Language-Parametric Incremental and Parallel Name Resolution

Master’s Thesis

Gabriël Konat
Language-Parametric Incremental and Parallel Name Resolution

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gabriël Konat
born in The Hague, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

www.ewi.tudelft.nl
Abstract

Static analyses and transformations are an important part of programming and domain specific languages. For example; integrated development environments analyze programs for semantic errors such as incorrect names or types to warn the programmer about these errors. Compilers translate high-level programs into programs of another language or machine code, with the purpose of executing the program.

Programmers make frequent and small edits to code fragments during development, making it infeasible to do analysis of the entire program for every change. To cope with this, each change must only trigger re-analysis of the changed fragment and its dependencies while keeping a consistent knowledge base of the program. In other words, the analysis must be incremental. Most computers today have multiple CPU cores and the trend is that CPU performance will scale in the number of cores, not in the performance of the core itself. To make use of these cores, the analyses must also be executable in parallel.

Traditionally, an incremental and/or parallel analysis is handcrafted for each language, requiring substantial effort. In this thesis, we present a framework for performing incremental and parallel static program analyses and transformations based on a name binding specification. If such a specification is given and the framework is used, the analyses and transformations are executed incrementally and in parallel. Additionally, name resolution is also derived from a name binding specification, reducing the implementation effort even more.

To specify name binding, we present the Spoofax Name Binding Language, a declarative meta-language for the specification of name binding and scope rules, which departs from the programmatic encodings of name binding provided by regular approaches. The specification is implemented using a symbol table infrastructure that automatically traces dependencies, and a language-parametric name resolution algorithm that performs name resolution and incremental scheduling of analyses. The framework is integrated in the Spoofax Language Workbench. Several case studies have been conducted to evaluate the approach.
Thesis Committee:

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair</td>
<td>Dr. E. Visser</td>
<td>Faculty EEMCS, TU Delft</td>
</tr>
<tr>
<td>University supervisor</td>
<td>Dr. E. Visser</td>
<td>Faculty EEMCS, TU Delft</td>
</tr>
<tr>
<td>Committee member</td>
<td>Dr. K. V. Hindriks</td>
<td>Faculty EEMCS, TU Delft</td>
</tr>
<tr>
<td>Committee member</td>
<td>Dr. L. C. L. Kats</td>
<td>Cloud9</td>
</tr>
<tr>
<td>Committee member</td>
<td>Dr. M. Pinzger</td>
<td>Faculty EEMCS, TU Delft</td>
</tr>
</tbody>
</table>
Preface

About this thesis  The main matter of this thesis consists of three parts, separated into three chapters. Chapter 2 presents the Name Binding Language, a declarative DSL for specifying name binding and scope rules. Chapter 3 presents the index infrastructure, an infrastructure for supporting incremental and parallel analyses with persistence. Chapter 4 presents a language-parametric name resolution algorithm based on the concepts of the Name Binding Language. Parts of Chapter 2 and Chapter 3 have also been published in [29].

Acknowledgements  I am grateful to Eelco Visser for the feedback sessions, valuable insights, supervision and keeping me motivated during research, engineering and writing. I would like to thank Lennart Kats for helping me getting started, providing support with Spoofax development and the continuous feedback during the project. I am grateful to Guido Wachsmuth for the many interesting and inspiring discussions on name binding and other subjects and the valuable feedback on my work. Many thanks to Karl Trygve Kalleberg for also helping me getting started and the support with using Stratego and Spoofax. Last but not least, I am grateful to my family for supporting me during my studies.

Gabriël Konat
Delft, the Netherlands
November 23, 2012
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
  1.1 Research Questions 3
  1.2 Contributions 4
  1.3 Outline 4

2 Declarative Specification of Name Binding and Scope Rules 7
  2.1 Introduction 7
  2.2 Concepts 8
  2.3 Name Binding Patterns 13
  2.4 Evaluation 16

3 Persistent Infrastructure for Incremental and Parallel Name Resolution 23
  3.1 Introduction 23
  3.2 Requirements 25
  3.3 Stratego Primer 26
  3.4 Design 29
  3.5 Implementation 40
  3.6 Evaluation 43
  3.7 Design Considerations 51

4 Language-Parametric Name Resolution 53
  4.1 Introduction 53
  4.2 Requirements 54
  4.3 NBL Code Generation 54
  4.4 Name Resolution Algorithm 56
  4.5 Parallel Name Resolution 73
  4.6 Evaluation 74
## CONTENTS

5 Related work 81

6 Conclusion 89
  6.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
  6.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

7 Future work 97

Bibliography 101

A Extended Entity language specification 105

B MiniC# specification 109

C C-like language specification 113

D Green-Marl specification 117

E Mobl specification 119
List of Figures

2.1 Class declarations in C# ................................................................. 8
2.2 Declaration of definitions and references for class names in C# ........ 9
2.3 Homonym class, field, method, and variable declarations in C# .... 9
2.4 Declaration of name bindings for different namespaces in C# ....... 10
2.5 Scoped homonym method and variable declarations in C# ........... 10
2.6 Declaration of scopes for different namespaces in C# ................. 10
2.7 Nested namespace declarations in C# ........................................... 11
2.8 Declaration of name bindings for nested namespace declarations in C# ................................................................. 11
2.9 Various forms of imports in C# .................................................... 12
2.10 Declaration of import mechanisms in C# ..................................... 12
2.11 Overloaded method declarations in C# ........................................ 12
2.12 Types in name binding rules for overloaded methods in C# ........ 13
2.13 Different ways to model file scope for top-level syntax tree nodes ... 14
2.14 Declaration of the visibility of definition sites inside scopes ........ 14
2.15 foreach loop with scoped iterator variable x in C# ................... 15
2.16 Declaration of definition sites outside of their scopes ............... 15
2.17 Alternative declaration of definition sites becoming visible after their declaration ................................................................. 15
2.18 Contextual use sites in C# ............................................................ 16
2.19 Declaration of contextual use sites ............................................. 16

3.1 Overview of the infrastructure and the components that interact with it ... 25
3.2 Constructors of index keys in Stratego ......................................... 30
3.3 Constructors of partitions in Stratego ......................................... 31
3.4 Query and modification operations on the index ......................... 31
3.5 Transaction operations on the index ............................................ 34
3.6 Protocol for using the index in a concurrent way using transactions .... 35
3.7 Example of concurrent operations on the index with transactions in 2 different threads ................................................................. 35
3.8 Stratego library for tracking dependencies in the index ............... 36
3.9 Protocol for performing analyses with inconsistency detection ...... 37
3.10 Revision diagram where 2 transactions $L^4$ and $R^3$ are started in parallel. Commit of $L^4$ causes an inconsistency during $R^3$, resulting in rescheduling of $R$. Numbers denote revision numbers
3.11 Stratego library for incremental analysis in the index
3.12 Protocol for performing incremental analysis
3.13 Revision diagram with where 2 transactions $X^1$ and $Y^2$ are started sequentially. Commit of $Y^2$ causes a change in $X$, resulting in a re-analysis
3.14 UML class diagram of infrastructure
3.15 Benchmark results for the clear operation
3.16 Average benchmark results for the clear operation
3.17 Implementation 1 benchmark results for the add operation
3.18 Average benchmark results for the add operation
3.19 Average benchmark results for the get operation
3.20 Average benchmark results for the get-children operation
3.21 Average benchmark results for the get-key-partitions operation
3.22 Average benchmark results for the persist operation
4.1 Binding rule for C# class
4.2 Aster representation of binding rule
4.3 Overview of the stages of the name resolution algorithm
4.4 Example of an AST with named vertical scoping. A class scopes fields and methods. The rectangular area is a vertical named scope from Class "String" Rounded rectangles are terms
4.5 Example of an AST with unnamed vertical scoping. A statement block scopes local variables. The rectangular area is a vertical unnamed scope with generated name "anon_1"
4.6 Example of an AST with horizontal scoping. A local variable creates a new unique scope for itself and subsequent sibling definitions. The first dotted rectangular area is a horizontal scope with generated name "subseq_1". The second dotted rectangular area is a horizontal scope with generated name "subseq_2"
4.7 Example of an AST with vertical and horizontal scoping combined. A method vertically scopes local variables, local variables horizontally scope itself and subsequent definitions. The outer rectangular area is a vertical scope from Method "M". The inner dotted rectangular areas are horizontal scopes for their variables
4.8 Example of an AST with for each term a list of namespaces mapping to the active path at that term
4.9 Example of an AST with duplicate definitions. Both fields can be accessed through identifier $a$
4.10 Example of an AST with duplicate definitions. Fields in $A^1$ and $A^2$ are distinguishable by using a unique identifier
4.11 AST parents and siblings in Aster
4.12 Benchmark results for baseline and increased number of definitions
4.13 Benchmark results for baseline and increased number of references
4.14 Benchmark results for baseline and no dependencies
4.15 Benchmark results for parallel and sequential baseline
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.16</td>
<td>Benchmark results for parallel and sequential baseline with increased number</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>of definitions.</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>Benchmark results for parallel and sequential baseline with increased number</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>of references.</td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>Syntax definition in SDF for the extended Entity language. The names in</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>annotations are term constructors.</td>
<td></td>
</tr>
<tr>
<td>A.2</td>
<td>Name binding and scoping definition in NBL for the extended Entity language.</td>
<td>107</td>
</tr>
<tr>
<td>B.1</td>
<td>Syntax definition in SDF for MiniC#. The names in annotations are term</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>constructors.</td>
<td></td>
</tr>
<tr>
<td>B.2</td>
<td>Name binding and scoping definition in NBL for MiniC#.</td>
<td>111</td>
</tr>
<tr>
<td>C.1</td>
<td>Syntax definition in SDF for the extended C-like language. The names in</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>annotations are term constructors.</td>
<td></td>
</tr>
<tr>
<td>C.2</td>
<td>Name binding and scoping definition in NBL for the C-like language.</td>
<td>115</td>
</tr>
<tr>
<td>D.1</td>
<td>Name binding and scoping definition in NBL for Green-Marl.</td>
<td>118</td>
</tr>
<tr>
<td>E.1</td>
<td>Name binding and scoping definition in NBL for Mobl, part 1.</td>
<td>120</td>
</tr>
<tr>
<td>E.2</td>
<td>Name binding and scoping definition in NBL for Mobl, part 2.</td>
<td>121</td>
</tr>
<tr>
<td>E.3</td>
<td>Name binding and scoping definition in NBL for Mobl, part 3.</td>
<td>122</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Programming languages are artificial languages designed to create programs. A program is an encoding of instructions that can be executed on a machine to perform a certain task. Typically, programs are not created directly by humans, but are constructed from multiple source files. Source files are (usually textual) representations of programs written in a programming language.

In the field of software engineering, engineers (or programmers) use programming languages to develop programs that form software. Since software need to be constructed, maintained and extended, engineers must be able to understand and reason about programs and their representations. To help them with this task, there exist integrated development environments (IDEs). IDEs contain many facilities to aid the development of software systems, one of which is the source code editor. Source code editors provide editor services to facilitate understanding, reasoning and editing of source code. Common editor services are:

- Error checking that verifies source files for errors and annotates the editor with error markers at the position errors occur.
- Content completion to predict what word or sentence a programmer wants to write and automatically try to complete this word or sentence.
- Reference resolution to move the editor from a reference to the point where the reference refers to; its definition.

Editors and editor services are specific to a programming language. Each programming language requires a different editor, error checking, content completion, reference resolution and so on. Since there are thousands of programming languages, it is not feasible to support all programming languages in an IDE. To overcome this, many IDEs have a plugin architecture that supports extending the IDE with new editors and editor services for other languages. However, this entails reimplementation of the editor, editor services and other facilities provided by IDEs. Usually some libraries and abstractions are provided to reduce reimplementation, such as a parser to parse source files. But for many facilities, no abstractions or libraries are made.

Language workbenches aim to reduce the implementation effort of constructing and maintaining an IDE for a language. They provide abstractions and libraries for defining languages, editor services and other tooling. Spoofax [24, 20] is a language
workbench that generates plugins for the Eclipse IDE from a language definition, editor service definitions and transformations. The syntax of languages are defined in Syntax Definition Formalism [33] (SDF), a domain specific language for syntax definitions. From SDF, parse tables are generated that are used by a parser to parse textual source files into abstract syntax trees (ASTs). In contrast to a general purpose language (GPL), a domain specific language (DSL) is dedicated to a specific problem domain. DSLs are less expressive than GPLs, but provide abstractions in the problem domain to make reasoning, constructing and maintaining programs easier. SDF is also declarative, meaning that a language engineer does not need to think about how to parse a source file but rather what should be parsed. This improves evolution, maintainability, and compositionality of syntax definitions. To define logic for editor services, Stratego [4], a language for defining (program) transformations based on rewriting strategies, is used. Transformations are specified using rewrite rules that operate on ASTs produced by the parser.

Xtext [9] is also a language workbench targeting the Eclipse IDE. The Xtext Grammar Language is used to specify syntax for languages. The logic for editor services is implemented using Java, or Xtend, a Java-like language with domain specific extensions.

To provide editor services, source code needs to be semantically analysed to gain understanding of the program. In the case of Spoofax, semantic analysis is expressed in Stratego. In Xtext, it is expressed in Xtend or Java. When programmers edit source code, many small edits to the program are made. If the program changes, the semantics of the program could change, so the source code needs to be reanalysed. However, finding out which semantics have changed is not trivial. Changes in one source file might change the semantics of another source file, or all source files. A naive approach is to do semantic analysis of the whole program, which guarantees that all changes are found. Obviously, this approach does not scale up to many source files and many programs. To scale, semantic analysis needs to be incremental. In other words, only the changes to the program should be analysed, not the entire program. Semantic analysis should also be executed in parallel, to make use of the extra computing power in every modern computer.

A large part of semantic analysis is the name binding analysis, also called name resolution. The essence of name binding is establishing relations between definitions and references in programs by use of names or identifiers. Almost all editor services require name resolution in some way, therefore each language that uses names or identifiers requires an algorithm that performs name resolution. Name resolution is typically implemented manually in an ad hoc manner for each language. The reason for this is that IDEs and language workbenches do not provide (enough) abstractions for name resolution. Spoofax has a name binding library. However, it does not fully capture the domain of name resolution, it is not declarative and requires programmatic name binding encodings. The Xtext Grammar language contains abstractions for name binding, but non trivial name binding is expressed programmatically in Xtend or Java. Other existing approaches suffer from the same problems.

To summarize, semantic analysis needs to be incremental and parallel to scale, and the implementation effort of name resolution needs to be reduced. We think that providing abstractions in the name binding domain can solve these problems. DSLs are the typical way of abstracting in Spoofax and other grammarware approaches. In
fact, we would like to do something similar as SDF, but in the name binding domain; a declarative DSL for name binding.

An implementation is required that performs the actual name resolution based on the name binding DSL, like the parser for SDF. This is a generic name resolution algorithm that operates on the abstractions from the DSL. The algorithm should be incremental and executable in parallel. Since the algorithm is generic it can be reused between languages, providing a massive implementation effort reduction. We also think it would improve evolution, maintainability, and compositionality of name binding definitions, in the same way as SDF does for syntax definitions. The dependency of editor services on name resolution could be exploited to make the evaluation of those editor services incremental and parallel as well.

