S, E, D \rangle) &:= E \\
\text{data}(\langle S, E, D \rangle) &:= D
\end{aligned}
$$

A.1.2 Terms

$$
\begin{aligned}
x \in \mathcal{V} &:= \text{variables} \\
t \in \mathcal{T} &:= x \mid d \mid s \mid p \mid [] \mid [t \mid t] \mid (t, ..., t)
\end{aligned}
$$

A.1.3 Name Resolution

$$\begin{aligned}
\mathit{WFD} &:= \text{predicate describing well-formedness of data}\\
\mathit{WFL} &:= \text{predicate describing well-formedness of data}\\
\leqslant_d &:= \text{ordering on data}\\
<_l &:= \text{ordering on labels}\\
\mathsf{query}\ \mathit{WFD}, \mathit{WFL}, \leqslant_d, <_l\ \mathsf{in}\ s \overset{r}{\longmapsto} \overline{(p,d)} &:= \text{A query in } s \text{ for relation } r\text{, resolving to paths and declarations } \overline{(p,d)}\\
q \in Q &:= \text{Set of queries}
\end{aligned}$$

A.2 Modules

Syntax

modules	$m \in M$	:=	some finite set
module graph	$\mathcal{G}_m$	::=	$\langle S_m, E_m, D_m \rangle$
module scopes	$s \in S_m$	::=	$\{s \mid s \in \text{scopes}(\mathcal{G}), \mathit{owner}(s) = m\}$
module edges	$e \in E_m$	::=	$\{e \mid e \in \text{edges}(\mathcal{G}), \mathit{owner}(e) = m\}$
module data	$d \in D_m$	::=	$\{d \mid d \in \text{data}(\mathcal{G}), \mathit{owner}(d) = m\}$

Formulae

parent	$M \nrightarrow M$		
children	$M \to \overline{M}$		
parents(m)	$M \to \overline{M}$	::=	$\{\{m'\} \cup \mathit{parents}(m') \mid m' = \mathit{parent}(m)\}$
descendants(m)	$M \to \overline{M}$	::=	$\{\{m'\} \cup \mathit{descendants}(m') \mid m' \in \mathit{children}(m)\}$
owner	$S \to M$		
owner	$E \to M$		
owner	$D \to M$		
extends	$S \to \overline{M}$		

A.3 Dependencies

$$\begin{aligned}
m \hookrightarrow m' &:= m \text{ depends on } m'\\
m \hookrightarrow m' \mid \langle V_e, V_d, R \rangle &:= m \text{ depends on } m' \text{ with detailed dependency information } V_e, V_d, R\\
(s,l) \in V_e &:= \text{set of scope, label combinations representing edges visited by a query}\\
(s,r) \in V_d &:= \text{set of scope, relation combinations representing edges visited by a query}\\
(s,l) \in R &:= \text{set of scope, label combinations representing edges traversed by a query}
\end{aligned}$$

Appendix B

Statix Specification

The Statix specification for Java is presented in Appendix B. The specification can also be found at https://github.com/MetaBorgCube/statix-incremental-artifact/blob/master/specifications/statics.stx.

```
module statics

signature

  ///////////////////////////////////////////////////////
  // 3.8. Identifiers
  ///////////////////////////////////////////////////////

  sorts Id constructors
    Id : string -> Id

  ///////////////////////////////////////////////////////
  // 4.3. Reference Types and Values
  ///////////////////////////////////////////////////////

  sorts Type constructors
    PT2T : PrimitiveType -> Type
    RT2T : ReferenceType -> Type
    Void : Type
    ArrayType : Type * list(Annotation) -> Type

  sorts PrimitiveType constructors
    NumericType : string -> PrimitiveType
    BooleanType : PrimitiveType

  sorts ReferenceType constructors
    CT2RT : ClassType -> ReferenceType
    AT2RT : ArrayType -> ReferenceType

  sorts ArrayType constructors
    ArrayTypePrimitive : PrimitiveType * list(Annotation) -> ArrayType
    ArrayTypeClassType : ClassType * list(Annotation) -> ArrayType

//    UCT2URT : ClassType -> ReferenceType

```

```
sorts ClassType constructors
  ClassType : list(Annotation) * Id * MaybeTypeArguments -> ClassType
  ClassOrInterfaceTypeMember : ClassType * list(Annotation) * Id *
  MaybeTypeArguments -> ClassType

////////////////////////////////////////////////////////
// 4.5. Parameterized Types
////////////////////////////////////////////////////////

sorts MaybeTypeArguments = list(TypeArguments)
sorts TypeArguments constructors
  TypeArguments : list(TypeArgument) -> TypeArguments

sorts TypeArgument = ReferenceType
                // = Wildcard

////////////////////////////////////////////////////////
// 6.5. Determining the Meaning of a Name
////////////////////////////////////////////////////////

sorts PackageName constructors
  PackageName : Id -> PackageName
  PackageName : PackageName * Id -> PackageName

sorts TypeName constructors
  TypeName : Id -> TypeName
  TypeName : PackageOrTypeName * Id -> TypeName

sorts PackageOrTypeName constructors
  PackageOrTypeName : Id -> PackageOrTypeName
  PackageOrTypeName : PackageOrTypeName * Id -> PackageOrTypeName

sorts ExpressionName constructors
  ExpressionName : Id -> ExpressionName
  ExpressionName : AmbiguousName * Id -> ExpressionName

sorts MethodName constructors
  MethodName : Id -> MethodName

sorts AmbiguousName constructors
  AmbiguousName : Id -> AmbiguousName
  AmbiguousName : AmbiguousName * Id -> AmbiguousName

////////////////////////////////////////////////////////
// 7.3. Compilation Units
////////////////////////////////////////////////////////

sorts CompilationUnit constructors
  CompilationUnit : MaybePackageDeclaration * list(ImportDeclaration) *
  list(TypeDeclaration) -> CompilationUnit

sorts TypeDeclaration constructors
```

```
  CD2TD : ClassDeclaration  -> TypeDeclaration
  ID2TD : InterfaceDeclaration  -> TypeDeclaration

////////////////////////////////////////////////////////
// 7.4. Package Declarations
////////////////////////////////////////////////////////

sorts MaybePackageDeclaration = list(PackageDeclaration)
sorts PackageDeclaration constructors
  PackageDeclaration : list(Annotation) * list(Id) -> PackageDeclaration

////////////////////////////////////////////////////////
// 7.5. Import Declarations
////////////////////////////////////////////////////////

sorts ImportDeclaration constructors
  SingleTypeImport   : TypeName -> ImportDeclaration
  TypeImportOnDemand : PackageOrTypeName -> ImportDeclaration

////////////////////////////////////////////////////////
// 8.1. Class Declarations
////////////////////////////////////////////////////////

sorts ClassDeclaration constructors
  NCD2CD : NormalClassDeclaration -> ClassDeclaration
  ED2CD  : EnumDeclaration -> ClassDeclaration

sorts NormalClassDeclaration constructors
  ClassDeclaration : list(ClassModifier) * Id * MaybeTypeParameters *
  MaybeSuperClass * MaybeSuperInterfaces * MaybeClassBodyDeclaration ->
  NormalClassDeclaration

sorts ClassModifier

sorts MaybeTypeParameters = list(TypeParameters)
sorts TypeParameters

sorts MaybeSuperClass = list(SuperClass)
sorts SuperClass constructors
  SuperClass : ClassType -> SuperClass

sorts MaybeSuperInterfaces = list(SuperInterfaces)
sorts SuperInterfaces constructors
  SuperInterface : list(ClassType) -> SuperInterfaces

sorts MaybeClassBodyDeclaration = list(ClassBodyDeclaration)
sorts ClassBodyDeclaration constructors
  CMD2CBD : ClassMemberDeclaration -> ClassBodyDeclaration

sorts ClassMemberDeclaration constructors
  FD2CMD : list(FieldDeclaration) -> ClassMemberDeclaration
  MD2CMD : MethodDeclaration -> ClassMemberDeclaration
  CD2CMD : ClassDeclaration -> ClassMemberDeclaration
  ID2CMD : InterfaceDeclaration -> ClassMemberDeclaration
  //Semicolon : ClassMemberDeclaration

sorts FieldDeclaration constructors
  TFieldDecl : list(FieldModifier) * Type * Id -> FieldDeclaration
  TFieldDeclArray : list(FieldModifier) * Type * Id * int -> FieldDeclaration

sorts MethodDeclaration constructors
  MethodDecl : list(MethodModifier) * string * MethodHeader * Statement -> MethodDeclaration

//Fields
sorts FieldModifier
sorts VarDecl

//Methods
sorts MethodModifier
sorts MethodHeader constructors
  MethodHeader : Type * Id * MethodParams * AnnotatedDimsEmpty * MaybeThrows -> MethodHeader
  MethodHeaderTypeParameters : TypeParameters * list(Annotation) * Type * Id * MethodParams * MaybeThrows -> MethodHeader

sorts MethodParams = list(FormalParam)
sorts FormalParam constructors
  FormalParam : list(VariableModifier) * Type * VarDeclId -> FormalParam
  ReceiverParamQual : list(Annotation) * Type * Id -> FormalParam
  ReceiverParam : list(Annotation) * Type -> FormalParam

sorts VariableModifier

sorts VarDeclId constructors
  VariableDecl : Id -> VarDeclId
  VariableDeclArray : Id * AnnotatedDims -> VarDeclId


sorts AnnotatedDimsEmpty
sorts AnnotatedDims = list(AnnotatedDim)
sorts AnnotatedDim
sorts MaybeThrows

/////////////////////////////////////////////////////////
// 8.9. Enum Types
/////////////////////////////////////////////////////////

sorts EnumDeclaration constructors
  EnumDeclComma : list(ClassModifier) * Id * MaybeSuperInterfaces * list(EnumConstant) * MaybeEnumBodyDeclarations -> EnumDeclaration
  EnumDecl : list(ClassModifier) * Id * MaybeSuperInterfaces * list(EnumConstant) * MaybeEnumBodyDeclarations -> EnumDeclaration