With the results of this thesis we hope to a) reduce the implementation effort of creating and maintaining languages, b) improve scalability with incremental and parallel evaluation and c) improve evolution, maintainability, and compositionality of name resolution

1.1 Research Questions

The research in this thesis is driven by the following research questions.

<table>
<thead>
<tr>
<th>Research Question A:</th>
<th>What are the fundamental concepts of name binding?</th>
</tr>
</thead>
</table>

First of all, we would like to find the fundamental concepts of name binding. These fundamental concepts can then form the basis of a declarative DSL, abstracting over the domain of name binding. Finding concepts which have wide coverage in the problem domain and are also easy to understand is not easy task. The number of fundamental concepts also needs to be minimized while trying to cover the problem domain to reduce complexity.

<table>
<thead>
<tr>
<th>Research Question B:</th>
<th>What are the formal semantics of the fundamental concepts of name binding?</th>
</tr>
</thead>
</table>

After we know what the fundamental concepts of name binding are, we would like to know the semantics behind them. The semantics will form the basis for a name resolution algorithm that operates on these fundamental concepts. It may also be possible to formalize the semantics into a formal semantics for the fundamental concepts.

<table>
<thead>
<tr>
<th>Research Question C:</th>
<th>How can name resolution and name dependent analyses be executed incrementally and in parallel?</th>
</tr>
</thead>
</table>

Some support is required to execute analyses in parallel and incrementally. For incremental analysis only changes are analysed, some kind of persistent data structure is required to keep track of dependencies. For parallel analysis, safe concurrent access to shared memory is required.
1. **Introduction**

**Research Question D:** How can incremental and parallel execution for name resolution and name dependent analyses be automatically derived using name binding definitions?

This question relates to the fact that many analyses depend on name binding information. We would like to know how this dependency can be exploited to automatically make those analyses incremental and parallel. Automatically deriving an incremental and parallel analysis would provide a big reduction in implementation effort.

1.2 **Contributions**

There are four main contributions in this thesis.

**Name Binding Language** The Name Binding Language (NBL) is a declarative DSL in the domain of name binding. Instead of implementing name resolution, language engineers create name binding specifications in NBL. Specifications consist of namespaces and definition, reference, scopes and import rules. The approach is evaluated by applying NBL to several case study languages.

**Index Infrastructure** The index infrastructure is an infrastructure for incremental and parallel name resolution based on NBL specifications, integrated in the Spoofax language workbench. It is a symbol table supporting fast hash lookups, persistence and thread-safe access using transactions. Protocols are provided for using the index in Stratego in a thread-safe and incremental way. The performance of the infrastructure is evaluated using a set of benchmarks.

**Language-Parametric Name Resolution** We present a language-parametric incremental and parallel name resolution algorithm. The language-parametric algorithm is parameterized by NBL definitions and uses the index infrastructure for persistence, incremental and parallel evaluation. Name-dependent analyses such as error checking and type analysis are incrementalized and parallelized as well. Case studies are used to evaluate the performance of the algorithm.

**Spoofax Integration** The previous contributions are integrated in the unstable branch of the Spoofax language workbench. It provides quicker language development and increased scalability for languages created with NBL, the index infrastructure and the generic name resolution algorithm.

1.3 **Outline**

The thesis is outlined as follows. The concepts of NBL are described by example and evaluated in Chapter[2] The requirements, design, implementation and evaluation of the index infrastructure are outlined in Chapter[3] In Chapter[4] the semantics, implementation and evaluation of the language-parametric name resolution are described.
Related work in name binding, incremental name resolution and parallel name resolution are shown in Chapter 5. A conclusion is given in Chapter 6 with a summary and conclusions to the research questions. Finally, future work is listed in Chapter 7.
Chapter 2

Declarative Specification of Name Binding and Scope Rules

2.1 Introduction

Software language engineering is concerned with linguistic abstraction, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity.

Declarative languages are of particular interest since they enable language engineers to focus on the What? instead of the How?. Syntax definitions are a prominent example. With declarative formalisms such as EBNF and SDF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declaratively is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [25].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language processors, not to specify language aspects. They lack domain concepts for these aspects and focus on the How?. That is a problem since (1) it entails overhead in encoding concepts in a programming language and (2) the encoding obscures the intention; understanding the definition requires decoding.

In this thesis, we are specifically concerned with name binding and scope rules. Name binding is concerned with the relation between definitions and references of identifiers in textual software languages, including scope rules that govern these relations. In language processors, it is crucial to make information about definitions available at the references. Therefore, traditional language processing approaches provide programmatic abstractions for name binding. These abstractions are centered around tree traversal and information propagation from definitions to references. Typically, they are not specifically addressing name binding, but can also be used for other
language processing tasks such as compilation and interpretation.

Name binding plays a role in multiple language engineering processes, including editor services such as reference resolution, code completion, refactorings, type checking, and compilation. The different processes need different information about definitions. For example, name resolution tries to find one definition, while code completion needs to determine all possible references in a certain place. The different requirements lead either to multiple re-implementations of name binding rules for each of these purposes, or to non-trivial, manual weaving into a single implementation supporting all purposes. This results in code duplication with as result errors, inconsistencies, and increased maintenance effort.

In this chapter we introduce the Name Binding Language (NBL), a declarative DSL in the domain of name resolution. Section 2.2 introduces the concepts of NBL by giving example programs in C# and their model in NBL. Section 2.3 highlights name binding patterns that are regularly found in languages and how to model them in NBL. Finally, in Section 2.4 we evaluate NBL and discuss its coverage and limitations.

2.2 Concepts

In this section we introduce the Spoofax Naming Binding Language (NBL) illustrated with examples drawn from the specification of name binding for a subset of C# called MiniC#. Figure B.1 defines the syntax of the subset in SDF. The subset is by no means complete; it has been selected to model representative features of name binding rules in programming and domain-specific languages. In the following subsections we discuss the following fundamental concepts of name binding: definition and use sites, namespaces, scopes, and imports. For each concept we give a general definition, illustrate it with an example in C#, and then we show how the concept can be modeled in NBL.

2.2.1 Definitions and References

The essence of name binding is establishing relations between a definition that binds a name and a reference that uses that name. Name binding is typically defined programmatically through a name resolution algorithm that connects references to definitions. A definition site is the location of a definition in a program. In many cases, definition sites are required to be unique, that is, there should be exactly one definition site for each name. However, there are cases where definition sites are allowed to be non-unique.

Example Figure 2.1 contains class definitions in C#. Each class definition binds the name of a class. Thus, we have definition sites for C1, C2, and C3. Base class specifications are references to these definition sites. In the example, we have references to C1 as the base class of C2 and C2 as the base class of C3. (Thus, C2 is a sub-class of / inherits from C1.) There is no reference to C3. The definition sites for C1 and C2 are unique. By contrast, there are two definition sites for C3, defining parts...
of the same class $C_3$. Thus, these definition sites are non-unique. This is correct in C#,
since regular class definitions are required to be unique, while partial class definitions
are allowed to be non-unique.

Abstract Syntax Terms In Spoofax, abstract syntax trees (ASTs) are represented
using first-order terms. Terms consist of strings ("x"), lists of terms ("[x", "y"]), and
constructor applications (ClassType("C1")) for labelled tree nodes with a fixed number
of children. Annotations in grammar productions (Figure B.1) define the constructors
to be used in AST construction. For example, Class(Partial(), "C3", Base("C2"),
[]) is the representation of the first partial class in Figure 2.1. A term pattern is a term
that may contain variables (x) and wildcards (_).

Model A specification in the name binding language consists of a collection of rules
of the form pattern : clause*, where pattern is a term pattern and clause* is a list
of name binding declarations about the language construct that matches with pattern.
Figure 2.2 shows a declaration of the definitions and references for class names in
C#. The first two rules declare class definition sites for class names. Their patterns
distinguish regular (non-partial) and partial class declarations. While non-partial class
declarations are unique definition sites, partial class declarations are non-unique def-
inition sites. The third rule declares that the term pattern Base(c) is a reference to a
class with name c. Thus, the : C1 in Figure 2.1 is a reference to class $C_1$. Similarly,
the second rule declares a class type as a reference to a class.

2.2.2 Namespaces

Definitions and references declare relations between named program elements and
their uses. Languages typically distinguish several names\-\-spaces, i.e. different kinds
of names, such that an occurrence of a name in one namespace is not related to an
occurrence of that same name in another.

Example Figure 2.3 shows several definitions for
the same name $x$, but of different kinds, namely a
class, a field, a method, and a variable. Each of these
kinds has its own namespace in C#, and each of these
namespaces has its own name $x$. This enables us
to distinguish the definition sites of class $x$, field $x$,
method $x$, and variable $x$, which are all unique.

Model We declared definitions and references for the namespace Class already in
the previous example. Figure 2.4 extends that declaration covering also the name-
namespaces Class Field Method Variable
rules
Field(_, f) : defines unique Field f
Method(_, m, _, _) : defines unique Method m
Call(m, _) : refers to Method m
Var(_, v) : defines unique Variable v
VarRef(v) : refers to Variable v otherwise refers to Field v

Figure 2.4: Declaration of name bindings for different namespaces in C#.

spaces Field, Method, and Variable. Note that it is required to declare namespaces to ensure the consistency of name binding rules. Definition sites are bound to a single namespace \( \text{defines Class } c \), but use sites are not. For example, a variable in an expression might either refer to a variable, or to a field, which is modeled in the last rule. In our example, this means that variable declarations hide field declarations, because variables are resolved to variables, if possible. Thus, both \( x \) in the assignment in Figure 2.3 refer to the variable \( x \).

2.2.3 Scopes

Scopes restrict the visibility of definition sites. A named scope is the definition site for a name which scopes other definition sites. By contrast, an anonymous scope does not define a name. Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes.

Example Figure 2.5 includes two definition sites for a method \( m \). These definition sites are not distinguishable by their namespace Method and their name \( m \), but, they are distinguishable by the scope they are in. The first definition site resides in class \( C \), the second one in class \( D \). In C#, class declarations scope method declarations. They introduce named scopes, because class declarations are definition sites for class names. The listing also contains three definition sites for a variable \( x \). Again, these are distinguishable by their scope. In C#, method declarations and blocks, scope variable declarations. Method declarations are named scopes, blocks are anonymous scopes. The first definition site resides in method \( m \) in class \( C \), the second one in method \( m \) class

\[
\begin{align*}
\text{class } C & \{ \\
& \quad \text{void } m() \{ \text{ int } x; \} \\
\} \\
\text{class } D & \{ \\
& \quad \text{void } m() \{ \\
& \quad \quad \text{int } x; \\
& \quad \quad \text{int } y; \\
& \quad \quad \{ \text{ int } x; x = y + 1; \} \\
& \quad \quad \text{x } = y + 1; \\
& \} \\
\}
\end{align*}
\]

Figure 2.5: Scoped homonym method and variable declarations in C#.

rules
Class(NonPartial(), c, _, _) : \text{defines unique Class } c \\
scopes Field, Method \\
Class(Partial(), c, _, _) : \text{defines non-unique Class } c \\
scopes Field, Method \\
Method(_, m, _, _) : \text{defines unique Method } m \\
scopes Variable \\
Block(_:): \text{scopes Variable }

Figure 2.6: Declaration of scopes for different namespaces in C#.
namespaces Namespace
rules
  Namespace(n, _):
    defines Namespace n
    scopes Namespace, Class

Figure 2.7: Nested namespace declarations in C#.

Figure 2.8: Declaration of name bindings for nested namespace declarations in C#.

in class D, and the last one in a nameless block inside method m in class D. In the assignment inside the block (line 9), x refers to the variable declaration in the same block, while the x in the outer assignment (line 10) refers to the variable declaration outside the block. In both assignments, y refers to the variable declaration in the outer scope, because the block does not contain a definition site for y.

Model The scopes ns clause in NBL declares a construct to be a scope for namespace ns. Figure 2.6 declares scopes for fields, methods, and variables. Named scopes are declared at definition sites. Anonymous scopes are declared similarly, but lack a defines clause.

Namespaces as Language Concepts C# has a notion of ‘namespaces’. It is important to distinguish these namespaces as a language concept from namespaces as a naming concept, which group names of different kinds of declarations. Specifically, in C#, namespace declarations are top-level scopes for class declarations. Namespace declarations can be nested. Figure 2.7 declares a top-level namespace N, scoping a class declaration N and an inner namespace declaration N. The inner namespace declaration scopes another class declaration N. The definition sites of the namespace name N and the class name N are distinguishable, because they belong to different namespaces (as a naming concept). The two definition sites of namespace name N are distinguishable by scope. The outer namespace declaration scopes the inner one. Also, the definition sites of the class name N are distinguishable by scope. The first one is scoped by the outer namespace declaration, while the second one is scoped by both namespace declarations.

Model The names of C# namespace declarations are distinguishable from names of classes, fields, etc. As declared in Figure 2.8 their names belong to the Namespace namespace. The name binding rules for definition sites of names of this namespace models the scoping nature of C# namespace declarations.

2.2.4 Imports

An import introduces definitions from another scope into the current scope, either under the same name or under a new name. An import that imports all definitions can be transitive.

Example Figure 2.9 shows different kinds of imports in C#. First, a using directive imports type declarations from namespace N. Second, another using directive imports
2. Declarative Specification of Name Binding and Scope Rules

```csharp
using N;
namespace M {
    class C {
        int f;
    }
}
namespace O {
    using D = M.C;
    class E:D {
        void m() {}
    }
    class F:E { }
}
```

Figure 2.9: Various forms of imports in C#.

class C from namespace M into namespace O under a new name D. Finally, classes E and F import fields and methods from their base classes. These imports are transitive, that is, F imports fields and methods from E and D.

Model Figure 2.10 shows name binding rules for import mechanisms in C#. The first rule handles using declarations, which import all classes from the namespace, the qualified name qname resolves to. The second rule models aliases, which either import a namespace or a class under a new name, depending on the resolution of qname. The last rule models inheritance, where fields and methods are imported transitively from the base classes.

2.2.5 Types

So far, we discussed names, namespaces, and scopes to distinguish definition sites for the same name. Types also play a role in name resolution and can be used to distinguish definition sites for a name or to find corresponding definition sites for a use site.

Example Figure 2.11 shows a number of overloaded method declarations. These share the same name m, namespace method, and scope class C. But we can distinguish them by the types of their parameters. Furthermore, all method calls inside method x can be uniquely resolved to one of these methods by taking the argument types of the calls into account.

Model Figure 2.12 includes type information into name binding rules for fields, methods, and variables. Definition sites might have types. In the simplest case, the type is part of the declaration. In the example, this holds for parameters. For method
calls, the type of the definition site for a method name depends on the types of the parameters. A type system is needed to connect the type of a single parameter, as declared in the rule for parameters, and the type of a list of parameters, as required in the rule for methods. We will discuss the influence of a type system and the interaction between name and type analysis later. For now, we assume that the type of a list of parameters is a list of types of these parameters.

Type information is also needed to resolve method calls to possibly overloaded methods. The refers to clause for method calls therefore requires the corresponding definition site to match the type of the arguments. Again, we omit the details how this type can be determined. We also do not consider subtyping here. Method calls and corresponding method declarations need to have the same argument and parameter types.

2.3 Name Binding Patterns

We now identify typical name binding patterns. These patterns are formed by scopes, definition sites and their visibility, and use sites referencing these definition sites. We explain each pattern first and give an example in C# next. Afterwards, we show how the example can be modelled with declarative name binding rules.

2.3.1 Unscoped Definition Sites

In the simplest case, definition sites are not scoped and globally visible.

Example In C#, namespace and class declarations (as well as any other type declaration) can be unscoped. They are globally visible across file boundaries. For example, the classes C1, C2, and C3 in Figure 2.1 are globally visible. In Figure 2.3 only the outer namespace N is globally visible.

In contrast to C#, C++ has file scopes and all top-level declarations are only visible in a file. To share global declarations, each file has to repeat the declaration and mark it as extern. This is typically achieved by importing a shared header file.

Model We consider any definition site, which is not scoped by another definition site or by an anonymous scope to be in global scope. These definition sites are visible over file boundaries. File scope can be modelled with a scoping rule in two different ways.
2. Declarative Specification of Name Binding and Scope Rules

Both are illustrated in Figure 2.13. The first rule declares the top-level node of abstract syntax trees as a scope for all namespaces which can have top-level declarations. This scope will be anonymous, because the top-level node cannot be a definition site (otherwise this definition site would be globally visible). The second rule declares a tuple consisting of file name and the abstract syntax tree as a scope. This tuple will be considered a definition site for the file name. Thus, the scope will be named after the file.

2.3.2 Definition Sites inside their Scopes

Typically, definition sites reside inside the scopes where they are visible. Such definition sites can either be visible only after their declaration, or everywhere in their surrounding scope.

Example In C#, namespace members such as nested namespace declarations and class declarations are visible in their surrounding scope. The same holds for class members. In contrast, variable declarations inside a method scope become visible only after their declaration.

Model Scoped definition sites are by default visible in the complete scope. Optionally, this can be stated explicitly in defines clauses. Figure 2.14 illustrates this for namespace declarations. The second rule in this listing shows how to model definition sites which become visible only after their declaration.

2.3.3 Definition Sites outside their Scopes

Some declarations include not only the definition site for a name, but also the scope for this definition site. In such declarations, the definition site resides outside its scope.

Example Let expressions are a classical example for definition sites outside their scopes. In C#, foreach statements declare iterator variables, which are visible in embedded statement. Figure 2.15 shows a method with a parameter \( x \), followed by a

---

**rules**

CompilationUnit(_, _):

scopes Namespace, Class

(f, CompilationUnit(_, _)):

defines unique File \( f \)

scopes Namespace, Class

---

Figure 2.13: Different ways to model file scope for top-level syntax tree nodes.

**rules**

Namespace(n, _):

defines non-unique Namespace \( n \) in surrounding scope

Var(t, v):

defines unique Variable \( v \) of type \( t \) in subsequent scope

---

Figure 2.14: Declaration of the visibility of definition sites inside scopes.
class C {
    void m(int[] x) {
        foreach (int x in x)
            System.Console.WriteLine(x);
    }
}

Figure 2.15: foreach loop with scoped iterator variable x in C#.

rules
Foreach(t, v, exp, body):
    defines unique Variable v of type t in body

Figure 2.16: Declaration of definition sites outside of their scopes.

foreach statement with an iterator variable of the same name. This is considered incorrect in C#, because definition sites for variable names in inner scopes collide with definition sites of the same name in outer scopes. However, the use sites can still be resolved based on the scopes of the definition sites. The use site for x inside the loop refers to the iterator variable, while the x in the collection expression refers to the parameter.

Model Figure 2.16 shows the name binding rule for foreach loops, stating the scope of the variable explicitly. Note that definition sites which become visible after their declaration are a special case of this pattern. Figure 2.17 illustrates how this can be modelled in the same way as the foreach loop. The first rule assumes a nested representation of statement sequences, while the second rule assumes a list of statements.

2.3.4 Contextual Use Sites

Definition sites can be referenced by use sites outside of their scopes. These use sites appear in a context which determines the scope into which they refer. This context can either be a direct reference to this scope, or has a type which determines the scope.

Example In C#, namespace members can be imported into other namespaces. Figure 2.7 shows a class N in a nested namespace. In Figure 2.18 this class is imported. The using directive refers to the class with a qualified name. The first part of this name refers to the outer namespace N. It is the context of the second part, which refers to the inner namespace N. The second part is then the context for the last part of the qualified name, which refers to the class N inside the inner namespace.

Figure 2.18 also illustrates use sites in a type-based context. In method m in class D, a field f is accessed. The corresponding definition site is outside the scope of the

rules
Seq(Var(t, v), stmts):
    defines unique Variable v of type t in stmts
[Var(t, v) | stmts]:
    defines unique Variable v of type t in stmts

Figure 2.17: Alternative declaration of definition sites becoming visible after their declaration.
2. **Declarative Specification of Name Binding and Scope Rules**

```csharp
using N.N.N;

namespace N' {
    class C {
        C f;
        void m(C p) { }
    }
    class D {
        void m(C p) {
            p.m(p.f);
        }
    }
}
```

Figure 2.18: Contextual use sites in C#.

```csharp
using N.N.N;

namespace N' {
    class C {
        C f;
        void m(C p) { }
    }
    class D {
        void m(C p) {
            p.m(p.f);
        }
    }
}
```

Figure 2.19: Declaration of contextual use sites.

method in class C. But this scope is given by the type of p, which is the context for the field access. Similarly, the method call is resolved to method m in class C because of the type of p.

Model Figure 2.19 illustrates how to model contextual use sites. The scope of the declaration site corresponding to a use site can be modelled in refers to clauses. This scope needs to be determined from the context of the use site. The first rule resolves the context of a qualified name part to a namespace ns and declares the use site to refer either to a namespace or to a class in ns. The remaining rules declare use sites for field access and method calls. They determine the type of the context, which needs to be a class type. A field access refers to a field in that class. Similarly, a method call refers to a method with the right parameter types in that class.

2.4 **Evaluation**

We evaluated NBL by constructing name binding definitions for several case study languages in NBL. We started out with applying NBL to simple languages and gradually increased the complexity. Each language has interesting name binding properties that occur in real-life languages as well. The goal of the evaluation is to increase confidence in the expressiveness and coverage of NBL. During evaluation we found
language concepts that could not be completely described in NBL. The reason and possible solutions for these issues are discussed.

A brief description, NBL definition and evaluation is shown for each case study language in the following subsections. In addition, NBL was used in the lab of the 2012-2013 Compiler Construction course given at the Technical University of Delft. A formal evaluation with students was not conducted, but some interesting anecdotal evidence is discussed instead.

2.4.1 Entity

The entity language is the default example language in Spoofax. It consists of entities which contain typed properties, forming a simple data model. For the evaluation, the language is extended with functions, imports and simple aspects. The syntax and name binding definition can be found in Figure A.1 and Figure A.2 respectively. Functions contain simple statements such as assignments and other function calls. Function overloading is disallowed. Imports behave like Java wildcard imports, importing one or all entities from a module. Aspects allow extra code to be generated before or after an assignment, they were added to test compilation with NBL. We were able to construct the complete name binding definition for the Entity language in NBL, including entities and imports which are similar to Java classes and package imports.

2.4.2 MiniC#

MiniC# is a simplified version if C#, which is used as a running example in this section. The full syntax and name binding definition can be found in Figure B.1 and Figure B.2 respectively. Most language concepts from MiniC# are similar to the Entity language, but with slightly more complexity. For example, classes can be inherited from, which is modeled using transitive imports. Partial classes are non unique classes where method and field definitions from all partial classes with the same name are treated as once class. Namespaces are similar in that they are non unique and all class definitions inside a namespace with the same name are treated as one collection. Namespaces and types can also be aliased, which resulted in less elegant rules than other NBL rules, but is still expressible in NBL. Since C# is a more complex language than the Entity language, some constructs from C# could not be modelled in NBL. These are discussed below.

Method overloading with subtyping While method overloading can be expressed using the interactions with the type system, subtyping in method overloading is not handled at all. With subtyping, calling an overloaded method can cause an ambiguity. The type system of the language must then decide which method the call binds to, or to shown an error message in case of a real ambiguity problem. The interactions with the type system in NBL are not expressive enough to allow solving ambiguities. One way to solve this is to allow a user defined function to resolve the ambiguity in a language specific way, but this would break some of the declaratively since the function would be written in a programming language. It could be specified declaratively in NBL, but then NBL is also entering the domain of specifying type systems.
NBL already has several interactions with the type system and it is unclear whether separating these and other typing rules into a separate DSL is a good idea. For example, if we would have a typing language similar to NBL, a typing rule for a method call could look like:

\[
\text{Call}(e, a^*): \\
\text{has type } r \\
\text{where } e \text{ has type } \text{Type}(c) \\
\text{where } a^* \text{ has type } t^* \\
\text{where } m \text{ refers to } \text{Method } m \text{ of type } (t^*, _) \text{ with subtyping in Class } c \\
\text{where } m \text{ has type } (_, r)
\]

This shares a lot of code with the name binding rule for method calls, specifying these kind of 'has type' rules in NBL is probably a good idea. However, specifying a subtype (or supertype) relation is more separated from NBL. Class inheritance is specified in NBL using import statements, but these do not specify a supertype relation and how to resolve ambiguities. A supertype relation could be specified as follows in a typing language:

\[
\text{Class}(\_, c, \text{Base}(s), \_): \\
\text{supertype of } \text{Type}(c) \text{ is } t \\
\text{where } s \text{ refers to } \text{Class } sc \\
\text{where } sc \text{ has type } t
\]

This looks different from the name binding rule for classes; a match on the base class is performed and the type of that class is retrieved. With more configuration about how subtyping works for a language, a generic typing algorithm could resolve subtyping ambiguities.

**Dynamic typing** A variable can be typed as dynamic, at which the compiler defers name binding to the runtime, effectively disabling static name binding for that variable. Static name binding of field access or method calls is not possible on dynamic variables, since the type is dynamic, but it is not possible to specify this in NBL. A possible way to solve this is to allow a reference to bind to any name:

\[
\text{FieldAccess}(e, f): \\
\text{refers to } \text{Field } f \text{ in Class } c \\
\text{where } e \text{ has type } \text{ClassType}(c) \\
\text{otherwise refers to } * \\
\text{where } e \text{ has type } \text{Dynamic}() 
\]

This follows the pattern of the alias rule in the NBL definition for MiniC#. The part that cannot be specified in NBL is \text{refers to } *, which would bind to any field name, disabling name binding for dynamically typed variables.

**This** Inside methods, the \textit{this} keyword refers to the object the method is operating on. For field access and method call purposes, \textit{this} refers to the surrounding class the method is defined in. To solve this, the \textit{in surrounding scope} concept on definitions could be used on references:

\[
\text{This}(): \\
\text{refers to } \text{Class in surrounding scope}
\]

Referring to a class in surrounding scope would try to find the first class in the surrounding scope.
Nested predefined definitions  Predefined references can be defined in NBL by referring to concrete names. However, in C#, the string and int keywords refer to the System.String class and System.Int32 struct respectively. Those references are nested inside the System namespace. Predefined definitions in other languages may be nested multiple times. It is not possible to express this in NBL. Allowing in to be used with concrete names could solve this:

```
StringType () : refers to Class "String" in Namespace "System"
```

2.4.3 C

The C language is a C-like language without macros. The main concepts of this language are structs, functions, typedefs and includes. Two main characteristics from C are that all definitions are scoped on a file-level and definitions must precede references. The syntax and name binding definition can be found in Figure C.1 and Figure C.2 respectively.

Scoping on file level is performed as explained in Section 2.3.1 about unscoped definition sites. The restriction that struct, variable and function definitions must precede references is specified using in subsequent scope. Defining name binding for the C-like language was relatively simple because macros are not considered. Macros increase the complexity because C macros are basically a Turing-complete programming language. This is definitely not something that NBL can or aims to express.

One problem that occurred in C is that typedefs can be cyclic, for example:

```
struct A { int a; }
typedef struct A A1;
typedef A1 A2;
typedef A2 A1;
```

This poses a problem when trying to retrieve the actual type in case of field access:

```
int a(A2* a2) { return a2->a; }
```

The real type that A2 refers to cannot be retrieved, because it is looping. It is the job of the type system to ensure that it does not loop and an error is shown instead. The type system for this language is written in Stratego, it recursively looks up the actual type of typedefs and prevents loops. However, if typing rules are added to NBL (as seen in the subtyping issue), such a recursive lookup could be specified as:

```
TypeDef(r, t) : defines Type t of type Type(s) in subsequent scope
where r refers to Type d
  recurse d refers to Type d’
until d’ refers to Struct s
```

```
Type(t) : refers to Type t
  has type st
where t has type st
```

The type definition will store its actual type by recursively referring to the inner type until a struct is reached. The implementation that performs the recursion should memorize which definitions have already been seen. If a memorized definition is seen again, a loop is encountered and the type of the definition should be unknown. A type reference then simply retrieves the type that is stored on the type definition. An alternative
2. **Declarative Specification of Name Binding and Scope Rules**

 approach is to do the recursion on the type reference instead of the type definition. The type is then only calculated when required, but each reference will recalculate the type.

### 2.4.4 Green-Marl

Green-Marl [18] is a DSL for graph analysis. From analyses specified in Green-Marl, efficient algorithms are generated for specific platforms. The name binding definition can be found in Figure D.1.

Defining name binding for Green-Marl was simple for most constructs, except for the node property construct. Node properties add properties to all graphs or to a certain graph. It can be seen as prototypes in a language as Javascript, but restricted and static (performed during compile time). The problem here again is that a more complex interaction with the type system is required.

### 2.4.5 Mobl

Mobl [17] is a DSL for mobile application development targeting HTML5 and Javascript. The name binding definition can be found in Figure E.1. This name binding definition for Mobl is not complete nor entirely correct. It was made as an experiment for testing if NBL could be applied to an existing DSL that is bigger than Green-Marl, and to find language constructs that cannot be expressed in NBL. Some language constructs could not be expressed in NBL, which are discussed below.

**Syntax verbosity**  
Mobl has a very verbose syntax that is not designed specifically for name binding with NBL. This is very apparent in the NBL definition, there are many duplicate entries for language constructs that have very similar or exactly the same name binding. For example, the rules for `Function`, `ExternalFunction`, etc. all have the same name binding semantics, but separate rules in NBL. This can be partially solved by desugaring the syntax in a simpler one, but that might throw away extra information that for example the compiler uses. If the extra information cannot be removed, a better syntax for NBL would be to put function kinds into a separate subterm:

| Function([], ...) |
| Function([External()], ...) |
| Function([Sync()], ...) |
| Function([External(), Sync()], ...) |

Then the 3 different rules for functions can be combined into one:

```
Function(_, _, m, _, _):  
defines Method m  
scopes Variable
```

This still entails changing the syntax, which would require refactoring of the other editor services and compiler as well. Providing a mechanism for combining pattern matches as follows would alleviate this:

```
Function(_, m, _, _, _), ExternalFunction(_, m, _, _),  
ExternalSyncFunction(_, m, _, _):  
defines Method m  
scopes Variable
```
This way the syntax can be kept the same, but the number of NBL rules is reduced to one as well. Of course this will only work if the same variables are bound to in each of the patterns.