```

```
sorts EnumConstant

sorts MaybeEnumBodyDeclarations = list(EnumBodyDeclarations)
sorts EnumBodyDeclarations

////////////////////////////////////////////////////////
// 9.1. Interface Declarations
////////////////////////////////////////////////////////

sorts InterfaceDeclaration constructors
  NormalInterface : list(InterfaceModifier) * Id * MaybeTypeParameters * MaybeExtendsInterfaces * list(InterfaceMemberDeclaration) -> InterfaceDeclaration
  ATD2ID : AnnotationTypeDeclaration -> InterfaceDeclaration

sorts InterfaceModifier

sorts MaybeExtendsInterfaces = list(ExtendsInterfaces)
sorts ExtendsInterfaces constructors
  ExtendsInterfaces : list(ClassType) -> ExtendsInterfaces

sorts InterfaceMemberDeclaration constructors
  CD2IMD : ClassDeclaration -> InterfaceMemberDeclaration
  ConstD2IMD : list(FieldDeclaration) -> InterfaceMemberDeclaration
  IMD2IMD : InterfaceMethodDeclaration -> InterfaceMemberDeclaration
  ID2IMD : InterfaceDeclaration -> InterfaceMemberDeclaration
  //Semicolon : InterfaceMemberDeclaration

sorts InterfaceMethodDeclaration constructors
  AbstractMethodDec : list(InterfaceMethodModifier) * string * MethodHeader * Statement -> InterfaceMethodDeclaration

sorts InterfaceMethodModifier

////////////////////////////////////////////////////////
// 9.6. Annotation Types
////////////////////////////////////////////////////////

sorts AnnotationTypeDeclaration constructors
  AnnoDec : list(InterfaceModifier) * Id * list(AnnotationTypeMemberDeclaration) -> AnnotationTypeDeclaration

sorts AnnotationTypeMemberDeclaration

////////////////////////////////////////////////////////
// 9.7. Annotations
////////////////////////////////////////////////////////

sorts Annotation

////////////////////////////////////////////////////////
// 14 Statements
```

```
/////////////////////////////////////////////////////////
sorts Statement constructors
  Block : list(Statement) -> Statement
  ExpressionStatement : Expression -> Statement
  ExpressionsStatement : list(Expression) -> Statement
  VarDeclStatement : Type * Id -> Statement
  ArrayVarDeclStatement : Type * Id * int -> Statement

/////////////////////////////////////////////////////////
// Expressions
/////////////////////////////////////////////////////////

sorts Expression constructors
  //References to names
  EN2E : ExpressionName -> Expression

  //Primary
  L2E : Literal -> Expression
  CL2E : ClassLiteral -> Expression
  This : Expression
  QThis : TypeName -> Expression

  //Fields
  FA2E : FieldAccess -> Expression

  //Methods
  Invoke       : MethodName * list(Expression) -> Expression
  InvokeQExp   : Expression * MaybeTypeArguments * Id * list(Expression) -> Expression
  InvokeSuper  : MaybeTypeArguments * Id * list(Expression) -> Expression
  InvokeQSuper : TypeName * MaybeTypeArguments * Id * list(Expression) -> Expression

  //ArrayAccess + Array creation
  AA2E : ArrayAccess -> Expression
  ACE2E : ArrayCreationExpression -> Expression

  //Assignments
  Assign : Expression * Expression -> Expression

  //Eq and Add are flattened to primitives
  InstanceOf : Expression * Type -> Expression
  Cond : Expression * Expression * Expression -> Expression

  //Creating new instances
  UI2E : UnqualifiedInstance -> Expression
  QualifiedInstance : Expression * UnqualifiedInstance -> Expression

  //Casts
  CastPrimitive : Type * Expression -> Expression
  CastReference : Type * Other * Expression -> Expression

  sorts ArrayAccess constructors
    ArrayAccess : Expression * Expression -> ArrayAccess

  sorts ArrayCreationExpression constructors
    NewArray : Type * Other * Other -> ArrayCreationExpression
    NewArrayInit : Type * Other * Other -> ArrayCreationExpression

  sorts Other

  //Second is actually type arguments or diamond
  sorts UnqualifiedInstance constructors
    NewInstance : MaybeTypeArguments * list(Annotation) * Id * list(QualifiedId) * MaybeTypeArguments * list(Expression) -> UnqualifiedInstance
    NewInstance : MaybeTypeArguments * list(Annotation) * Id * list(QualifiedId) * MaybeTypeArguments * list(Expression) * list(ClassBodyDeclaration) -> UnqualifiedInstance

  sorts QualifiedId constructors
    QualifiedId : list(Annotation) * Id -> QualifiedId

  sorts FieldAccess constructors
    Field : Expression * Id -> FieldAccess
    SuperField : Id -> FieldAccess
    QSuperField : TypeName * Id -> FieldAccess

  sorts Literal constructors
    Literal : Type -> Literal
    Null : Literal

  sorts ClassLiteral constructors
    TypeNameClassLiteral    : TypeName * int -> ClassLiteral
    NumericTypeClassLiteral : PrimitiveType * int -> ClassLiteral
    BooleanClassLiteral     : int -> ClassLiteral
    VoidClassLiteral        : ClassLiteral

signature

  sorts PKG = (path * (occurrence * scope))

  sorts TYPE constructors
    ANNO  : scope -> TYPE
    CLASS : scope -> TYPE
    INTF  : scope -> TYPE
    ENUM  : scope -> TYPE
    PRIMITIVE : TYPE
    ARRAY : TYPE -> TYPE
    FUNCTION : list(TYPE) * TYPE -> TYPE
    VOID : TYPE

  name-resolution
    labels LEX  // lexical parent
```

```
           PKG  // package composition
           IMP  // import
           EXT  // extends
           IMPL // implements
    resolve Pkg  filter LEX*
                 min    $ < LEX
    resolve Type filter LEX*(e|PKG|IMP)
                 min    $ < LEX, $ < PKG, $ < IMP, IMP < PKG, PKG < LEX
    resolve Method filter LEX*EXT*IMPL*
                   min    $ < LEX, $ < EXT, $ < IMPL, EXT < LEX, IMPL < LEX,
    EXT < IMPL
    resolve Var filter LEX*EXT*
                min $ < LEX, $ < EXT, EXT < LEX
    resolve Builtin filter LEX*
                    min $ < LEX

  namespaces
    Type : Id
    Pkg  : list(Id)
    Method : Id
    Var : Id
    Builtin : string

  relations
    types    : occurrence -> TYPE
    packages : occurrence -> scope
    scopes   : occurrence -> scope
    supers   : occurrence -> list(occurrence)

rules

  projectOk : scope

  projectOk(s) :- {s_object s_array tLength}
    //Extend object in the program scope
    CLASS(s_object) == classTypeOkNoId(s,
    ClassOrInterfaceTypeMember(ClassOrInterfaceTypeMember(ClassType([],
    Id("java"), []), [], Id("lang"), []), [], Id("Object"), [])),
    s -EXT-> s_object,

    //Array with length field
    new s_array, s_array -LEX-> s_object,
    !scopes[Builtin{"array"@-}, s_array] in s,
    tLength == Id("length"),
    !types[Var{tLength@-}, PRIMITIVE()] in s_array.


rules

  fileOk : scope * CompilationUnit

  fileOk(s, CompilationUnit(maybePackageDeclaration, importDeclarations, typeDeclarations)) :-
  {s_cu}
    new s_cu, s_cu -LEX-> s,
    maybePackageDeclarationOk(s, maybePackageDeclaration, s_cu),
    importDeclarationsOk(s, importDeclarations, s_cu),
    typeDeclarationsOk(s_cu, typeDeclarations).


rules

  maybePackageDeclarationOk maps packageDeclarationOk(*, list(*), *)

  packageDeclarationOk : scope * PackageDeclaration * scope

  packageDeclarationOk(s, p@PackageDeclaration(annotations, ids), s_pkg) :-
  {pkgs}
    !packages[Pkg{ids@p}, s_pkg] in s,
    packages of Pkg{ids@-} in s |-> pkgs,
    composePackages(s_pkg, pkgs).


  composePackages maps composePackage(*, list(*))

  composePackage : scope * PKG

  composePackage(s, (_, (_, s'))) :-
    s -PKG-> s'.


rules

  importDeclarationsOk maps importDeclarationOk(*, list(*), *)

  importDeclarationOk : scope * ImportDeclaration * scope