**By name arguments**  When calling a method, arguments can be passed by parameter name instead of in order, like so:

```java
rename(name: "toString", newName: "ToString")
```

The by name arguments should then reference parameters from the calling method. The abstract term of a by name parameter is `NamedArg("name", ...)`, which does not contain a reference to the method. It was not possible to express this NBL, for the same reason that the `this` keyword could not be expressed. The solution is also similar:

```java
NamedArg(a, _):
  refers to Variable a in Method m
where m refers to Method in surrounding scope
```

This time `in surrounding scope` is used as part of a where rule.

### 2.4.6 Compiler Construction lab

In the lab of the 2012-2013 Compiler Construction course given at the Technical University of Delft, NBL was used to specify name binding for MiniJava []. In the previous 2 years, name resolution was implemented in Stratego using a Stratego library called the namespace library. The namespace library contained concepts for storing and retrieving data in different namespaces and scopes, much like the concepts from NBL. However, the concepts had to be programmed (encoded) using Stratego rules. These rules are verbose, causing them to be split over multiple files. With NBL, the concepts are part of the language and can be specified in one file in a much conciser way. NBL also features imports, anonymous scopes and better interaction with the type system, which are not part of the namespace library.

Another problem is that Stratego has a steep learning curve, especially for students that have not been exposed to functional programming yet. Performing name resolution with Stratego and the namespace library requires significant knowledge about Stratego, including traversals. Understanding Stratego is not a required to perform name resolution with NBL. Thus students can begin by implementing name resolution using NBL which introduces them the concept of terms and pattern matching without the other complexity from Stratego. This results in a better learning curve.

Students were able to understand name binding concepts and how to specify them in NBL faster compared to the namespace library. They were thinking *what* name resolution should do instead of thinking about *how* to implement name resolution. Instead of the 3 labs required to implement name binding using the namespace library, only 1 lab is required to specify name binding using NBL. Most of the implementation effort that went into encoding a name resolution algorithm and programming editor services such as reference resolution and code completion is not required any more. These implementations are now generated or derived, as we will show in Chapter 5 and Chapter 4.
Chapter 3

Persistent Infrastructure for Incremental and Parallel Name Resolution

3.1 Introduction

Programmers make a lot of small changes to programs; incrementally changing programs to the desired state. An IDE needs to be responsive to changes from the programmer, source code highlighting and error markers must be updated immediately. This is commonly solved by updating the model of the source code incrementally, only requiring an update to the model that is proportional to the change made. However, creating incremental analyses requires a lot of implementation effort.

Currently, incrementality in Spoofax is ad hoc; each language has a separate implementation of incrementality, or no incrementality at all. For example, big languages that have been in development for years such as WebDSL and Stratego have their own incremental analyses. There is some library/common code for performing incremental analyses, but it is very primitive and requires a lot of programmatic extensions. Analyses are also non-parallel, only one CPU core is exploited because only one analysis is executed at a time. A parallel implementation is important because the trend is that the number of cores in computers will grow faster than the raw computational power per core. Because of the required implementation effort, languages made using Spoofax are rarely made incremental and parallel. In Xtext [9], an incremental compiler can be automatically derived, but only if language concepts directly map to Java concepts [1]. For languages that do not map to java concepts, a compiler is not generated. The compiler is also not parallel.

A typical analysis that is found in every language is name resolution, as described in the previous chapter. Other analyses and transformations usually found in Spoofax languages are constraint checking, visible definitions (code completion) and compilation. These analyses use information from name resolution, to resolve references, check name binding and scoping, etc. In other words, these are name-dependent analyses. Name-dependent analyses can be exploited to automatically make them incremental if the name resolution is incremental.

3. Persistent Infrastructure for Incremental and Parallel Name Resolution

Using NBL, a language-parametric incremental name resolution algorithm can be created that NBL definitions can parameterize, removing the need for ad hoc solutions to the name resolution problem. Name-dependent analyses can also be performed incrementally if such a name resolution exists. This will reduce the implementation effort significantly, resulting in more incremental and parallel language implementations.

Currently, dynamic rewrite rules [5] are used to keep state in Stratego. These are used in current implementations of incremental analysis and name resolution [16]. Dynamic rules are typically scoped; scoped rules can only be used in the scope they are defined in and are removed when the scope is closed. Unscoped dynamic rules [24] on the other hand stay valid after an analysis finishes and can be reused in the next analysis. Unscoped rules are persistent and can be used for storing global state.

But the implementation of dynamic rules in Spoofax does not match the requirements of a generic incremental and parallel name resolution based on NBL definitions. The main drawback is that dynamic rules are not thread safe and cannot be used in parallel efficiently. Using dynamic rules from multiple threads would mean having to lock the dynamic rule store or splitting and merging it. Locking introduces latency overhead, small operations such as adding or retrieving dynamic rules would require locking, resulting in threads waiting for access to a dynamic rule. Splitting and merging introduces computational overhead, the entire dynamic rule store is copied which does not scale well with many dynamic rules. It is hard, if not impossible, to have efficient thread safe dynamic rules because of their expressiveness. A simpler and less expressive system has more options for efficient thread safety.

Dynamic rules do not support retrieving multiple values per key, since dynamic rules internally are hashmaps. Only one value per key can be mapped. This is problematic for non-unique definitions from NBL, such as partial classes. All non-unique definitions have the same key, so adding a partial class would override the existing partial class. This could be solved by having a list of values as value, but that would increase the complexity of managing dynamic rules. The same problem arises with detecting duplicate definitions for unique definitions.

Managing multiple files incrementally is also not an easy task with dynamic rules, as there are conflicting performance concerns. If a file is removed, all data about that file should be removed. To do this efficiently, all data for a file should be contained in one dynamic rule so that the entire collection can be removed in one operation. However, performing a lookup for a certain key should consult all files. If data is contained in multiple dynamic rules for each file, this entails multiple lookups need to be done, making the lookup linear instead of constant. Dynamic rules were not made to efficiently solve these kind of problems.

In any case, a simpler and less expressive infrastructure is required for performing efficient incremental and parallel name resolution based on NBL definitions. Note that this new infrastructure should not be a replacement for dynamic rules in general, that would possibly make it too general and expressive again. It should only be a replacement of dynamic rules for the purpose of incremental and parallel name resolution.

This chapter is outlined as follows. In Section 3.2, the requirements for the new infrastructure are discussed. The design of the infrastructure is described in Section 3.4. Section 3.5 details the object-oriented design, implementation and testing of the infrastructure. The implemented infrastructure is evaluated in Section 3.6. Finally, in
3.2 Requirements

The new infrastructure has a number of functional and non-functional requirements which are detailed in this section.

3.2.1 Functional requirements

The main functional requirement is that the infrastructure contains a symbol table with key-value pairs, similar to how dynamic rules are used. Key-value pairs can be added to the symbol table and retrieved or removed using the key. The symbol table and key-value pairs must be set up in such a way that a generic name resolution algorithm based on the NBL can use it.

Data in the symbol table should be persisted if the IDE is closed so that this information can be reused when the IDE is opened again.

The infrastructure will be used in Spoofax and must be integrated with Stratego so that analyses in Stratego can use the symbol table. A pure Stratego implementation is not feasible because of the limitations of Stratego. This means that the infrastructure must be implemented in Java with an API that exposes the index in both Java and Stratego.

Although the infrastructure is used in Spoofax, it must also be usable outside of Spoofax, for example to do command line compilation. However, the main focus is to create an infrastructure for the IDE.

The required infrastructure is shown in Figure 3.1. The bottom layer is the infrastructure itself. It provides a persistent and thread-safe symbol table that is used by the name resolution algorithm through an API. An API to the infrastructure is also provided in Java so that command line compilation can be performed through this API. The algorithm layer provides a language-parametric name resolution algorithm and an API for incremental and parallel scheduling. The API to the infrastructure is also provided because editor services like code completion require access to the persisted name resolution information. The top layer is a Spoofax language implementation using Stratego, SDF and NBL.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

3.2.2 Non-functional requirements

The functional requirements of the infrastructure are relatively simple. The non-functional requirement of supporting parallel analyses will cause most of the complexity in the design and implementation. To fully utilize the CPU, multiple analyses must be run in parallel when possible. For example, if four new files are added to the project, they can be analyzed in parallel to use all the cores of a quad-core CPU. An intelligent data structure and API is required to efficiently support parallel analyses. Furthermore, because there is concurrent read and write access to shared data, there must be some form of consistency so that analyses using the shared data are also consistent.

Performing whole program analyses does not scale beyond small projects. To scale up, the infrastructure must support incremental analysis.

There will be many small query and addition operations on the symbol table, so these operations must be efficient and fast. Latency should be kept to a minimum so that analyses do not spend most of their time blocking.

Finally, the implementation must be usable in the open source environment of Spoofax, which currently uses the LGPL license. It is problematic to make dependencies on closed source software in such an environment, especially if the software is commercial. If for example a commercial library is used, everyone who wants to build Spoofax needs a license for this commercial library. A dependency to GPL licensed software is also problematic because it would infect the entire codebase with the GPL license. Dependencies on software with LGPL-compatible licenses, such as LGPL itself, BSD/MIT or the Apache license, are possible.

3.2.3 Existing approaches

An existing approach for controlling access to shared data is software transactional memory (STM), first conceived in [31]. In STM, access to shared memory is controlled by transactions, similar to transactions in databases. Modifications in transactions are isolated from other transactions and transactions are atomic. In contrast to STM, transactional memory can also be supported by hardware. In new Intel processors with the Haswell architecture, instructions for hardware transactional memory were added, called restricted transactional memory [19].

A method commonly used to implement STM in an efficient and consistent way is multiversion concurrency control (MVCC), first conceived in [2]. MVCC assigns an incrementing revision number or timestamp to data and multiple versions of data can exist. When inside a transaction, shared data is viewed as a snapshot that is consistent during the transaction, even if other transactions change this data. Reading data does not require any locking or waiting, but there is a space overhead of storing older information. We will use a variant on MVCC for the design and implementation of the infrastructure.

3.3 Stratego Primer

In this and following chapters Stratego code is used to illustrate parts of the implementation and to show examples. Since Stratego is not a very typical programming
language we explain the basics required to understand Stratego programs.

**Stratego** Stratego is a language for defining (program) transformations based on rewriting strategies. Transformations are specified using term rewrite rules. These rules are combined into transformations by means of strategies, which control the application of rules. Transformations operate on terms, which are tree-based representations of data structures. In particular, a transformation operates on the current term. The current term is transformed into the desired state, or a failure occurs. When a failure occurs the current transformation is cancelled and failure is propagated to the calling transformation, similar to exceptions.

Terms can be represented in Stratego. The following term represents an addition of the integer literal 21 with the value of variable $x$:

$\text{Add}(\text{Int}(21), \text{Var}(\text{"x"}))$

Add, Int and Var are named constructors with a fixed number of sub terms. 21 is an integer literal, "x" a string literal. Lists and tuples can also be defined:

$\text{["List", "of", "strings", ("and", "a", "tuple")]}$

Brackets denote a list of arbitrary length while parentheses denote a fixed length tuple, or unnamed constructor.

**Rules** The most commonly used unit of computation in Stratego is the *rule*, not unlike functions in many other programming languages. Rules try match the current term and apply a transformation to transform it to the desired state. For example, the following rule transform additions into multiplications:

$\text{add-to-mul} : \text{Add(left, right)} \rightarrow \text{Mul(left, right)}$

To apply this rule on a term we can either use a strategy that controls the application, or we can use the *application* operator:

$\langle\text{add-to-mul} \rangle \text{ Add(1, 2)} \Rightarrow \text{Mul(1, 2)}$

$\langle\text{add-to-mul} \rangle \text{ 1} \Rightarrow \text{fail}$

Angle brackets around a rule denote the application of it on the term right of it. In the Stratego examples of this thesis, we use $\Rightarrow$ to denote the results of an application, although this is not valid Stratego syntax. We can see that applying *add-to-mul* on an *Add* term produces a *Mul* term, but application on an integer literal fails.

**Strategies** Strategies are the basic unit of computation in Stratego. In fact, rules are actually syntactic sugar for strategies. Whenever a strategy is used, a rule can be used as well. Strategies can be used as parameters to strategies, creating high-order strategies. For example, the *map* high-order strategy takes a strategy and applies it to every element in a list:

$\langle\text{map}\{\text{add-to-mul}\} \rangle \text{ [Add(4, 2), Add(2, 4)]}$

$\Rightarrow$

$\text{[Mul(4, 2), Mul(2, 4)]}$

**Operators** Stratego provides several operators and combinators, we will describe the most important ones.

**build**

The build operation, denoted by $\text{!}$, replaces the current term with the given term.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

| !Add(1, 2) ⇒ Add(1, 2) !1 ⇒ 1 |

**sequential composition**

Strategies can be applied in sequence using the ; operator. If any of the strategies fails, the composition fails as well.

| !Add(1, 2); !1 ⇒ 1 |

**match or bind**

Stratego has one operator for match and bind operations: the ? operator. If it is applied to an unbound variable, such as ?a, it binds the current term to a. If it is applied to a bound variable, or fully bound term, it acts as a match. For example, !1; ?a; !2; ?a fails because the second ?a will try to match the bound value 1 against the current term 2. It is also possible to match and bind at the same time, _ denotes the always unbound variable:

| !Add(21, 21); ?Add(a, _); !a ⇒ 21 |

A commonly used alternative to the ? operator is the := operator which is not applied to the current term, but to a given term.

| a := 1; a := 2 ⇒ fail Add(a, _) := Add(21, 21); !a ⇒ 21 |

Lists have special semantics for match that allows matching the head and rest of a list:

| [head|rest] := [1, 2, 3]; !head ⇒ 1 !rest ⇒ [2, 3] |

**choice**

A choice can be made using the left choice operator <+ . If the left strategy fails, the second strategy is applied.

| !Add(1, 3); ?Mul(3, 7) <+ ?Add(a, _); !a ⇒ 1 |

An alternative is the if then else end notation:

| term := Add(1, 3); if Mul(3, 7) := term then !42 else if Add(a, _) := term then !a end |

**where**

Extra matches and binds can be added to a rule using a where clause. If any of the strategies inside the where clause fails, the entire rule fails.

| add-to-mul: Add(left, right) → Mul(a, b) where a := <add> (left, 2); b := <add> (right, 4) <add-to-mul> Add(1, 3) ⇒ Mul(3, 7) |

Stratego provides an array of built in strategies, we describe the most important ones.
concat, conc
Concatenates a list of lists, or a tuple of lists.

```
<concat> [[1, 2], [3, 4]] ⇒ [1, 2, 3, 4]
<conc> (([1, 2], [3, 4])) ⇒ [1, 2, 3, 4]
```

mapconcat
Applies a strategy on each element in a list and concatenates the results of each application.

```
makelist: num → [num, num]
<mapconcat(makelist)> [[1, 2], [3, 4]] ⇒ [1, 2, 3, 4]
```

filter
Applies a strategy on each element in a list. If application fails, element is not included in the resulting list.

```
<filter (?2)> [1, 2, 2, 3] ⇒ [2, 2]
```

make-set
Takes a list and returns a set without duplicates.

```
<make-set> [1, 2, 2, 3] ⇒ [1, 2, 3]
```

gt
Succeeds if the first element in a tuple is greater than the second one.

```
<gt> (1, 2) ⇒ fail
<gt> (2, 1) ⇒ (2, 1)
```

eq
Succeeds if the both elements in a tuple are equal.

```
<eq> (1, 2) ⇒ fail
<eq> (2, 2) ⇒ (2, 2)
```

getfirst
Gets the first element for which a strategy succeeds.

```
gt2: x → (x, 2)
<getfirst(gt2)> [1, 4, 6] ⇒ 4
```

args
Returns a list of arguments for given constructor.

```
<args> (1, 3) ⇒ [1, 3]
<args> Add(3, 7) ⇒ [3, 7]
```

3.4 Design

3.4.1 Symbol table
The infrastructure contains a global symbol table; the index. Each project has a separate index because by default files in a project cannot access the files from other projects, unless some kind of dependency is specified. Languages are also separated since two languages could use the same identifiers, but are not related to each other. The index contains key-value pairs called entries. Entries are immutable, the key nor the value can be modified to prevent concurrent modification of entries.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

signature constructors
// Generic
_     : String        → PathPart
Anon  : String        → PathPart
_     : List(PathPart) → Path
_     : [Namespace|Path] → URI

// Analysis (name resolution)
Def    : URI            → Key
Ref    : URI            → Key
Type   : URI * UType    → Key

// Language specific
Class  : _              → Namespace
Method : _              → Namespace
Field  : _              → Namespace
CSType : Identifier    → UType

Figure 3.2: Constructors of index keys in Stratego.

Keys A key identifies named and unnamed program elements, based on their namespace, identifier and scope from the NBL definition.

To take identifiers and scope into account, a key contains a path to a program element. The path of a program element is defined by the identifiers of sibling and parent program elements that lexically scope it, merged with its own identifier. For example, a namespace in C# scopes classes and classes scope methods. A path to a method is constructed by taking identifiers of the namespace and class that lexically scope it, and its own identifier. Paths can also contain anonymous scopes for which a unique identifier is generated. Name resolution needs to append and remove sections of the path, so paths should be easy to construct and modify.

Keys also define what kind of data the entry represents. Examples of kinds of data from the NBL are: definitions, references, types and imports.

Thus the key should contain the kind, namespace and a list that describes a path. For example, to store a definition to a class String in namespace System, a key with kind Def, namespace Class and path System, String is needed.

Signatures In Stratego, the signature of a key is defined in Figure 3.2. A signature definition consists of 3 parts; an optional name, its subterms and the type of the signature. For example _ : String → PathPart defines that builtin String, produces a PathPart. It is unnamed so it cannot be constructed in Stratego, but can be used as an injection in place of a PathPart. A named signature such as Def : URI → Key defines a new constructor Def that has an Uri subterm and produces a Key.

Signatures for index keys are composed from three parts; a generic, analysis specific and language specific part. The generic part is always the same and is required to use the index, since it defines how the path and namespace are composed into a URI which is used in the index. We call the composition of a namespace and path a URI since they can be visualised as one: Class://System/String, which is a URI to a Class in path System, String.

The analysis specific part parameterizes a URI with a kind, turning it into a key that can be used in the index. Note that the Def and Ref constructors for name resolution do not define a value, the key itself (or the URI of the key) is the value. The Type
constructor does contain a value; a user defined type. In both cases the key is reused as the value, to reduce the number of constructors that have to be defined.

Finally, the language specific part is provided by each language to parameterize the namespaces and types in that language, which are defined in NBL. An example of a type key according to the signature is:

```
Type([Class(), "String", "System"], CSType("String")
```

Note that the order of identifiers is reversed for easier appending of new identifiers when constructing a path. This is because lists in Stratego are constructed as nested (functional) cons/nil terms. Adding an identifier to the end of the list requires a traversal over the entire list, whereas adding an identifier to the beginning is a constant operation.

Keys in the index are not unique, one key can point to multiple values. Values can be any Stratego term.

**Partitions** The index is divided into partitions which are used to isolate entries. The signature of partitions is defined in Figure 3.3. A partition consists of either the name of a file, a URI or a tuple with the file name and URI. This way a partition can refer to an entire file, to all entries with a certain URI, or to an entry with a certain URI in some file. For example "file.cs" is the partition that contains all entries in file.cs. 

```
[Class(), "C3"]
```

is the partition that contains all entries of each (partial) class named C3. The partition 

```
("file.cs", [Class(), "C3"])
```

contains all entries of (partial) class C3 that are defined in file.cs.

**Operations** To query and modify the index, the operations in Figure 3.4 are defined on the index. Each operation has a name, argument types and a return type, _ denotes the unit type. This is not valid Stratego code, but can be seen as a sort of Stratego
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

A 'header file' for the index. A remove operation is not defined since it is not required and introduces complexity.

To make the operations easier to use in Stratego, we do not require that the index and partition to operate on are passed to each operation. Instead, the `setup` operation is used to set the current index and `set-partition` is used to set the current partition. The current index and partition are implicitly passed to each operation.

We now describe the informal semantics and show examples of each operation.

**setup**
Sets up the current index for given language and project path (as strings). Must be called before performing any other operation.

If called with a language and project path that does not have an index yet, a new index is created and stored in memory. If the index was persisted to the project path before using `persist`, that is loaded from disk into the index in memory. When the index for given language and project path was already set up before, it uses the index in memory instead. This is a fast operation.

```
<setup> ("CSharp", "/path/to/project")
```

**set-partition**
Sets the current partition on which get, add and clear operations operate on.

```
<set-partition> "String.cs"
<set-partition> ("String.cs", [Class(), "String", "System"])
```

**persist**
Persists the current index to disk. Uses the project path set by `setup` to determine where the index is stored.

**get**
Query for entries with a certain key, irrespective of partition. Partitions are not respected, because when querying for the key of `class B`, it is unknown in which partition `B` is. If the entry was not found, the operation fails. When querying for an entry with a value, such as an entry of kind `Type`, the value can be omitted by using an empty tuple.

```
<get> Def([Class(), "String", "System"])) ⇒ [Def([Class(), "String", "System"]))
<get> Def([Class(), "String"])) ⇒ fail
<get> Type([Class(), "String", "System"], ()) ⇒ [Type([Class(), "String", "System"), CSType("String")]
```

**get-children**
A variation of `get`, which instead of querying directly on the given key, queries in all children of that key. If no children are found, an empty list is returned. To retrieve all classes in the `System` namespace, the following strategy is performed:

```
<get-children> Def([Class(), "System"])) ⇒ [Def([Class(), "String", "System"]), Def([Class(), "Array", "System"])]
<get-children> Def([Class(), "String", "System"])) ⇒ []
```
**get-partition-entries**

Given a partition, get a list of all entries in that partition.

```
<get-partition-entries> "String.cs" ⇒
[Def([Class(), "String", "System"],
   Type([Class(), "String", "System"],
   CSType("String")))]
```

**get-key-partitions**

Gets all partitions that contain the given key. If a key is used in more than one partition, all partitions are returned.

```
<get-key-partitions> Def([Class(), "String", "System"]) ⇒
["String.cs"]
```

**add, add-all**

Adds one entry or a list of entries to the index under the current partition.

```
<add> Def([Class(), "String", "System"])
```

**clear**

Removes all entries from the current partition.

### 3.4.2 Transactions

**Semantics** Some of the index operations may only be used under certain conditions. For example, add, clear or query operations may only be executed in either a single threaded environment, or during a transaction in a multi threaded environment. Failing to do this could result in conflicts, race conditions and uncontrolled concurrent modification of the index data structure.

To support safe concurrency, transactions are used as concurrency control. A transaction contains a log of modifications that are to be applied to the index. Since entries are immutable and removal of individual entries is not defined, only `add`, `add-all` and `clear` are valid modifications.

Transactions are also used as a view over the index and the modifications in the transaction. All `get` queries under the view take into account the modifications in the transaction. For example, if the current partition is cleared, entries from that partition become invisible in the transaction. Entries that are added become immediately visible inside the transaction, but stay invisible to other transactions.

A transaction is closed by committing it. When a transaction is committed, the index is write-locked and modifications in the transaction are applied to the index. All modifications become visible for other transactions. This is the only place where a write-lock on the index is necessary. Applying modifications to a transaction however does not block other transactions, since modifications are stored in the transaction, not immediately applied to the index. Since committing a transaction will happen a lot less often than performing modifications, the number of locks are relatively low.

There are restrictions on transactions to make sure that multiple transactions can run in parallel on different threads. A transaction may only apply modifications to one partition, the current partition at the time of starting the transaction. Furthermore, only one transaction per partition may exist to guarantee that modifications to a partition are isolated by one transaction thread. In summary, one thread executes one transaction...
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

**signature constructors**

\[
_ : \text{Long} \rightarrow \text{Revision}
\]

**operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start-transaction</td>
<td>_: _ \rightarrow \text{Revision}</td>
<td>Starts a new transaction for the current partition (set with set-partition). Returns a unique (incrementing) revision number that is assigned to the current partition. This number is generated so that it is higher than all previous revision numbers.</td>
</tr>
<tr>
<td>commit-transaction</td>
<td>_: _ \rightarrow _</td>
<td>Ends the current transaction. Commits the modifications to the index. Also assigns a new (incremented) revision to the current partition, but is not returned. This revision number is generated in the same way as start-transaction.</td>
</tr>
<tr>
<td>get-partition-revision</td>
<td>_: \text{Partition} \rightarrow \text{Revision}</td>
<td>Retrieves the revision assigned to given partition. If no revision has been assigned to the partition, -1 is returned.</td>
</tr>
</tbody>
</table>

Figure 3.5: Transaction operations on the index.

that updates one partition. This allows multiple transactions to run in parallel without waiting or locking for queries or modifications.

Transactions never fail because modification conflicts on the index do not occur. Transactions can only change their own partition and there can only be one transaction per partition. So there cannot be two (or more) transactions that modify the same entry in the index, hence no modification conflicts occur. This also means that race conditions cannot occur, since two threads cannot edit the same data at the same time. Consequently, no undo log is required, reducing the complexity and overhead.

The signatures and operations for transactions are defined in Figure 3.5. The informal semantics of the operations are as follows:

**start-transaction**

Starts a new transaction for the current partition (set with set-partition). Returns a unique (incrementing) revision number that is assigned to the current partition. This number is generated so that it is higher than all previous revision numbers.

\[
\text{<set-partition >} \ "\text{String.cs}"; \\
\text{start-transaction} \Rightarrow 2
\]

**commit-transaction**

Ends the current transaction. Commits the modifications to the index. Also assigns a new (incremented) revision to the current partition, but is not returned. This revision number is generated in the same way as start-transaction.

**get-partition-revision**

Retrieves the revision assigned to given partition. If no revision has been assigned to the partition, -1 is returned.

\[
\text{<get-partition-revision>} \ "\text{String.cs}" \Rightarrow 2 \\
\text{<get-partition-revision>} \ "\text{Unknown.cs}" \Rightarrow -1
\]

**Protocol** Improper use of the index and its operations could still result in uncontrolled concurrency which leads to the problems described earlier. To explain the correct usage of transactions and which operations are allowed, we define a protocol. See Figure 3.6 for the protocol that, if followed, guarantees that no conflicts, race conditions and uncontrolled concurrent modifications occur inside the index. The protocol is a mix of Stratego and regular expression-like syntax. Parenthesis open and close groups, they either contain new groups or strategy applications that must be called at that moment. A star denotes that the element before it can be quantified zero or more times. Vertical bars denotes choice between elements in a group. The dot denotes valid characters for an identifier in Stratego.
Design

```plaintext
<setup> (language, project-path);
(get.★;|add.★;|clear;|persist;★)

{ fork-thread;
  {<set-partition> partition;
   start-transaction;
   (get.★;|add.★;|clear;★
    commit-transaction;
  )★
  }
  join-thread
}

Figure 3.6: Protocol for using the index in a concurrent way using transactions.
```

```plaintext
<setup> ("CSharp", "path/to/project")

fork-thread;
<set-partition> "String.cs";
start-transaction;
clear;
<add> Def([Class(),
  "String", "System"])
commit-transaction;
join-thread

fork-thread;
<set-partition> "Array.cs"
start-transaction;
clear;
<add> Def([Class(),
  "Array", "System"]);
<get> Def([Class(),
  "String", "System"])
<+
<add> Def([Method(),
  "ToString", "Array",
  "System"])
commit-transaction;
join-thread
```

Figure 3.7: Example of concurrent operations on the index with transactions in 2 different threads.

First, the index is initialized by calling setup once. From that point, all index operations may be safely used if there are no other threads active at that time. However, once a thread is started, index operations may not be used until a transaction is opened. We use fork-thread and join-thread to denote creating a thread and joining it with the main thread respectively. Inside the thread set-partition is called and may only be called on partitions that are not being modified in a transaction. A transaction is then started using start-transaction. In the transaction, only query and modification operations are allowed. Calling setup, set-partition or start-transaction is not allowed. Finally, the transaction must be committed. In the case of a thread pool, after committing the previous transaction, the partition can be changed and a new transaction can be started in the same thread.

As an example to what is allowed, see the Stratego code in Figure 3.7. The two columns in the table represent two threads executing in parallel. This code follows the protocol so we can guarantee that no conflicts or uncontrolled concurrent modifications occur. However, the outcome of the right transaction depends on data from the left one. This is actually an inconsistency. If the left transaction commits before the get query is performed, the ToString method is added to the index. If it commits after the get query is performed, it is not added. The protocol we described does not guarantee that this is handled correctly. In the following 2 sections we describe how to handle
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

signature constructors
Read : URI → Key
ReadAll : URI → Key

strategies
index-uri :
key → uri
where
[uri|_] := <args> key

index-get :
key → <get> key
where
<add> Read(<index-uri> key);

index-get-children :
key → <get-children> key
where
<add> ReadAll(<index-uri> key);

index-dependencies :
entries → deps
where
reads := <filter(?Read(_))> entries;
readAlls := <filter(?ReadAll(_))> entries;
defs := <filter(index-uri; \x → Def(x))> reads;
defsAlls := <filter(index-uri; \x → Def(x))> readAlls;
children := <mapconcat(get-children)> defsAlls;
deps := <make-set> <mapconcat(index-key-partitions)>
<concat> {defs, children}

Figure 3.8: Stratego library for tracking dependencies in the index.

inconsistencies and dependencies.

3.4.3 Inconsistencies

If the protocol from the previous section is followed, conflicts do not occur. However, in Figure 3.7 we saw that entries that are queried from the index could be inconsistent. Fortunately, inconsistencies can be detected by tracking dependencies and comparing revision numbers.

We define a small Stratego library for tracking dependencies and a protocol for detecting inconsistencies based on this library. The library is defined in Figure 3.8.

The informal semantics of the strategies in the library are as follows:

index-uri
Retrieves the URI from a given key.