  importDeclarationOk(s, SingleTypeImport(typeName), s_cu) :-
  {s_imp l}
    new s_imp, s_cu -IMP-> s_imp,
    resolveTypeNameAsList(s, typeName) == l,
    addImportIfFound(s, s_imp, typeName, l).

  importDeclarationOk(s, TypeImportOnDemand(packageOrTypeName), s_cu) :-
  {pkgs}
    resolvePackageOrTypeName(s, packageOrTypeName) == pkgs,
    importPackages(s_cu, pkgs).

  //Add the first import that we find (if there are multiple classes with the same full name, we pick the first one).
  //Otherwise, invent the imported type and associate it a class.
  addImportIfFound : scope * scope * TypeName * list((path * (occurrence * TYPE)))
```

```
addImportIfFound(s, s_imp, _, [(_, (occ, type))]) :-
  !types[occ, type] in s_imp.
addImportIfFound(s, s_imp, _, [(_, (occ, type)) | _]) :-
  !types[occ, type] in s_imp,
  false.
addImportIfFound(s, s_imp, TypeName(id), []) :- {s_notfound}
  new s_notfound, s_notfound -LEX-> s,
  !types[Type{id@-}, CLASS(s_notfound)] in s_imp,
  false.
addImportIfFound(s, s_imp, TypeName(_, id), []) :- {s_notfound}
  new s_notfound, s_notfound -LEX-> s,
  !types[Type{id@-}, CLASS(s_notfound)] in s_imp,
  false.


resolveTypeNameAsList : scope * TypeName -> list((path * (occurrence * TYPE)))

resolveTypeNameAsList(s, TypeName(id)) = l :-
  types of Type{id@-} in s |-> l.

resolveTypeNameAsList(s, TypeName(packageOrTypeName, id)) = l :-
{s_imp pkgs}
  new s_imp,
  resolvePackageOrTypeName(s, packageOrTypeName) == pkgs,
  importPackages(s_imp, pkgs),
  resolveTypeNameAsList(s_imp, TypeName(id)) == l.

resolveTypeName : scope * TypeName -> (occurrence * TYPE)

resolveTypeName(s, TypeName(id)) = type :-
  types of Type{id@-} in s |-> [(_, type)]. // FIXME Is `s` correct here, or should that be the package scope?

resolveTypeName(s, TypeName(packageOrTypeName, id)) = type :-
{s_imp pkgs}
  new s_imp,
  resolvePackageOrTypeName(s, packageOrTypeName) == pkgs,
  importPackages(s_imp, pkgs),
  resolveTypeName(s_imp, TypeName(id)) == type.

resolveTypeName2 : scope * TypeName -> (occurrence * TYPE)

resolveTypeName2(s, TypeName(id)) = type :-
  types of Type{id@-} in s |-> [(_, type)]. // FIXME Is `s` correct here, or should that be the package scope?

resolveTypeName2(s, TypeName(packageOrTypeName, id)) = type :-
{s_imp pkgs}
  new s_imp,
  resolvePackageOrTypeName(s, packageOrTypeName) == pkgs,
  importPackages(s_imp, pkgs),
  resolveTypeName2(s_imp, TypeName(id)) == type.
resolvePackageOrTypeName : scope * PackageOrTypeName -> list(PKG)

resolvePackageOrTypeName(s, packageOrTypeName) = pkgs :-
{x i}
  packageOrTypeName2Pkg(packageOrTypeName, []) == Pkg{x@i},
  packages of Pkg{x@i} in s |-> pkgs.


packageOrTypeName2Pkg : PackageOrTypeName * list(Id) -> occurrence

  packageOrTypeName2Pkg(PackageOrTypeName(id), ids) = Pkg{[id|ids]@-}.

  packageOrTypeName2Pkg(PackageOrTypeName(packageOrTypeName, id), ids) =
  packageOrTypeName2Pkg(packageOrTypeName, [id|ids]).


  importPackages maps importPackage(*, list(*))

  importPackage : scope * PKG

importPackage(s, (_, (_, s'))) :-
  s -IMP-> s'.


rules

typeDeclarationsOk maps typeDeclarationOk(*, list(*))

typeDeclarationOk : scope * TypeDeclaration

typeDeclarationOk(s, CD2TD(cd)) :-
  classDeclarationOk(s, cd).

typeDeclarationOk(s, ID2TD(id)) :-
  interfaceDeclarationOk(s, id).


rules

classDeclarationsOk maps classDeclarationOk(*, list(*))

classDeclarationOk : scope * ClassDeclaration

classDeclarationOk(s, NCD2CD(ncd)) :-
  normalClassDeclarationOk(s, ncd).

classDeclarationOk(s, ED2CD(ed)) :-
  enumDeclarationOk(s, ed).
```

```
rules

  normalClassDeclarationsOk maps normalClassDeclarationOk(*, list(*))

  normalClassDeclarationOk : scope * NormalClassDeclaration

  normalClassDeclarationOk(s, ClassDeclaration(normalClassModifiers, id,
    maybeTypeParameters, maybeSuperClass, maybeSuperInterfaces,
    normalClassBodyDeclarations)) :-
//  modbound normalClassDeclarationOk(s, ClassDeclaration(normalClassModifiers,
    id@Id(name), maybeTypeParameters, maybeSuperClass, maybeSuperInterfaces,
    normalClassBodyDeclarations))
//  | $[[name]] :-
  {s_cls tThis}
    new s_cls, s_cls -LEX-> s,
    !types[Type{id@id}, CLASS(s_cls)] in s,
    tThis == Id("this"), !types[Var{tThis@-}, CLASS(s_cls)] in s_cls,
    maybeSuperClassOk(s, maybeSuperClass, s_cls),
    maybeSuperInterfacesOk(s, maybeSuperInterfaces, s_cls),
    maybeClassBodyDeclarationsOk(s, normalClassBodyDeclarations, s_cls).

  flatten : list(list(occurrence)) -> list(occurrence)
  flatten([]) = [].
  flatten([[]]) = [].
  flatten([ l ]) = l.
  flatten([ [h | t1] | t2]) = [h | join(t1, flatten(t))].

  join: list(occurrence) * list(occurrence) -> list(occurrence)
  join([], l) = l.
  join([h | t], l) = [h | join(t, l)].

rules

  enumDeclarationsOk maps enumDeclarationOk(*, list(*))

  enumDeclarationOk : scope * EnumDeclaration

  enumDeclarationOk(s, EnumDeclComma(classModifiers, id, maybeSuperInterfaces,
    enumConstants, maybeEnumBodyDeclarations)) :-
//  modbound enumDeclarationOk(s, EnumDeclComma(classModifiers, id@Id(name),
    maybeSuperInterfaces, enumConstants, maybeEnumBodyDeclarations))
//  | $[[name]] :-
  {s_enum tThis}
    new s_enum, s_enum -LEX-> s,
    !types[Type{id@id}, ENUM(s_enum)] in s,
    tThis == Id("this"), !types[Var{tThis@-}, ENUM(s_enum)] in s_enum,
    maybeSuperInterfacesOk(s, maybeSuperInterfaces, s_enum).

  enumDeclarationOk(s, EnumDecl(classModifiers, id, maybeSuperInterfaces,
    enumConstants, maybeEnumBodyDeclarations)) :-
//  modbound enumDeclarationOk(s, EnumDecl(classModifiers, id@Id(name),
    maybeSuperInterfaces, enumConstants, maybeEnumBodyDeclarations))
```

```
//  | $[[name]] :-
  {s_enum tThis}
    new s_enum, s_enum -LEX-> s,
    !types[Type{id@id}, ENUM(s_enum)] in s,
    tThis == Id("this"), !types[Var{tThis@-}, ENUM(s_enum)] in s_enum,
    maybeSuperInterfacesOk(s, maybeSuperInterfaces, s_enum).

rules

  interfaceDeclarationsOk maps interfaceDeclarationOk(*, list(*))

  interfaceDeclarationOk : scope * InterfaceDeclaration

  interfaceDeclarationOk(s, NormalInterface(interfaceModifiers, id, maybeTypeParameters, maybeExtendsInterfaces, interfaceMemberDeclarations)) :-
//  modbound interfaceDeclarationOk(s, NormalInterface(interfaceModifiers, id@Id(name), maybeTypeParameters, maybeExtendsInterfaces, interfaceMemberDeclarations))
//  | $[[name]] :-
  {s_intf tThis}
    new s_intf, s_intf -LEX-> s,
    !types[Type{id@id}, INTF(s_intf)] in s,
    tThis == Id("this"), !types[Var{tThis@-}, INTF(s_intf)] in s_intf,
    maybeExtendsInterfacesOk(s, maybeExtendsInterfaces),
    maybeInterfaceMemberDeclarationsOk(s, interfaceMemberDeclarations, s_intf).

  interfaceDeclarationOk(s, ATD2ID(atd)) :-
    annotationTypeDeclarationOk(s, atd).

rules

  annotationTypeDeclarationsOk maps annotationTypeDeclarationOk(*, list(*))

  annotationTypeDeclarationOk : scope * AnnotationTypeDeclaration

  annotationTypeDeclarationOk(s, AnnoDec(interfaceModifiers, id, annotationTypeMemberDeclarations)) :-
  {s_anno}
    new s_anno, s_anno -LEX-> s,
    !types[Type{id@id}, ANNO(s_anno)] in s.


rules

  maybeSuperClassOk maps superClassOk(*, list(*), *)
  superClassOk : scope * SuperClass * scope
  superClassOk(s, SuperClass(classType), s_this) :-
  {s_super}
    classTypeOk(s_this, classType) == CLASS(s_super), //was lookup in s
    s_this -EXT-> s_super.