[index-uri] Def([Class(), "String", "System"])) ⇒
[Class(), "String", "System"]

index-get
Gets entries from the index using get and stores a dependency (as a Read) to the URI of the requested key.

index-get-children
Gets children entries from the index using get-children and stores a dependency (as a ReadAll) to the URI of the requested key.
<setup> (language, project-path);
(index-get.*;|get-partition.*;|get-key.*;|add.*;|persist;)*
{
  fork-thread;

  (set-partition) partition;
  start-transaction ⇒ analysisRevision;
  (index-get.*;|get-partition.*;|get-key.*;|add.*;|clear;)*

  // Find inconsistencies
  entries := <get-partition-entries> partition;
  partitions := <index-dependencies> entries;
  revs := <map( get-partition-revision )> reads;
  if <getfirst (r ⇒ (r, analysisRevision)), gt> revs then
    // Inconsistency detected, e.g. reschedule analysis.
    end;
  commit-transaction;
  }

  join-thread
}

Figure 3.9: Protocol for performing analyses with inconsistency detection.

index-dependencies
Given a list of entries, retrieves a set of all partitions that the list of entries depends on.

string := <get-partition-entries> "String.cs"
@index-dependencies> string ⇒ [
"String.cs"
]

array := <get-partition-entries> "Array.cs"
@index-dependencies> string ⇒ [
"Array.cs", "String.cs"
]

The protocol is defined in Figure 3.9. It follows the previous protocol, so it is guaranteed to be free of conflicts, race conditions and uncontrolled concurrent modifications. In addition, following this protocol guarantees that inconsistencies are detected and can be resolved.

The analysis should use the index-get strategies from the library so that dependencies can be tracked. Immediately after the analysis a search for inconsistencies is performed. To find inconsistencies, all partition the analysis depends on are retrieved. The revisions for each partition are retrieved and compared to the revision number of the analysis. If any revision number is higher than analysisRevision it means a partition the analysis depends on was updated during the analysis. This indicates that the analysis is possibly inconsistent because the data for a partition changed during analysis. The inconsistency can then be resolved by, for example, rescheduling analysis for the current partition at a later time.

See Figure 3.10 which illustrates the previous example from Figure 3.7 with a revision diagram. In this diagram, L represents the left thread (String.cs) and R represents the right thread (Array.cs). Both transactions are started concurrently, but the transaction of L commits during the transaction of R, resulting in an inconsistency. However, the protocol is used to detect this inconsistency and after R commits a new transaction is started in which no inconsistencies occur.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

Figure 3.10: Revision diagram where 2 transactions $L^4$ and $R^5$ are started in parallel. Commit of $L^4$ causes an inconsistency during $R^5$, resulting in rescheduling of $R$. Numbers denote revision numbers

Inconsistency Reschedule

4 5 6

Figure 3.11: Stratego library for incremental analysis in the index.

3.4.4 Dependencies

In the previous subsection we explained how inconsistencies can be detected and resolved by rescheduling analysis at a later time. However, there is another (simpler) case where analyses have to be rescheduled.

Consider the example from Figure 3.7 again. In our IDE we have both String.cs and Array.cs open. If we change a name in String.cs that Array.cs refers to, for example String is renamed to CharArray, we would expect the IDE to display an error in Array.cs at the position of that name. In other words, if partition $A$ changes, all partitions that depend on entries from $A$ have to be reanalyzed. This is required for performing incremental analysis. To support this, something similar to detecting inconsistencies is performed.

We again define a Stratego library and protocol for incremental analysis. The library is defined in Figure 3.11. The informal semantics of the strategies in the library are as follows:

**index-get-reads**

Given a URI, returns all reads (Read and ReadAll entries) for that URI.

|$\langle \text{index-get-reads} \rangle \langle \text{Class()}, \text{"String"}, \text{"System"} \rangle \Rightarrow$
index-dependent

Given a list of entries, return a set of all partitions that depend on any of the entries in the list. Not to be confused with index-dependencies, which returns outgoing dependencies. This strategy returns incoming dependencies.

string := <get-partition-entries> "String.cs"
@index-dependent> string ⇒
["Array.cs", "String.cs"]

array := <get-partition-entries> "Array.cs"
@index-dependent> string ⇒
["String.cs"]

The protocol is defined in Figure 3.12. Entries in the index before and after the analysis are stored for comparison later. After the inconsistencies check, a diff is performed between the entries before and after analysis, resulting in all added and removed entries. All dependent partition of changed entries are taken using the index-dependent library strategy. These partitions, minus the current partition, must be rescheduled for analysis.

If in Figure 3.7 the right thread finishes entirely before the left thread, no inconsistency would be detected. This situation is wrong though, because the second add operation will not happen. These situations are however correctly handled if dependencies of changed elements are rescheduled for analysis. See Figure 3.13 for a revision diagram illustrating this situation. The commit of L causes a change (like String being renamed to CharArray), this in turn causes Array.cs to be reanalyzed.

Figure 3.12: Protocol for performing incremental analysis.
3. **Persistent Infrastructure for Incremental and Parallel Name Resolution**

Figure 3.13: Revision diagram with where 2 transactions $X^1$ and $Y^2$ are started sequentially. Commit of $Y^2$ causes a change in $X$, resulting in a re-analysis.

3.4.5 **Persistence**

The entire index for a project and language is cached to disk on command, for example after saving a file in the IDE. All transactions are processed and the index is read locked before persisting to disk. If the index for a project and language has not been loaded yet, it is loaded from disk into memory if it was persisted before. Otherwise, an empty index is constructed and all files in the project are queued for analysis.

Since the data in the index are mostly Stratego terms, the Stratego term serialization classes can be used for serializing and deserializing the data structure. Data that are not Stratego terms are simply converted to and from terms.

3.5 **Implementation**

3.5.1 **Object-oriented design**

A class design of the infrastructure is shown in Figure 3.14. For brevity the inherited operations are omitted in the class diagram.

The **Key**, **Entry**, **Partition**, **PartitionDescriptor**, **ISymbolTable** and **Index** are classes that come forth from the design in Section 3.4.1. A **Key** contains a kind, namespace and names in Stratego term form for easy integration with Stratego. Entries contain a value as a Stratego term and their corresponding **Key**, but also a reference to a **PartitionDescriptor** so it is possible to find the partition corresponding to the entry.

A **PartitionDescriptor** describes how a partition can be found using a **Key** and URI to a file, similar to how a key describes how to find an entry. Partitions contain a revision number and their corresponding **PartitionDescriptor**. For all of these classes there are methods for converting to Stratego terms. These are required for interfacing with Stratego using primitives, since data in Stratego are always terms.

**ISymbolTable** is the interface for symbol tables which has the described operations. A concrete implementation of the symbol table is **Index** which implements it using several hash(multi)maps for fast lookup in queries. Another implementation is **Transaction**. It implements the interface so that a transaction can be used as a view over the global index and the changes inside the transaction. When calling a query method such as **get**, it returns entries from its global index plus entries from its trans-
The `IndexManager` class manages indexes and transactions for multiple projects and languages in Spoofox. To prevent the user from having to pass the project, language and partition for each operation on the index, these values only have to be set once on the index manager. Each operation is then executed on the correct index and partition automatically. To make this work even in a threaded environment, all the current values are thread local.

Starting a transaction is simply done by replacing the current symbol table with a new `Transaction` object and returning a new revision number. Committing a transaction reverses this and applies the modifications in the transaction index.

Figure 3.14: UML class diagram of infrastructure.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

3.5.2 Data structure

The Index class contains the main implementation of the data structure. It needs to support efficient retrieval, addition and removal operations, but does not need to be thread safe.

A hash map is required to efficiently look up keys to entries. One of the requirements is that a key can point to multiple values, so instead of a map, a multimap is required. Java does not provide a multimap by default. It can be emulated by using `Map<Key, Collection<Value>>`, but that requires manual bookkeeping and extra implementation effort.

The Guava library, formerly the Google-collections library, provides a multimap interface and several implementations of a multimap. It is licensed under the Apache 2.0 license, which is compatible with the LGPL license, so it is usable within Spoofax.

One of the possible multimap implementations is `ArrayListMultimap`, which implements a multimap roughly as `HashMap<Key, ArrayList<Value>>`. An alternative is the `LinkedHashMultimap` which could have better performance characteristics, because adding an entry is constant in a linked list, but linear in an array. However, `LinkedHashMultimap` does not handle storing duplicate key-value pairs, which should be allowed in the index. Consequently, the best multimap implementation for the index is `ArrayListMultimap`.

The initial implementation of Index has three data structures with the following types. To look up keys and parent keys to entries, two `ArrayListMultimap<Key, Entry>` are used. A `ArrayListMultimap<Partition, Entry>` data structure is used to look up partitions to their entries. Using these hashed multimaps backed by array lists, get and add operations are (near) constant in run-time and multiple entries per key are supported.

However, the results of evaluating (in Section 3.6 below) the implementation hinted at a performance problem in removal operations. The problem with the data structure is that two nested loops are required to remove all entries from the three different multimaps. Also, some entries are removed using an equality lookup instead of a hash lookup, which requires the entire collection to be compared against the entry.

To improve performance a different data structure for retrieving entries is used: `ConcurrentHashMap<Key, ArrayListMultimap<PartitionDescriptor, Entry>>`. Using this data structure, get operations will stay almost identical by just taking the value collection of the retrieved key. The add operation requires some extra bookkeeping to construct multimaps when they do not exist yet. On the other hand, clear operations will be much more efficient since they only require one loop over the value set of the hash map.

3.5.3 Testing

A set of unit tests is created to test both the functionality and performance of the index. These units tests are very useful during the development of the index, ensuring that the index behaves in the same way after every change. Usually, unit tests only cover the functional part of a system. However, the non-functional requirements are also very important in this system because it must be able to scale to real-world projects.

Performance unit tests are used to measure the impact on performance of changes made to the system. This helps with improving performance and making sure that changes do not negatively impact the performance.

For both functional and non-functional unit tests, JUnit is used, a commonly used testing framework in Java. However, JUnit does not properly support performance benchmarking of a system. It only measures the ‘wall clock’ time taken of a test, and only runs the test once. This introduces errors in the benchmark; the JVM might still be optimizing the code, a garbage collection could occur or another process on the system might use some CPU time.

Fortunately, JUnit can be extended with the JUnitBenchmarks extension to add proper benchmarking support. Instead of measuring ‘wall clock’ time, it measures ‘CPU time’, which is the actual time taken of the benchmark (and the JVM itself) without influence of other processes on the system. Benchmarks can also be configured to run multiple times, the average and standard deviation are calculated from the results of multiple runs. The first few runs of a benchmark are discarded because the JVM optimizes code as it runs it. All this improves the precision of the benchmark results. Incorrect results from benchmarks can be detected by looking at the standard deviation for each result. The results of the benchmarks are discussed in the next section.

3.6 Evaluation

To evaluate the new infrastructure, benchmarks are used to test how far it scales with relation to the number of entries and partitions in the index. For each of the performance sensitive operations on the index a performance benchmark is conducted. Both data structures are benchmarked. The initial implementation of the data structure is referred to as implementation 1. The revised data structure implementation is referred to as implementation 2.

To measure scaling, the number of entries and partitions is varied. In benchmarks the number of entries used is: 500, 5000, 50000, 100000, 250000, 500000, 750000, 1000000. There are 5 different kind of entries which are uniformly spread. The number of partitions used is: 1, 10, 100, 500, 1000. This gives us a maximum of 1000 entries for each of the 1000 files, which seems like a reasonable number for a very big project. All benchmarks are run with 5 warmup rounds (for JVM optimization), 15 benchmark rounds and force garbage collection before each benchmark, unless otherwise stated. Benchmarks are run on an Intel Core i7-2600K processor that runs at 3.8GHz. The JVM has 2GB of heap space available and allocated.

The set-up and results of each benchmark are described in the following subsections.

3.6.1 clear

Benchmarking the clear operations is conducted by adding entries spread over partitions and then measuring the time it takes to clear one partition. Due to an error in the benchmarking suite, the time it takes to add entries to the index is also measured, re-
resulting in a higher time to clear a partition. The results are plotted in Figure 3.15. Note that this benchmark only goes up to 100000 entries because of performance problems.

Surprisingly, clearing a partition from an index with 10 partitions takes the most time. In fact, it takes almost 45 seconds to clear a partition with 100000 entries in the index. This could be contributed to a partition containing more entries when there are only 10 partitions in contrast to 100. However, if this were the case then we would expect the benchmark with 1 partition to take the most time. In any case, the implementation 1 does not scale.

Figure 3.15: Benchmark results for the clear operation.

The results of implementation 2 are plotted in Figure 3.16. The second implementa-

Figure 3.16: Average benchmark results for the clear operation.
Evaluation

tation handles clearing partitions a lot more efficiently. When subtracting the time it
takes to add entries (caused by the benchmarking error), the time taken by the clear
operation even seems negligible.

### 3.6.2 add

Benchmarking the add operation is straightforward, we measure the time it takes to add
entries spread over a number of partitions. The results for the initial implementation
are plotted in Figure [3.17](3.17). The low entries data points should be ignored, as they did
not even measure in the benchmark.

As expected, the lines are mostly constant. Adding more entries does not increase
the time to add one more entry, with the exception of the 500 partitions benchmark.
Yet, benchmarks for different partition counts show that the time taken is more or less
constant, so these are most likely measurement errors.

The number of partitions does influence the time it takes to add one entry. This
is most likely caused by the fact that more new partitions are added to the index.
When adding an entry with a partition that has not been added before, extra operations
are executed to add that partition. It is interesting however that there is almost no
difference in time taken for 500 and 1000 partitions in contrast to the larger difference
between 1 and 100 partitions. We were unable to find the cause of this difference.

![Figure 3.17: Implementation 1 benchmark results for the add operation.](3.17)

A comparison between both implementations is plotted in Figure [3.18](3.18). Unfortunately,
implementation 2 requires more bookkeeping when adding new entries, so it
performs worse. However, the performance is still constant and fast enough.
Figure 3.18: Average benchmark results for the add operation.
3.6.3 get, get-children

To benchmark the get and get-children operations, entries are added to the index using the same entry and partition variations from the add benchmark. Then 1000000 get and get-children operations (spread over the 5 different entries) are executed. This benchmark does not suffer from the error of the clear benchmark, purely the time taken to get an entry is measured. The results are plotted in Figure 3.19 and Figure 3.20.

Both implementations have a constant run-time for the operations since they are just simple hash lookups. The get-children operation is slightly slower because the parent of given key must be calculated before performing the lookup. Implementation 2 is slightly faster because it uses a standard java map instead of a Guava multimap.

Figure 3.19: Average benchmark results for the get operation.
Figure 3.20: Average benchmark results for the get-children operation.
3.6.4 get-key-partitions

Benchmarking get-key-partitions is conducted by adding entries beforehand and then executing get-key-partitions on the 5 different keys. The results are plotted in Figure 3.21. Note that the run-time is in milliseconds, not nanoseconds.

For implementation 1 a loop over all entries with the key was required to extract all partitions. In the second implementation this loop is not necessary and is replaced by a hash lookup to get the multimap followed by retrieving the key set (partitions) of the multimap. These operations are very fast, so the run-time of the second implementation could not be measured in the benchmark.

Figure 3.21: Average benchmark results for the get-key-partitions operation.
3. Persistent Infrastructure for Incremental and Parallel Name Resolution

3.6.5 persist

The persist benchmark is conducted by adding entries to the index and then simply persisting the index to disk. The results are plotted in Figure 3.22.

As expected, persisting more entries to disk takes linearly more time because it is basically a loop over all entries. The difference between both implementations is very small because the implementation of persist is unchanged. Implementation 2 is slightly slower because looping over all entries requires an extra indirection for the nested multimaps.