```

```

rules

  maybeSuperInterfacesOk maps superInterfacesOk(*, list(*), *)
  superInterfacesOk : scope * SuperInterfaces * scope
  superInterfacesOk(s, SuperInterface(classTypes), s_this) :-
  {types}
    classTypesOk(s_this, classTypes) == types, //was lookup in s
    implementInterfaces(types, s).

  implementInterfaces maps implementInterface(list(*), *)
  implementInterface : TYPE * scope
  implementInterface(INTF(s_intf), s_this) :-
    s_this -IMPL-> s_intf.


rules

  maybeExtendsInterfacesOk maps extendsInterfacesOk(*, list(*))
  extendsInterfacesOk : scope * ExtendsInterfaces
  extendsInterfacesOk(s, ExtendsInterfaces(classTypes)) :-
    classTypesOk(s, classTypes) == _.


rules

  classTypesOk maps classTypeOk(*, list(*)) = list(*)
  classTypeOk : scope * ClassType -> TYPE
  classTypeOk(s, ClassType(annotations, id, maybeTypeArguments)) = ty :-
    types of Type{id@-} in s |-> [(_, (_, ty))].

  classTypeOk(s, ClassOrInterfaceTypeMember(classType, annotations, id,
    maybeTypeArguments)) = ty :-
  {x i pkgs s_imp}
    classType2Pkg(classType, []) == Pkg{x@i},
    packages of Pkg{x@i} in s |-> pkgs,
    new s_imp,
    importPackages(s_imp, pkgs),
    classTypeOk(s_imp, ClassType(annotations, id, maybeTypeArguments)) == ty.

  //Does not add a termId
  classTypeOkNoId : scope * ClassType -> TYPE
  classTypeOkNoId(s, ClassType(annotations, id, maybeTypeArguments)) = T :-
    types of Type{id@-} in s |-> [(_, (_, T))].

  classTypeOkNoId(s, ClassOrInterfaceTypeMember(classType, annotations, id,
    maybeTypeArguments)) = T :-
  {x i pkgs s_imp}
    classType2Pkg(classType, []) == Pkg{x@i},
    packages of Pkg{x@i} in s |-> pkgs,
    new s_imp,
    importPackages(s_imp, pkgs),
    classTypeOkNoId(s_imp, ClassType(annotations, id, maybeTypeArguments)) == T.


  classType2Pkg : ClassType * list(Id) -> occurrence
  classType2Pkg(ClassType(_, id, _), ids) = Pkg{[id|ids]@-}.

  classType2Pkg(ClassOrInterfaceTypeMember(classType, _, id, _), ids) =
    classType2Pkg(classType, [id|ids]).

//----------------------------------------------
//Bodies
//----------------------------------------------
rules
  maybeClassBodyDeclarationsOk maps maybeClassBodyDeclarationOk(*, list(*), *)
  maybeClassBodyDeclarationOk: scope * ClassBodyDeclaration * scope

  maybeClassBodyDeclarationOk(s, CMD2CBD(FD2CMD(fields)), s_cls) :-
    maybeFieldDeclsOk(s, fields, s_cls).
//  modbound maybeClassBodyDeclarationOk(s, CMD2CBD(MD2CMD(MethodDecl(mods, uid, header, Block(stmts)))), s_cls) | $[[uid]] :- {s_mtd}
  maybeClassBodyDeclarationOk(s, CMD2CBD(MD2CMD(MethodDecl(mods, uid, header, Block(stmts)))), s_cls) :- {s_mtd}
    new s_mtd, s_mtd -LEX-> s_cls,
    methodHeaderOk(s, header, s_cls, s_mtd),
    statementsOk(s, stmts, s_mtd).

  //Inner class
  maybeClassBodyDeclarationOk(s, CMD2CBD(CD2CMD(cd)), s_cls) :-
    classDeclarationOk(s_cls, cd).

  //Inner interface
  maybeClassBodyDeclarationOk(s, CMD2CBD(ID2CMD(id)), s_cls) :-
    interfaceDeclarationOk(s_cls, id).

  //Initializer / Constructor
  maybeClassBodyDeclarationOk(_, _, _).

  //Interfaces
  maybeInterfaceMemberDeclarationsOk maps interfaceMemberDeclarationOk(*, list(*), *)
  interfaceMemberDeclarationOk : scope * InterfaceMemberDeclaration * scope
  interfaceMemberDeclarationOk(s, IMD2IMD(AbstractMethodDec(mods, _, header, Block([]))), s_intf) :-
  {s_mtd}
    new s_mtd, s_mtd -LEX-> s_intf,
    methodHeaderOk(s, header, s_intf, s_mtd).
//  modbound interfaceMemberDeclarationOk(s, IMD2IMD(AbstractMethodDec(mods, uid, header, Block(stmts@[_ | _]))), s_intf) | $[[uid]] :-
  interfaceMemberDeclarationOk(s, IMD2IMD(AbstractMethodDec(mods, uid, header, Block(stmts@[_ | _]))), s_intf) :-
  {s_mtd}
    new s_mtd, s_mtd -LEX-> s_intf,
    methodHeaderOk(s, header, s_intf, s_mtd),
    statementsOk(s, stmts, s_mtd).

  interfaceMemberDeclarationOk(s, CD2IMD(cd), s_intf) :-
    classDeclarationOk(s_intf, cd).

  interfaceMemberDeclarationOk(s, ID2IMD(id), s_intf) :-
    interfaceDeclarationOk(s_intf, id).

  interfaceMemberDeclarationOk(s, ConstD2IMD(fields), s_intf) :-
    maybeFieldDeclsOk(s, fields, s_intf).

  //Semicolon
  interfaceMemberDeclarationOk(_, _, _).


//-------------------------------------------------------------------------------------------------
//Fields
//-------------------------------------------------------------------------------------------------
rules
  maybeFieldDeclsOk maps maybeFieldDeclOk(*, list(*), *)
  maybeFieldDeclOk : scope * FieldDeclaration * scope
  maybeFieldDeclOk(s, TFieldDecl(mods, ty, name), s_cls) :-
    !types[Var{name@name}, typeOk(s_cls, ty)] in s_cls.
  maybeFieldDeclOk(s, TFieldDeclArray(mods, ty, name, count), s_cls) :-
    !types[Var{name@name}, ARRAY(typeOk(s_cls, ty))] in s_cls.

//-------------------------------------------------------------------------------------------------
//Methods
//-------------------------------------------------------------------------------------------------
rules
  methodHeaderOk : scope * MethodHeader * scope * scope
  methodHeaderOk(s, MethodHeader(ty, name, params, _, throws), s_cls, s_mtd) :- {rty ptys}
    !types[Method{name@name}, FUNCTION(ptys, rty)] in s_cls,
    rty == typeOk(s_cls, ty),
    ptys == paramTypesOk(s_cls, params, s_mtd).
  methodHeaderOk(s, MethodHeaderTypeParameters(typs, annots, ty, name, params, throws), s_cls, s_mtd) :- {rty ptys}
    !types[Method{name@name}, FUNCTION(ptys, rty)] in s_cls,
    rty == typeOk(s_cls, ty),
    ptys == paramTypesOk(s_cls, params, s_mtd).

  paramTypesOk maps paramTypeOk(*, list(*), *) = list(*)
  paramTypeOk : scope * FormalParam * scope -> TYPE
  paramTypeOk(s, FormalParam(mods, ty, VariableDecl(id)), s_mtd) = T :-
    T == typeOk(s, ty),
    !types[Var{id@id}, T] in s_mtd.
  paramTypeOk(s, FormalParam(mods, ty, VariableDeclArray(id, dims)), s_mtd) = T :-
    T == typeOk(s, ty),
    !types[Var{id@id}, ARRAY(T)] in s_mtd.

  paramTypeOk(s, ReceiverParam(annots, ty), s_mtd) = typeOk(s, ty).
  paramTypeOk(s, ReceiverParamQual(annots, ty, id), s_mtd) = typeOk(s, ty).

//----------------------------------------------------------------------------------------------
//Statements
//----------------------------------------------------------------------------------------------
rules
  statementsOk maps statementOk(*, list(*), *)
  statementOk: scope * Statement * scope
  statementOk(s, Block(stmts), s_method) :- {s_block}
    new s_block,
    s_block -LEX-> s_method,
    statementsOk(s, stmts, s_block).

  statementOk(s, ExpressionStatement(exp), s_mtd) :- {T}
    expressionOk(s, exp, s_mtd) == T.

  statementOk(s, ExpressionsStatement(exps), s_mtd) :- {Ts}
    expressionsOk(s, exps, s_mtd) == Ts.

  statementOk(s, VarDeclStatement(ty, name), s_mtd) :-
    !types[Var{name@name}, typeOk(s_mtd, ty)] in s_mtd.

  statementOk(s, ArrayVarDeclStatement(ty, name, dim), s_mtd) :-
    !types[Var{name@name}, ARRAY(typeOk(s_mtd, ty))] in s_mtd.

//----------------------------------------------------------------------------------------------
//Expressions
//----------------------------------------------------------------------------------------------
rules
  expressionsOk maps expressionOk(*, list(*), *) = list(*)
  expressionOk : scope * Expression * scope -> TYPE

  expressionOk(s, EN2E(ExpressionName(id)), s_mtd) = T :-
    types of Var{id@-} in s_mtd |-> [(_, (_, T))].
  expressionOk(s, EN2E(ExpressionName(ambName, id)), s_mtd) = T :-
  {tyAmb sTyAmb}
    handleAmbiguous(s_mtd, ambName) == [(_, (_, tyAmb))],
    scopeOf(s, tyAmb) == sTyAmb,
    types of Var{id@-} in sTyAmb |-> [(_, (_, T))].

  //Fields
  expressionOk(s, FA2E(Field(exp, id)), s_mtd) = T :-
    {Texp s_ty}
    Texp == expressionOk(s, exp, s_mtd),
    s_ty == scopeOf(s, Texp),
    types of Var{id@-} in s_ty |-> [(_, (_, T))].

  expressionOk(s, FA2E(SuperField(id)), s_mtd) = T :-
    query types filter LEX* EXT EXT* and { Var{id@-} }
                min $ < EXT, EXT < LEX, $ < LEX and true
```

```
              in s_mtd |-> [(_, (_, T))].

expressionOk(s, FA2E(QSuperField(typeName, id)), s_mtd) = T :- {tyTN sTN}
  resolveTypeName(s_mtd, typeName) == (_, tyTN),
  scopeOf(s, tyTN) == sTN,
  types of Var{id@-} in sTN |-> [(_, (_, T))].

//Methods
expressionOk(s, Invoke(MethodName(id), args), s_mtd) = T :- {methods}
  types of Method{id@-} in s_mtd |-> methods,
  (_, FUNCTION(_, T)) == findMethod(s, expressionsOk(s, args, s_mtd),
  methods, s_mtd).

expressionOk(s, InvokeQExp(exp, tyargs, id, args), s_mtd) = T :- {Texp sExp
  methods}
  expressionOk(s, exp, s_mtd) == Texp,
  scopeOf(s, Texp) == sExp,
  types of Method{id@-} in sExp |-> methods,
  (_, FUNCTION(_, T)) == findMethod(s, expressionsOk(s, args, s_mtd),
  methods, s_mtd).

expressionOk(s, InvokeSuper(tyargs, id, args), s_mtd) = T :- {methods}
  query types filter LEX* EXT EXT* and { Method{id@-} }
              min $ < EXT, EXT < LEX, $ < LEX and true
              in s_mtd |-> methods,
  (_, FUNCTION(_, T)) == findMethod(s, expressionsOk(s, args, s_mtd),
  methods, s_mtd).

expressionOk(s, InvokeQSuper(typeName, tyArgs, id, args), s_mtd) = T :- {tyTN
  sTN methods}
  resolveTypeName(s_mtd, typeName) == (_, tyTN),
  scopeOf(s, tyTN) == sTN,
  types of Method{id@-} in sTN |-> methods,
  (_, FUNCTION(_, T)) == findMethod(s, expressionsOk(s, args, s_mtd),
  methods, s_mtd).

/*
 * Tries to find a matching method. If not successful, returns the first
  match.
 */
findMethod : scope * list(TYPE) * list((path * (occurrence * TYPE))) * scope
  -> (occurrence * TYPE)
findMethod(s, argTys, [(_, p@(_, FUNCTION(argTys, _))) | _], s_mtd) = p.
findMethod(s, argTys, [(_, p) | tail], s_mtd) = findMethodOrGuess(argTys,
  tail, p).

findMethodOrGuess : list(TYPE) * list((path * (occurrence * TYPE))) *
  (occurrence * TYPE) -> (occurrence * TYPE)
findMethodOrGuess(argTys, [], p) = p.
findMethodOrGuess(argTys, [(_, p@(_, FUNCTION(argTys, _))) | _], _) = p.
findMethodOrGuess(argTys, [_ | tail], p) = findMethodOrGuess(argTys, tail, p).

```

```
//Literals
expressionOk(s, L2E(Literal(ty)), s_mtd) = typeOk(s, ty).
expressionOk(s, L2E(Null()), s_mtd) = CLASS(s).
expressionOk(s, This(), s_mtd) = T :- {tThis}
  tThis == Id("this"), types of Var{tThis@-} in s_mtd |-> [(_, (_, T))].
expressionOk(s, QThis(tyName), s_mtd) = T :-
  resolveTypeName(s_mtd, tyName) == (_, T).
expressionOk(s, CL2E(_), s_mtd) = T :-
  resolveTypeName2(s_mtd,
  TypeName(PackageOrTypeName(PackageOrTypeName(Id("lang")), Id("java")),
  Id("Class"))) == (_, T).

//InstanceCreation
expressionOk(s, UI2E(NewInstance(typeArgs, annots, id, qids, typeArgs2,
  args)), s_mtd) = T :-
{tyArgs}
  expressionsOk(s, args, s_mtd) == tyArgs,
  resolveTypeName(s_mtd, toTypeName([id | qidsToIds(qids)])) == (_, T).
expressionOk(s, UI2E(NewInstance(typeArgs, annots, id, qids, typeArgs2, args,
  cbds)), s_mtd) = T :-
{tyArgs}
  expressionsOk(s, args, s_mtd) == tyArgs,
  resolveTypeName(s_mtd, toTypeName([id | qidsToIds(qids)])) == (_, T).
expressionOk(s, QualifiedInstance(exp, ui), s_mtd) = T :-
{s_exp}
  scopeOf(s, expressionOk(s, exp, s_mtd)) == s_exp,
  expressionOk(s, UI2E(ui), s_exp) == T.

expressionOk(s, AA2E(ArrayAccess(e1, e2)), s_mtd) = T :-
  expressionOk(s, e1, s_mtd) == ARRAY(T),
  expressionOk(s, e2, s_mtd) == _.
expressionOk(s, ACE2E(NewArray(ty, _, _)), s_mtd) = typeOk(s_mtd, ty).
expressionOk(s, ACE2E(NewArrayInit(ty, _, _)), s_mtd) = typeOk(s_mtd, ty).

expressionOk(s, Assign(e1, e2), s_mtd) = T :-
  expressionOk(s, e1, s_mtd) == T,
  expressionOk(s, e2, s_mtd) == _.

expressionOk(s, Cond(e1, e2, e3), s_mtd) = T :-
  expressionOk(s, e1, s_mtd) == _,
  expressionOk(s, e3, s_mtd) == _,
  expressionOk(s, e2, s_mtd) == T.

expressionOk(s, InstanceOf(e1, ty), s_mtd) = PRIMITIVE() :-
  expressionOk(s, e1, s_mtd) == _,
  typeOk(s_mtd, ty) == _.

expressionOk(s, CastPrimitive(ty, exp), s_mtd) = typeOk(s_mtd, ty) :-
  expressionOk(s, exp, s_mtd) == _.
expressionOk(s, CastReference(ty, _, exp), s_mtd) = typeOk(s_mtd, ty) :-
```

```
    expressionOk(s, exp, s_mtd) == _.

  //Eq, Add are flattened to primitives

  expressionOk(s, e, s_mtd) = PRIMITIVE().

  qidsToIds maps qidToId(list(*)) = list(*)
  qidToId : QualifiedId -> Id
  qidToId(QualifiedId(annots, id)) = id.
  toTypeName : list(Id) -> TypeName
  toTypeName([ id ]) = TypeName(id).
  toTypeName([ id | tail]) = TypeName(toPackageOrTypeName(tail), id).

  toPackageOrTypeName : list(Id) -> PackageOrTypeName
  toPackageOrTypeName([ id ]) = PackageOrTypeName(id).
  toPackageOrTypeName([ id | tail]) =
    PackageOrTypeName(toPackageOrTypeName(tail), id).

//-----------------------------------------------------------------------------------------------
//Types
//-----------------------------------------------------------------------------------------------
rules
  typesOk maps typeOk(*, list(*)) = list(*)
  typeOk: scope * Type -> TYPE
  typeOk(s, RT2T(CT2RT(ct))) = classTypeOk(s, ct).
  typeOk(s, RT2T(AT2RT(at))) = arrayTypeOk(s, at).
  typeOk(s, PT2T(pt)) = PRIMITIVE().
  typeOk(s, Void()) = VOID().
  typeOk(s, ArrayType(ty, _)) = ARRAY(typeOk(s, ty)).

  arrayTypeOk: scope * ArrayType -> TYPE
  arrayTypeOk(s, ArrayTypePrimitive(_, _)) = ARRAY(PRIMITIVE()).
  arrayTypeOk(s, ArrayTypeClassType(ct, _)) = ARRAY(classTypeOk(s, ct)).

  scopeOf : scope * TYPE -> scope
  scopeOf(sg, CLASS(s)) = s.
  scopeOf(sg, ANNO(s)) = s.
  scopeOf(sg, INTF(s)) = s.
  scopeOf(sg, ENUM(s)) = s.
  scopeOf(sg, ARRAY(_)) = s :-
    scopes of Builtin{"array"@-} in sg |-> [(_, (_, s))].

  maybeScopeOf : scope * TYPE -> list(scope)
  maybeScopeOf(_, CLASS(s)) = [s].
  maybeScopeOf(_, ANNO(s)) = [s].
  maybeScopeOf(_, INTF(s)) = [s].
  maybeScopeOf(_, ENUM(s)) = [s].
  maybeScopeOf(sg, ARRAY(_)) = [s] :-
    scopes of Builtin{"array"@-} in sg |-> [(_, (_, s))].
  maybeScopeOf(_, _) = [].

//-----------------------------------------------------------------------------------------------
```

```
//Subtyping
//----------------------------------------------
rules


//----------------------------------------------
//Ambiguous handling
//----------------------------------------------
rules
  /*
   * Resolves a type in the given scope and returns its results.
   */
  resolveTypeToPOT : scope * Id -> list((path * (occurrence * TYPE)))
  resolveTypeToPOT(s, id) = results :-
    types of Type{id@-} in s |-> results.
  resolveTypeToPOTIfEmpty : list((path * (occurrence * TYPE))) * scope * Id ->
    list((path * (occurrence * TYPE)))
  resolveTypeToPOTIfEmpty([], s, id) = resolveTypeToPOT(s, id).
  resolveTypeToPOTIfEmpty(l, _, _) = l.

  /*
   * Resolves a variable in the given scope. If that has no results, resolves
    as type instead.
   */
  //Tries as var, if that fails, tries as type.
  resolveVarsOrTypesToPOT maps resolveVarOrTypeToPOT(list(*), *) = list(*)
  resolveVarOrTypeToPOT : scope * Id -> list((path * (occurrence * TYPE)))
  resolveVarOrTypeToPOT(s, id) = results :-
  {varResults}
    types of Var{id@-} in s |-> varResults,
    //if results is empty, tries to resolve as type. Otherwise, returns the
    input
    resolveTypeToPOTIfEmpty(varResults, s, id) == results.
  resolveVarOrTypeToPOTIfEmpty : list((path * (occurrence * TYPE))) * scope *
    Id -> list((path * (occurrence * TYPE)))
  resolveVarOrTypeToPOTIfEmpty([], s, id) = resolveVarOrTypeToPOT(s, id).
  resolveVarOrTypeToPOTIfEmpty(l, _, _) = l.

  /*
   * Handles an ambiguous name, by either resolving it as a Var or as a Type
    (in that order).
   */
  handleAmbiguous : scope * AmbiguousName -> list((path * (occurrence * TYPE)))
  handleAmbiguous(s_mtd, AmbiguousName(id)) = results :-
  {varResults}
    types of Var{id@-} in s_mtd |-> varResults,
    //if varResults is empty, tries to resolve as type. Otherwise, returns the
    input
    resolveTypeToPOTIfEmpty(varResults, s_mtd, id) == results.
  handleAmbiguous(s_mtd, ambCur@AmbiguousName(amb, id)) = results :-
  {firstResults}
    //Resolve first part of name
    handleAmbiguous(s_mtd, amb) == firstResults,
    //If empty, we now try to resolve it as a FQT. Otherwise, we try to find our id in the scopes of our results.
    handleAmbiguousIfFirstEmpty(firstResults, s_mtd, ambCur, id) == results.

  /*
   * Helper function for ambiguous resolution.
   * If the given list is empty, it tries to resolve the given AmbiguousName as a fully qualified name.
   * Otherwise, it tries to find the given id in the scopes of the list.
   */
  handleAmbiguousIfFirstEmpty : list((path * (occurrence * TYPE))) * scope * AmbiguousName * Id -> list((path * (occurrence * TYPE)))
  handleAmbiguousIfFirstEmpty([], s, amb, id) = resolveAmbiguousAsType(s, amb).
  handleAmbiguousIfFirstEmpty(l, s, _, id) = flattenPOT(resolveVarsOrTypesToPOT(flattenScopes(potToScopes(s, l)), id)).

  /*
   * Resolves an ambiguous name as a fully qualified type.
   */
  resolveAmbiguousAsType : scope * AmbiguousName -> list((path * (occurrence * TYPE)))
  resolveAmbiguousAsType(s, AmbiguousName(id)) = results :-
    types of Type{id@-} in s |-> results.
  resolveAmbiguousAsType(s, AmbiguousName(amb, id)) = results :-
  {x i pkgs s_ref}
    amb2Pkg(amb, []) == Pkg{x@i},
    packages of Pkg{x@i} in s |-> pkgs,
    new s_ref,
    importPackages(s_ref, pkgs),
    resolveTypeToPOT(s_ref, id) == results.

  /*
   * Converts an ambiguous name to a package.
   */
  amb2Pkg : AmbiguousName * list(Id) -> occurrence
  amb2Pkg(AmbiguousName(id), ids) = Pkg{[id|ids]@-}.
  amb2Pkg(AmbiguousName(amb, id), ids) = amb2Pkg(amb, [id|ids]).

//------------------------------------------------------------------------------------------
//Helper functions
//------------------------------------------------------------------------------------------
rules
  joinPOT : list((path * (occurrence * TYPE))) * list((path * (occurrence * TYPE))) -> list((path * (occurrence * TYPE)))
  joinPOT([], l) = l.
  joinPOT(l@[_ | _], []) = l.
  joinPOT([h | t], l) = [h | joinPOT(t, l)].

  potToScopes maps potToScope(*, list(*)) = list(*)
  potToScope : scope * (path * (occurrence * TYPE)) -> list(scope)
  potToScope(s, (_, (_, T))) = maybeScopeOf(s, T).
```

```
flattenPOT : list(list((path * (occurrence * TYPE)))) -> list((path *
  (occurrence * TYPE)))
flattenPOT([]) = [].
flattenPOT([ l ]) = l.
flattenPOT([ [h | t1] | t2]) = [h | joinPOT(t1, flattenPOT(t2))].

flattenScopes : list(list(scope)) -> list(scope)
flattenScopes([]) = [].
flattenScopes([ l ]) = l.
flattenScopes([ [h | t1] | t2]) = [h | joinScopes(t1, flattenScopes(t2))].

joinScopes : list(scope) * list(scope) -> list(scope)
joinScopes([], l) = l.
joinScopes(l@[_ | _], []) = l.
joinScopes([h | t], l) = [h | joinScopes(t, l)].
```