The performance is not very good; at 1000000 entries it takes over 4 seconds to persist. One of the reasons is that Stratego terms are serialized. Some of these terms are lists, which are implemented as nested (functional) cons/nil terms. This causes many recursive serializations which is not very efficient. Another reason is that the entire data structure is serialized, not just the parts that changed. However, persisting does not happen often and can be executed in a background thread (although this blocks committing transactions, but does not block the user interface). This performance problem is not as serious as the problems with the clear operation, but should be investigated in the future.

![Figure 3.22: Average benchmark results for the persist operation.](image)
3.7 Design Considerations

In this section, design considerations and alternatives to design decisions are discussed.

3.7.1 Inconsistencies

In MVCC, inconsistencies are typically resolved by keeping older data available so that transactions always read the old, but consistent data. This approach has the advantage that transactions are always consistent, but could be outdated. The disadvantage is a space overhead for keeping this redundant data, and the complexity of managing this.

In many cases, it is not interesting to show outdated (consistent) information. For example when performing a refactoring, many partitions will change. Old information is may be wrong in that case. Analysis will be performed on a refactored partition, but information about other partitions is outdated, from before the refactoring.

We chose not to keep data consistent, but instead to provide a protocol for detecting and managing inconsistencies, so that the user of the infrastructure can resolve them. This simplifies the infrastructure immensely, storing old data and managing which data to make visible would make the design a lot more complex. It would also incur an extra overhead on space, and a slight computation overhead.

3.7.2 Persistence

An alternative to serializing the entire index is only serializing partitions when they change. This would reduce the time spent writing to disk, and unused partitions can be unloaded temporarily to free up some memory. However, serializing and keeping the entire index in memory has not caused problems so far. If large amounts of data are stored in the index it will become necessary to reduce serialization time, as pointed out by the benchmarks.

3.7.3 Data structure

The simplest alternative is a standard concurrent collection provided by the Java API, such as ConcurrentHashMap or CopyOnWriteArrayList. The problem with these kinds of collections is that locking is required for write operations, reading operations generally do not block. Especially in the situation where there are multiple writers, many writers will be blocked, since only one write may be executed at a time. In our solution, multiple writers do not block each other, only a commit does. Also, Java does not provide a multimap in their API, which is required to have one key point to multiple values.

Another alternative is using a client-server database, such as MySQL. With client-server databases, objects are marshalled into byte streams by the client, transferred to the server and marshalled into the internal database format and vice versa. Analyses do a lot of small read and write operations which would then cause a lot of computational overhead and latency. Operations on the index are mostly simple hash lookups and collection mutations. SQL is not very suited for these kind of small operations, it is suited for performing complex queries that affect multiple entries in

[^4]: http://www.mysql.com
bigger data sets. Also, the database will have to run in a separate process that has to be managed.

A better kind of database for the index is an embedded database such as HyperSQL 5 or H2 6. Instead of a client-server networking model, these databases can be embedded directly into an application and tables can be stored in memory. However, these databases still use internal representations and SQL as query language. The advantage is that these databases have support for persistence and consistency. They also allow a traditional client-server deployment that can be useful for running the analysis on a server and communicating with a client, which is a possibility when running IDE’s like Spoofox on the web [26]. Nonetheless, we chose not to use these databases because of the marshalling overhead and use of SQL for querying and mutation.

5 http://hsqldb.org/
6 http://www.h2database.com/
Chapter 4

Language-Parametric Name Resolution

4.1 Introduction

Name resolution algorithms are commonly written by hand for each language. Writing a name resolution algorithm that is incremental and works in parallel requires a large implementation effort. The complexity and implementation effort of the name resolution algorithm also scales with the complexity of the language. These algorithms can get so complex and filled with small language specific details that they are only understood by the writer of the algorithm. Because they are language specific, they also cannot be reused between languages, causing reimplementation for each language. All this is caused by a lack of abstraction over the name binding domain.

In Chapter 2 we presented NBL, a declarative DSL in the name binding domain. NBL provides the previously lacking abstractions over the name binding domain. Because of these abstractions, a language-parametric (generic) name resolution algorithm can be constructed. Such an algorithm operates on abstractions from NBL, instead of directly on name binding concepts from a language. A language-parametric algorithm based on abstractions from NBL can be re-used between any language that specifies name binding using NBL. Any benefits to the generic algorithm is a benefit to all languages using it. In Chapter 3 we presented an infrastructure for incremental and parallel name resolution. The language-parametric name resolution algorithm can be incrementalized and executed in parallel by using the index, operations and protocols from the infrastructure. Any language with a NBL specification will then get an incremental and parallel name resolution, without the large implementation effort.

This chapter is outlined as follows. First, Section 4.2 briefly introduces the requirements of the algorithm. In Section 4.2 the NBL code generation is described, which is required to understand the algorithm. The language-parametric name resolution algorithm is explained in Section 4.4. Parallel execution of the algorithm is described in Section 4.5. The chapter concludes with an evaluation of the algorithm in Section 4.6.
4.2 Requirements

The functional and non-functional requirements of the name resolution algorithm are described in this section.

The main functional requirement is that the algorithm operates on the abstractions from NBL: namespaces, definitions, references, scopes and imports. In fact, the algorithm will be the first implementation of NBL. It should follow the semantics from the examples and patterns described in Chapter

The algorithm must be reusable between languages, no language-specific elements may be encoded in the algorithm. Since it operates from abstractions in NBL, this should follow naturally. To make the algorithm reusable between platforms, it should be written in (or compiles/transforms to) Stratego code. It should not depend on Eclipse specific functionality.

The algorithm should support semantic editor services in Spoofax; reference resolution, code completion and error checking. Other name-dependent analyses such as compilation must also be supported.

As non-functional requirements, the algorithm should perform incremental name resolution and work in a parallel environment.

4.3 NBL Code Generation

A language-parametric name resolution algorithm is parameterized by the NBL definition for a language. However, to use NBL definitions with the generic name resolution algorithm, they have to be transformed into a format that the name resolution can use. Since the algorithm is written in Aster and Stratego, NBL definitions need to be converted to Aster and Stratego definitions.

Aster is a language that combines strategic programming from Stratego with the declarative attributes and evaluation of reference attribute grammars (RAGs). In Aster, attributes can be defined in several ways. In the pattern major specification form a term is pattern matched and several attributes can be defined on matching terms. This form is very similar to how binding rules are specified in NBL. In NBL, a term is pattern matched and several properties about the matched term are specified. Therefore, it is straightforward to convert binding rules to pattern major attribute specifications in Aster.

Figure 4.1 contains a binding rule for a class definition in C#, its Aster representation is shown in Figure 4.2. The properties of the binding rule are lined up so that their corresponding Aster representation is on the same line. This shows that transforming a binding rule to Aster is almost a one-to-one mapping. The Aster representation is more verbose, but this is not a problem since it is generated code that is not read or modified by users. We now give an overview of all generated attributes.

nbl-def-site

The term where the definition resides, usually set to the current term. In the case of a definition in subsequent scope, this is set to the next sibling. Also used to denote that this is actually a definition.
NBL Code Generation

Class(x, y, _) :
defines
unique
Class x
of type Type(x)
scope for Field, Function
refers to Class y
imports Field from Class y
{transitive}
imports Function from Class y
{transitive}

Figure 4.1: Binding rule for C# class.

eq Class(x, y, _) :
x.nbl-def-site := id
x.nbl-is-unique := id
x.nbl-def-ns := Class()
id.nbl-scope-name := x
x.nbl-type := Type(x)
id.nbl-scoped-ns :=
{Field(), Function()}
y.nbl-use-site := id
y.nbl-use-ns := Class()
y.nbl-imports :=
{Import(Field(),
Trans()),
Import(Function(),
Trans())}

Figure 4.2: Aster representation of binding rule.

nbl-def-ns
The namespace of the definition.

nbl-is-unique
Whether the definition is unique. Only set to id if the definition is a unique definition.

nbl-scope-name
If the current term is a definition but also scopes something, this is set to the name of the definition. Used to point to the name of the current term.

nbl-scoped-ns
The namespaces that this term scopes.

nbl-use-site
The use site term. Used to denote that this term is a reference to a definition.

nbl-uses-ns
If this is a definition that also defines a type, set to the type it defines.

nbl-type
If this is a definition that also defines a type, set to the type it defines.

nbl-imports
A list of terms that describe the imports.

Normal imports are listed as: Import(namespace, {}).
Transitive imports: Import(namespace, Trans()).
Current file imports: Import(namespace, File())

In addition to generating Aster code, some Stratego code is generated as well. The Stratego code generation is shown in Section 4.4.3 in the ‘Adjust Rules’ paragraph.
4. Language-Parametric Name Resolution

4.4 Name Resolution Algorithm

In this section the semantics and implementation of the name resolution algorithm is described.

The name resolution algorithm takes as input an NBL definition and an AST. As output it produces an annotated AST, persisted information in the symbol table and dependency information. The annotated AST is the original AST where all references have been resolved to their definitions according to the rules in the NBL definition. Persisted information in the symbol table contains all definitions, resolved references and other data such as type information. It is used for calculating dependencies and for use in other analyses such as constraint checking and compilation. Dependency information is used to determine which partitions are invalidated by changes, which are scheduled for re-analysis.

The algorithm is divided into four stages: the definition, data, resolve and dependency stages. In the definition stage, each definition is assigned a key. The data stage assigns additional data to definitions, such as type information. The resolve stage is concerned with resolving references to their definitions by performing lookups on the symbol table. Finally, the dependency stage finds changes in data and which partitions depend on this data, making the name resolution incremental. In Figure 4.3, an overview of the stages, their inputs and outputs and the flow of data can be seen.

Each stage requires the previous stage to be completed. For example the reference stage requires that all definitions are known because it needs to resolve references to definitions. It also requires that all data is known since resolving some references for example require data about types or imports. Not performing this kind of staging and only performing one traversal leads to problems. For example, if a reference refers to a definition that is defined later, it would always fail to resolve. This is how the C and C++ compiler works, which requires that all definitions are defined before references. However, many languages allow forward references. Multiple traversals are required to support forward references.

As implementation platforms, Aster and Stratego are used. Because calculations are specified declaratively using attributes in Aster, the resulting algorithm is easier to understand and reason about. However, not all calculations can be expressed in Aster. Aster evaluates attributes defined on terms in an AST, but data from the index is not an AST. Since the resolve and dependency stages require queries on the index, they cannot be expressed in Aster and have to be expressed in Stratego.

The algorithm presented in this thesis has a limitation on the otherwise keyword from NBL. Using otherwise in conjunction with where clauses and import rules is not

Figure 4.3: Overview of the stages of the name resolution algorithm
supported.

In the following subsections, the semantics and implementation of the definition (Section 4.4.1), data (Section 4.4.2), resolve (Section 4.4.3) and dependency (Section 4.4.4) stages are described in detail.

### 4.4.1 Definition Stage

The definition stage is concerned with constructing a key for each program element that is marked as a definition in NBL. Definitions are assigned a definitive key that describe their position (scoping) and identifier in the program.

Keys must be built in such a way that they conform to the NBL definition of the language, but also needs to take into account the lookup algorithm from the resolve stage. The key of a definition consists of the namespace, name and path of the definition. Filling in the namespace and name of a definition is simple. If we look at an NBL rule for classes: `Class(x, _): defines Class x`, the namespace is `Class` and the name is `x`. The path of a class term depends on the scoping rules of the sibling and parent terms. Constructing a path such that references can be properly resolved to the definition corresponding to that path is the main challenge.

#### Scopes and Definitions

To construct a definition key that can be resolved and conforms to the NBL definition, scope and definition rules need to be taken into account. Scopes and definition rules in NBL define how the path, name and namespace of a key is constructed. From now on we assume that the name of a definition is part of the path for simplicity. Two types of scopes can be distinguished: *vertical* scopes and *horizontal* scopes.

Vertical scopes are named or unnamed scopes that scope some or all subterms, for example namespaces, classes or statement blocks. They are defined in NBL as `Class(x, _): defines Class x scope for Field, Method for the named variant or Block(_): anonymous scope for Variable for the unnamed variant`. An example of a named vertical scope is a class that scopes fields and methods, as seen in Figure 4.4. The paths of field `A` and method `B` are extended with the identifier `X` of the parent scoping class. Unnamed vertical scopes behave the same way, with the exception that a unique identifier is generated since the scope doesn’t have a name.

![Figure 4.4: Example of an AST with named vertical scoping. A class scopes fields and methods. The rectangular area is a vertical named scope from Class "String" Rounded rectangles are terms.](image)
4. LANGUAGE-PARAMETRIC NAME RESOLUTION

example, in Figure 4.5 the unique identifier anon_1 is generated and is used to extend the path of both local variables A and B.

Horizontally scoped terms on the other hand add an unnamed scope to the term itself and all right-sibling terms. The horizontally scoped term is then defined in the new unnamed scope, it is not defined in the preceding scope. These are defined in NBL as \texttt{Var(x, _): defines Variable x in subsequent scope}.

An example of a definition that defines a horizontal scope is a local variable, as seen in Figure 4.6. The unique identifier subseq_1 is generated and is used to extend the path of the local variable itself and subsequent definitions. Horizontal scopes are always unnamed because they do not really scope subsequent terms. Only the visibility of the horizontally scoped term is changed so that it is not visible in the preceding scope.

Vertical and horizontal scoping is frequently combined, as seen in Figure 4.7.

Scopes and Namespaces

Vertical scopes only scope certain namespaces, such as a class that scopes fields and methods. It does not scope variables or namespaces. If a method has a parent class (which it should have according to syntax), the path of the method is extended with the name of the class, to indicate that the class scopes the method. If a variable would
Figure 4.7: Example of an AST with vertical and horizontal scoping combined. A method vertically scopes local variables, local variables horizontally scope itself and subsequent definitions. The outer rectangular area is a vertical scope from Method "M". The inner dotted rectangular areas are horizontal scopes for their variables.

appear in a class (even if the syntax does not allow it), the path of variable is not extended with the name of the class, since it is not scoped. In most cases, the syntax and scope rules align. Variables and namespaces cannot occur in classes, so they can never be scoped in classes anyway.

Conceptually this means that each term has multiple active paths depending on the namespace. For example, see Figure 4.8. In this figure, each term has a mapping from namespaces to their active path. The path for the method namespace inside a class includes the name of the class, because classes scope methods. The path for the variable namespace inside a class does not include the class name, because classes do not scope variables. However, the path for the variable namespace in a method does include the class name in addition to the method name, because classes scope methods and methods scope variables.

Horizontal scopes act the same way as vertical scopes, with the namespace of the definition as scope. So \texttt{Var(x)}: defines \texttt{Variable x} in subsequent scope only scopes variables.