Appendix C

Benchmark Results

This chapter presents additional benchmark results not used in the evaluation chapter.

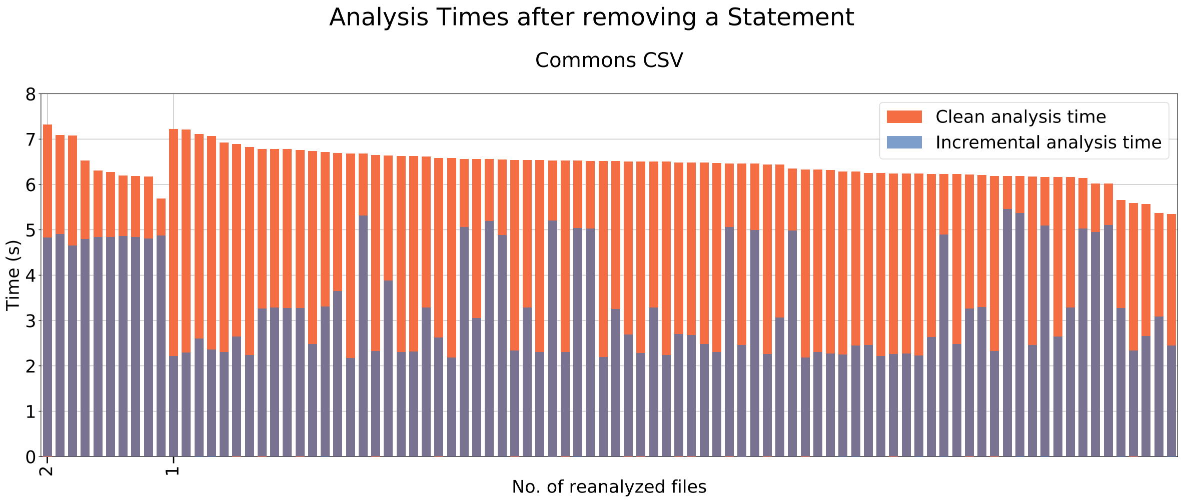


Figure C.1: A comparison of the analysis time for 100 removals of a randomly selected statement for Commons CSV (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

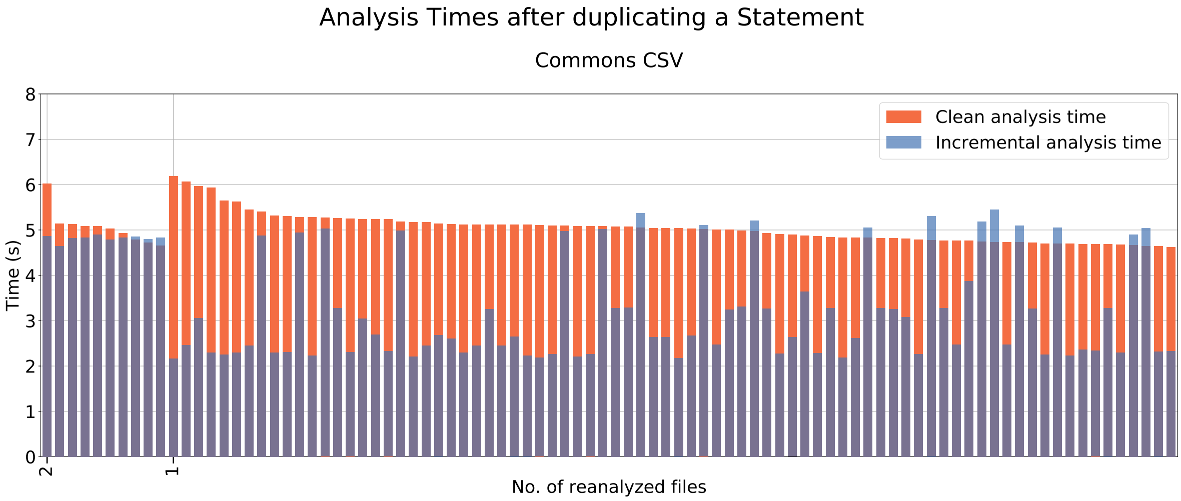


Figure C.2: A comparison of the analysis time for 100 duplications of a randomly selected statement for Commons CSV (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

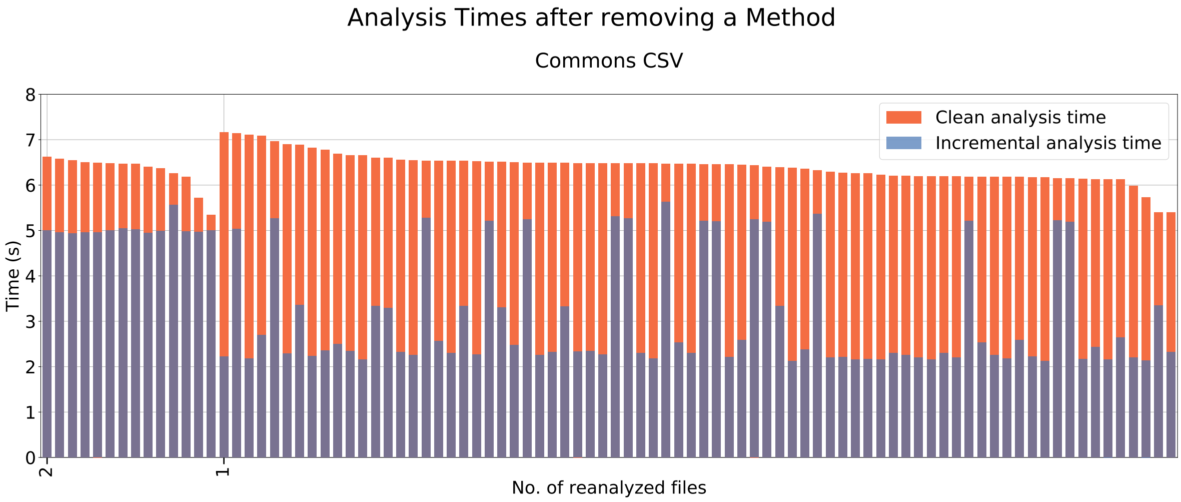


Figure C.3: A comparison of the analysis time for 100 removals of a randomly selected method for Commons CSV (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

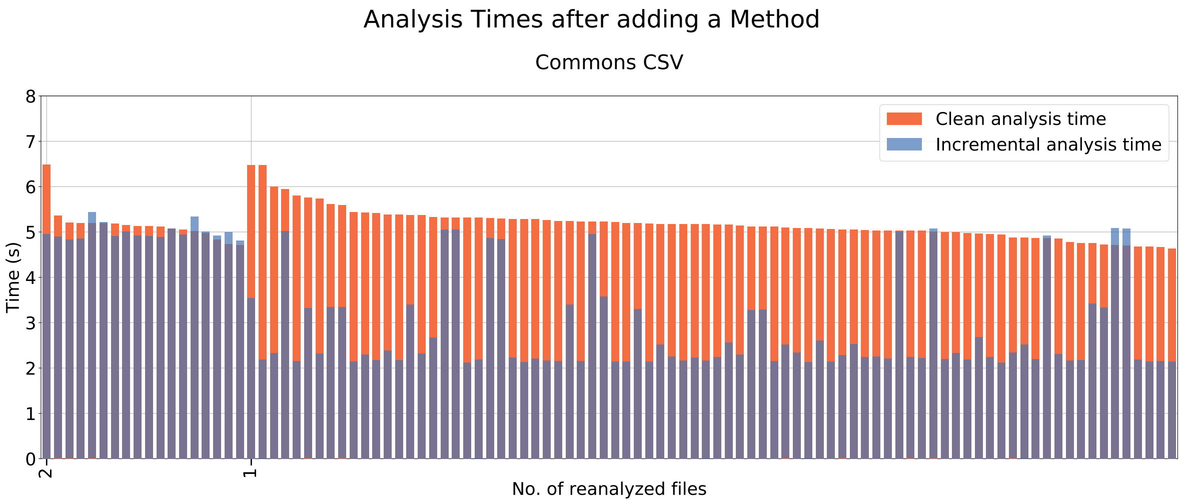


Figure C.4: A comparison of the analysis time for 100 additions of a randomly selected method for Commons CSV (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

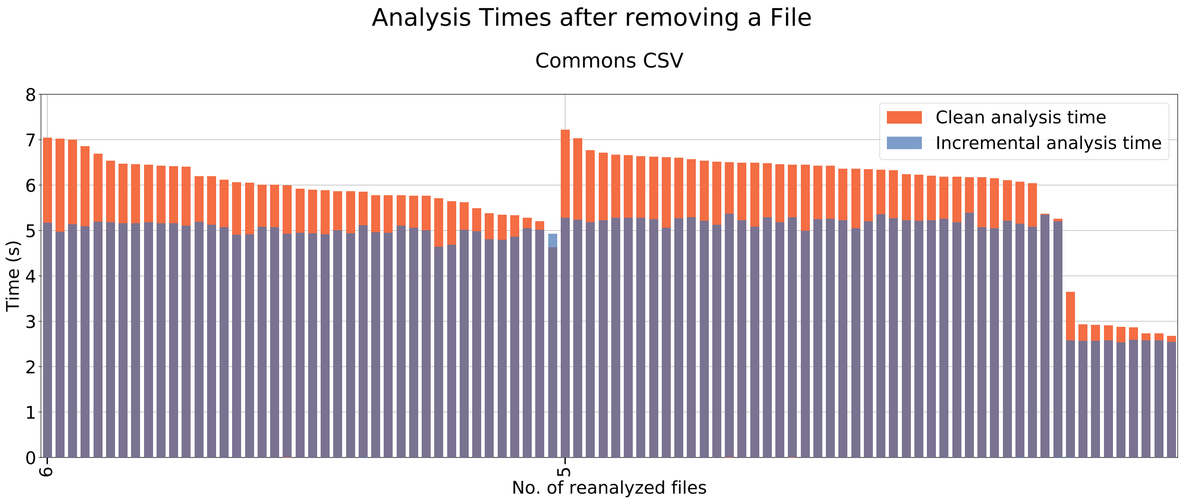


Figure C.5: A comparison of the analysis time for 100 removals of a randomly selected files for Commons CSV (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

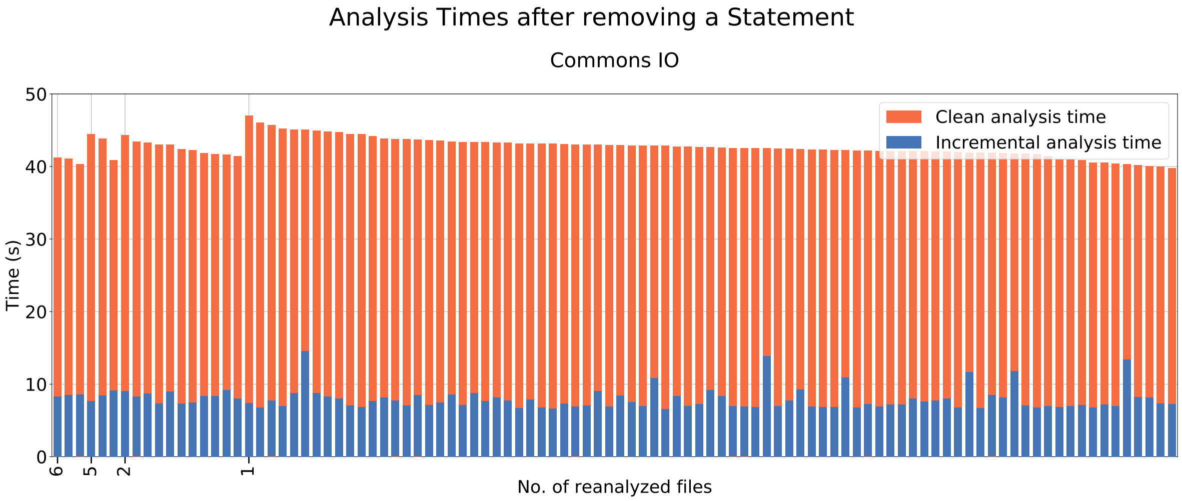


Figure C.6: A comparison of the analysis time for 100 removals of a randomly selected statement for Commons IO (large). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

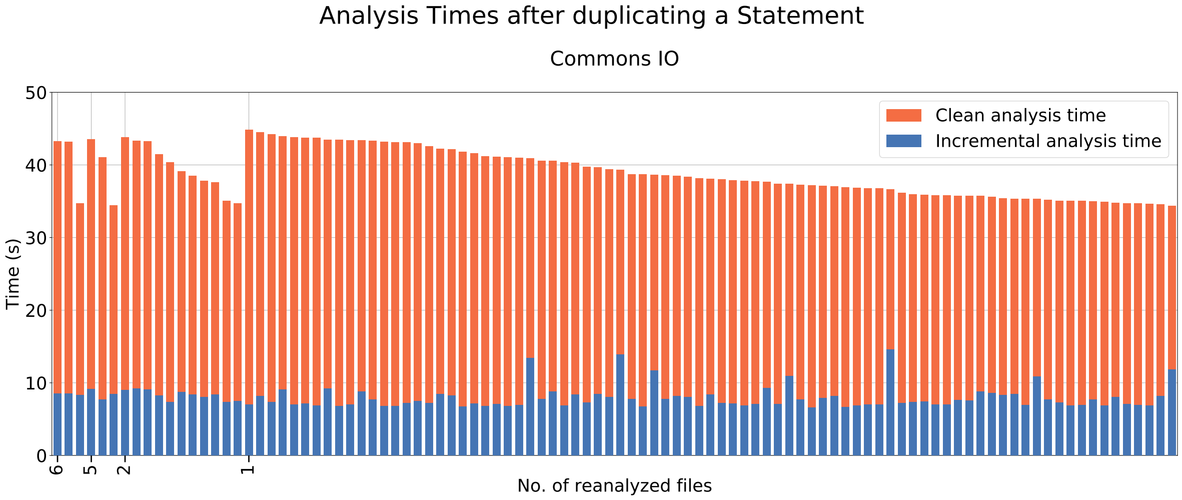


Figure C.7: A comparison of the analysis time for 100 duplications of a randomly selected statement for Commons IO (large). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

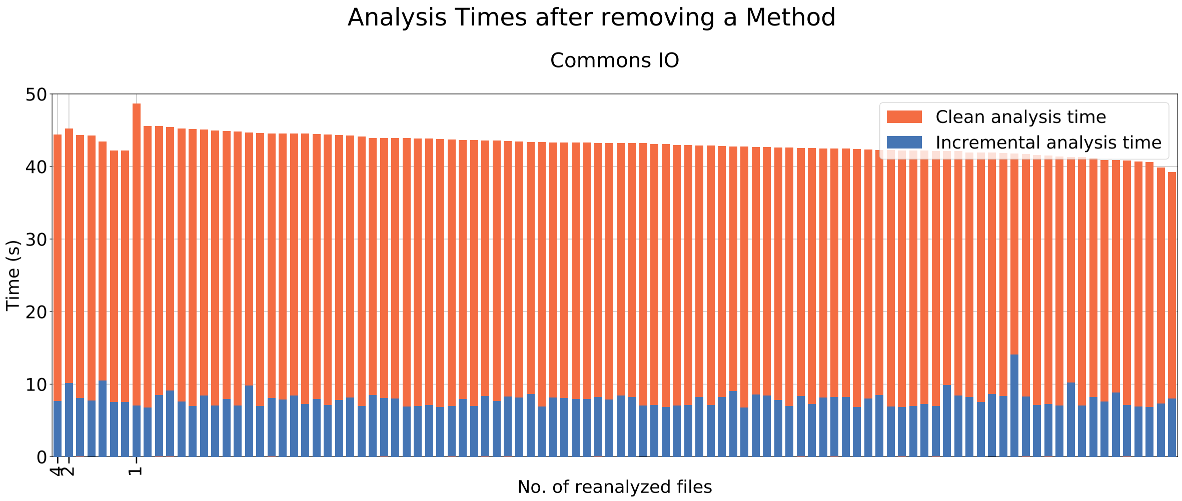


Figure C.8: A comparison of the analysis time for 100 removals of a randomly selected method for Commons IO (large). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

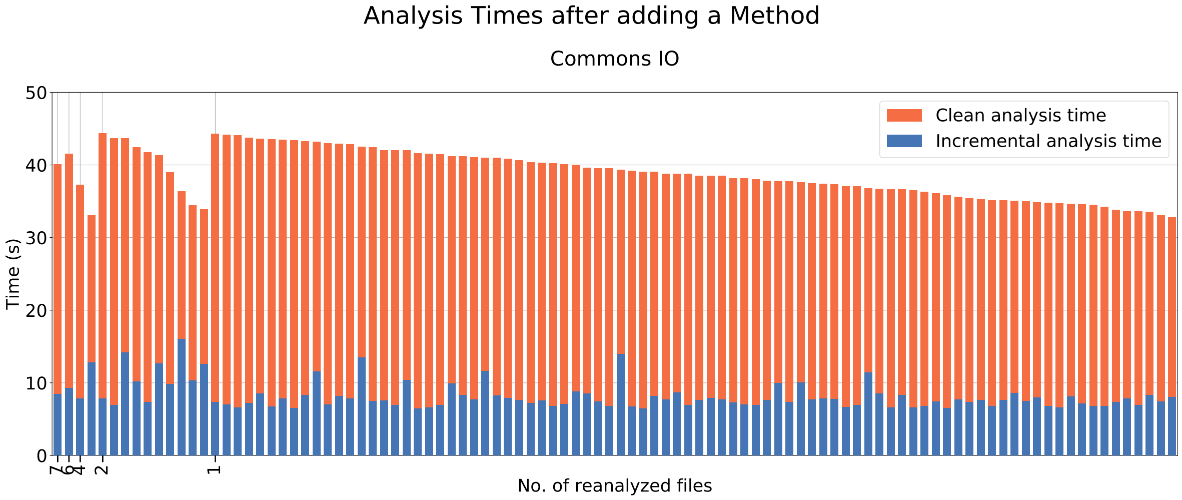


Figure C.9: A comparison of the analysis time for 100 additions of a randomly selected method for Commons IO (large). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.

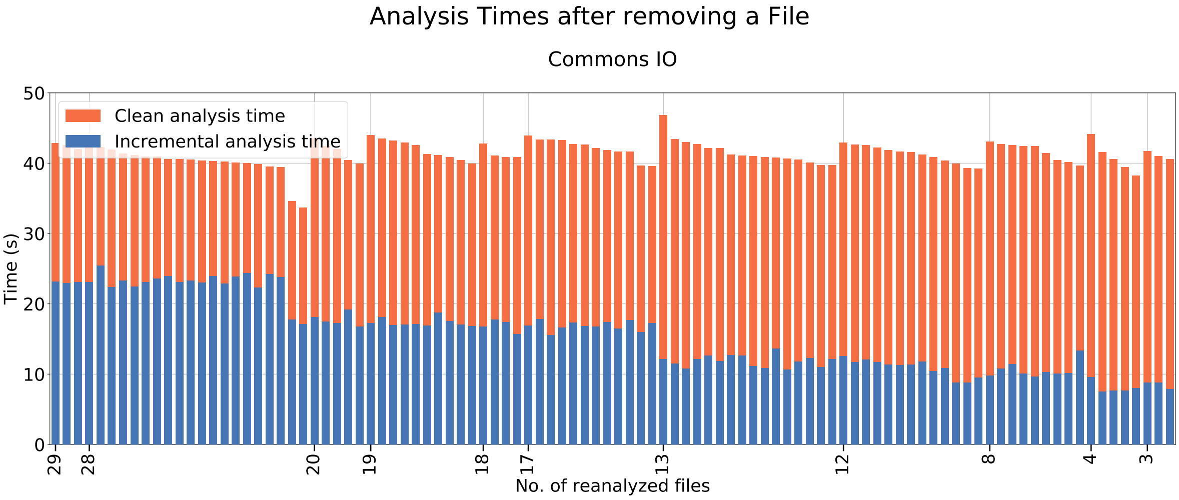


Figure C.10: A comparison of the analysis time for 100 removals of a randomly selected file for Commons IO (small). On the x-axis the number of reanalyzed files. On the y-axis, the time in seconds for reanalysis.