To create and modify these mappings in Stratego, we define a extend-mapping and replace-mapping strategy. The extend-mapping strategy takes a list of namespaces to extend the path of, the term to extend the path with and the existing mapping to update. If a namespace is not in the mapping yet, it is added with an empty list. For example, the following application will update the mapping from the Class term from Figure 4.8 to the Method term.

\[
\langle \text{extend-mapping} \left( \left[ \text{Method}, \text{Field} \right], \text{"String"} \right) \rangle \\
\Rightarrow \left[ \text{Method} \rightarrow \left[ \text{"String"} \right], \text{Field} \rightarrow \left[ \text{"String"} \right], \text{Var} \rightarrow \left[ \right] \right]
\]

The replace-mapping performs a similar operation, but instead of extending the path it replaces it. The following application updates the mapping from the Method term to the Var term.

\[
\langle \text{replace-mapping} \left( \left[ \text{Var} \right], \left[ \text{"String"}, \text{"Substr"} \right] \right) \rangle \\
\Rightarrow \left[ \text{Method} \rightarrow \left[ \text{"String"} \right], \text{Field} \rightarrow \left[ \text{"String"} \right], \text{Var} \rightarrow \left[ \right] \right]
\]
4. Language-Parametric Name Resolution

Figure 4.8: Example of an AST with for each term a list of namespaces mapping to the active path at that term.

| Var() -> ["String", "SubString"] |

Unique Definitions

Besides taking into account scopes and namespaces, unique definitions also need to be taken into account. In NBL, definitions default to being unique, but this can also be explicitly stated using \texttt{Class(x, _): defines unique Class x}. Incorrect results may be encountered when duplicate definition errors occur, if not taken into account.

For example, consider two classes \(A^1\) and \(A^2\) with the same identifier \(A\) in Figure 4.9 which causes a duplicate definition error. Class \(A^1\) contains a field named \(x\), \(A^2\) contains a field named \(y\). Methods in \(A\) can then use both \(x\) and \(y\), because both fields are scoped by classes with the same identifier \(A\). However, only \(x\) should be visible in \(A^1\) and \(y\) in \(A^2\). To solve this problem, all scoped definitions inside a unique definition have their path extended with a unique identifier, seen in Figure 4.10. This way fields in \(A^1\) are distinguishable from \(A^2\) because their path contains a different unique identifier.
Figure 4.9: Example of an AST with duplicate definitions. Both fields can be accessed through identifier A.

Figure 4.10: Example of an AST with duplicate definitions. Fields in A\textsuperscript{1} and A\textsuperscript{2} are distinguishable by using a unique identifier.
Algorithm

The algorithm that calculates the output from the previous example is defined in Aster. It consists of a part that calculates definition paths, and a part that turns paths into keys.

Aster provides means of retrieving the parent and sibling terms, as seen in Figure 4.11. This is in contrast to Stratego in which only child (subterms) can be retrieved. The idea of calculating a path for a definition term is recursively tracing its siblings and parents to the root node. During the trace, names are added to the path until the root node is reached.

The path calculation in Aster is split into several attributes. Some attributes are defined on the name of a definition, for example on "String" in \texttt{Class("String", ...)}, while others are defined on the entire term. This can also be seen in the NBL to Aster code generation. For example \texttt{nbl-scope-name} is defined on the entire term, \texttt{nbl-def-sites} is defined on the name. This distinction is important, as it determines which attributes are defined for a given term.

First, for each term in the AST, attributes that calculate the paths are defined. To calculate the mapping between namespaces and their active paths, the \texttt{nbl-paths} attribute is used:

\begin{verbatim}
  def nbl-paths =
      id.prev-sibling.nbl-horizontal-paths <+
      id.parent.nbl-vertical-paths <+
  ![](
\end{verbatim}

The attribute takes into account both horizontal and vertical scoping. First it tries to get the previous (left) sibling of the current term to calculate horizontal scoping. If the current term does not have a left sibling, it tries to retrieve the parent to calculate vertical scoping. If a term has no left sibling nor parent, it is the root term and the mapping is empty.

Horizontal path calculation takes into account definitions that are defined in subsequent scope.

\begin{verbatim}
  def nbl-subsequent := Subseq(<newname> "subseq")
\end{verbatim}
Name Resolution Algorithm

```python
def nbl-horizontal-paths =
    left := id;
    if ns* := left.nbl-subseq-ns then
        <extend-paths(|ns*, left.nbl-subsequent)> left.nbl-paths
    else
        left.nbl-paths
    end

If the left sibling is defined in subsequent scope, a unique identifier generated by evaluating nbl-subsequent and is used to extend the path. If the left sibling does not horizontally scope, the mapping from the left sibling is just copied.

Vertical path calculation takes into account (unique) named or unnamed definitions and their scoping.

def nbl-anonymous := Anon(<newname> "anon")
def nbl-vertical-paths =
    parent := id;
    paths := parent.nbl-paths;
    if ns* := parent.nbl-scoped-ns then
        if name := parent.nbl-scope-name then
            [path|_] := name.nbl-def-paths;
            <replace-paths(|ns*, path)> paths
        else
            <extend-paths(|ns*, node.nbl-anonymous)> paths
        end
    else
        paths
    end

If the parent does not scope anything (nbl-scoped-ns is not defined for the parent term), the namespace-path mapping is not modified and just copied. If the parent does scope, it is either a named or unnamed definition with a list of namespaces (ns*) that it scopes. For named scopes the path must be replaced since the path might have changed in between, as seen with the variable example earlier. The path in the mapping is replaced by the first path returned by nbl-def-paths which is defined on the name of the definition. This attribute will be explained below. For unnamed scopes, the path can be extended since the scope is unnamed and is not updated with a new name.

To get the path of a term instead of the namespace-path mapping, the nbl-path attribute is defined:

def default([[]]) nbl-path(|ns) =
    id.nbl-paths.clookup(|ns)

Given a namespace, it will retrieve the path of that namespace in the mapping of the current term. If a path cannot be retrieved, an empty path is returned by default.

The previous attributes calculated the mapping and paths for terms. These paths only contain scopes, not names of the actual definition. To get the full path of a definition site, including its name, an attribute defined on names is required.

def nbl-unique := Uniq(<newname> "unique")
def nbl-def-paths:
    name → name.nbl-def-sites.map(<conc> (segment, id.nbl-path(|ns)));
    where
        ns := name.nbl-def-ns;
        if unique := name.nbl-unique then
            segment := [name, unique]
```

63
4. LANGUAGE-PARAMETRIC NAME RESOLUTION

```plaintext
else
    segment := [name]
end
```

The `nbl-def-paths` attribute creates a full path by adding the name of the definition site to the active path for that definition. Since a definition site can create multiple definitions, a list of paths is returned instead of one. For unique definitions a unique identifier is added to the path as well to make definitions with the same name distinguishable.

Now that path for each definition is known, a URI and definition key can be constructed:

```plaintext
|def nbl-uris:
    name → nbl-def-paths.map(!([ns|id]))
where
    ns := name.nbl-def-ns;
|def nbl-defs = nbl-uris.map(!Def(id))
```

The URI is simply a composition of the namespace and path of the definition. A definition key is constructed using the `Def` constructor, that contains the URI.

The only operations required now are adding all definition keys to the index and annotating the AST:

```plaintext
|def collect-bagof after(concat) nbl-all-defs := id.nbl-defs
|def rewrite-bu nbl-annotate-defs =
    if defs := id.nbl-defs then
        <set-annotations> {id, defs}
    end

nbl-definition-stage:
ast1 → (ast2, defs)
where
    defs := ast1.nbl-all-defs;
    <add-all> defs;
    ast2 := ast1.nbl-annotate-defs;
```

The `collect-bagof` decorator on `nbl-all-defs` modifies the attribute to be evaluated on the entire AST where the results are aggregated into a list. Afterwards `after(concat)` concatenates all lists into one, since `nbl-defs` returns lists. Similarly, `rewrite-bu` on the `nbl-annotate-defs` attribute modifies the attribute to do a bottom-up rewrite over the AST. The entire definition stage is then composed by the `nbl-definition-stage` Aster rule (not an attribute) which adds all definitions to the index and annotates the AST with definitions.

4.4.2 Data stage

The data stage is concerned with collecting data about definitions and persisting this information to the index. In particular, type information and import information is collected, since these are required in the resolve stage. However, user defined data can be stored as well.

The evaluation of type information is done as follows:

```plaintext
|def nbl-type-key =
    !Type(id.nbl-uri, id.nbl-type)
|def collect-bagof nbl-all-type-keys := id.nbl-type-key
```
For each term that has a nbl-uri and nbl-type attribute, the nbl-type-key attribute contains a key that can be stored in the index. A collect attribute is used to retrieve all type keys.

The evaluation of imports is more complex because of the different import configurations.

```plaintext
class A {
```
4. Language-Parametric Name Resolution

```java
int b;
int m(int b) {
    return b;
}
```

From the definition stage, the following definitions are stored in the index for this program:

- Def([Class(), "A"])
- Def([Field(), "b", "A"])
- Def([Method(), "m", "A"])
- Def([Variable(), "b", "m", "A"])

The `b` in `return b` can either refer to the parameter of method `m`, or the field of class `A`. When performing a lookup for `b`, it starts by looking for either a field or variable called `b` in the current scope: method `m` inside class `A`. This comes down to two get operation on the index:

```
<get> Def([Variable(), "b", "m", "A"]) ⇒ [Def([Variable(), "b", "m", "A"])]  
<get> Def([Field(), "b", "m", "A"]) ⇒ []
```

The first call to get succeeds because it can be found in the index, so the definition for `b` is found. If for example, the parameter was called `c` instead of `b`, both get operations would return an empty list and the lookup will try a smaller scope: class `A`

```
<get> Def([Variable(), "b", "A"]) ⇒ []  
<get> Def([Field(), "b", "A"]) ⇒ [Def([Field(), "b", "A"])]
```

In this case, the second call to get will succeed and the reference is resolved to the field.

Requirements  Almost all editor services require some mechanism to resolve references to definitions in some way, most with different requirements. Some language constructs also have requirements for resolving references.

Reference resolution  IDE reference resolution ("Go to definition") requires a lookup to one or more definition with exactly the name of the reference. In most cases a reference resolves to one definition site, but multiple definition sites are also possible in the case of non-unique definitions such as partial classes. Currently, Spoofax only supports reference resolution to one definition site, so only the first definition is taken.

Code completion  Code completion requires resolving to all visible definitions that begin with (or contain) a certain string. This string comes from a partial identifier in a program. For example, `Str` can complete to `String` and `StringBuilder`. This is different from reference resolution, where we are only interested in exact matches.

Constraint checking  For constraint checking it matters if imported definitions are taken into account. For example, if you define 2 classes with the same name in C#, this should trigger a duplicate definition error. On the other hand, consider the situation where a class has the same name as a class in another namespace that is imported. This
Name Resolution Algorithm

is not a duplicate definition error on the definition of the class, since they are in different namespaces and thus have a different fully qualified name.

However, if imports are taken into account, the lookup would return 2 class definitions with the same name. This would then trigger a duplicate definition error on both classes, even though both classes have a different fully qualified name. Instead, an ambiguity error should be triggered on use sites. In other words, some constraint checks require imports while others do not.

Type system
Types also need to be taken into account during lookup.

For example, with property access, the default lookup strategy illustrated earlier does not work since property access is not part of lexical scoping. Instead, a lookup inside the class of the left hand side expression has to be performed.

Algorithm
The solution is to have one lookup algorithm with strategy parameters that can be varied to cover all concerns.

The first concern is supporting both exact and prefix name matching so that exact reference resolution and code completion can use the same lookup algorithm. This is done by parameterizing the lookup algorithm with a name comparison strategy. Given an index entry in Stratego form (key), the strategy decides whether the key matches the target definition(s) to look up. Disabling imports is simply done by adding a strategy parameter that enables imports when id is passed and disables imports when fail is passed. To support language constructs such as property access, adjust rules can be added that influence the paths the lookup will search in.

The entry point of the algorithm is the lookup strategy.

```
lookup(import, adjust, compare|namespace, path, name):
  x → defs3
  where
defs1 := <lookup-path(import, adjust, compare|namespace, path, name)> x;
  if [parent-path] := path then
    defs2 := <lookup(import, adjust, compare|namespace, parent-path, name)> x;
defs3 := <conc> (defs1, defss2)
  else
defs3 := defss1
end
```

The lookup strategy is a recursive algorithm that collects definitions and keeps calling itself until the root (empty) path is reached. It has several strategy and term parameters that modify the behaviour of the lookup:

import Set to id to enable imports, fail to disable imports. Used to prevent infinite recursive imports.

adjust Set to id to enable adjust rules, fail to disable adjust rules. Used to prevent infinite recursive adjust rules.

compare Name comparison strategy that is used to filter out unrelated definitions.

namespace Target namespace to search in.
4. LANGUAGE-PARAMETRIC NAME RESOLUTION

**path**  Target path to search in.

**name**  Name to search for.

Definitions from the index are retrieved by calling `lookup-path` which returns all definitions inside a certain path and namespace with a certain name.

```
lookup-path (import, adjust, compare | namespace, path, name):
  x → defs
  with
    defs := <do-adjusts-imports (import, adjust, compare | namespace, path, name)> x
  <+ 
    defs := <filter (compare) > <index-get-children>
    Def {[namespace | path]}
```

The lookup-path strategy first tries to perform lookup using adjust rules and imports with `do-adjusts-imports`. If this strategy does not fail, it returns a filtered list of definitions that come from adjust rules or imports. If no adjustments or imports are performed, the `index-get-children` operation is used to retrieve all definitions inside given namespace and path. We do not want to return all definitions, so a filter using the name comparison strategy is performed. After filtering, `defs` only contain definitions that correspond to the target names the lookup is searching for.

For exact matching, the following name comparison strategy is used:

```
compare-exact (| name):
  entry → <id>
  where
    uri := <index-uri> element;
    <eq> (<index-uri-name> uri, name)
```

Given an entry from the index and the name to match for, it retrieves the name for the entry and compares it to the given name.

The comparison strategy for code completion is similar, but uses substring matching instead of exact matching:

```
compare-substring (| prefix):
  entry → <id>
  where
    uri := <index-uri> element;
    <is-substring (|prefix)> <index-uri-name> uri
```

To perform an exact lookup for variable `b` in the example C# program at the position of the return statement, the following lookup call is performed:

```
lookup (id, id, compare-exact (|"b") | Variable(), ["m", "A"], "b")
```

Performing code completion for classes starting with `Str` at the position of the class uses the following lookup call:

```
lookup (id, id, compare-substring (|"Str") | Class(), ["A"], "Str")
```

**Adjust rules**  Adjust rules are used to change the behaviour of the lookup algorithm if the standard behaviour does not suffice. They are typically used in cases where name binding does not follow lexical scoping, such as property access. They are also used in cases where a program element can refer to multiple different definitions, such as variable references that can point to either local variables or class fields. Some adjust rules are generated from NBL, but they can be written manually as well if the behaviour from NBL does not suffice.
Field access requires interaction with the type system to look up a field inside a class. In the MiniC# NBL definition, field access is defined as follows:

```
FieldAccess(e, f) :
  refers to Field f in Class c
  where e has type ClassType(c)
```

An adjust rule for the MiniC# field access construct looks like this:

```
adjust-index-lookup(target-name|namespace, path, prefix):
  FieldAccess(e, f) -> defs
  where
    <target-name> f;
    ClassType(t) := <type-of> e;
    defs := <index-lookup-children(Property(), prefix)> t
```

Adjust rules are called by the lookup algorithm with the namespace and path it is looking into and the prefix (f) of the definition it is looking for. It matches a property access constructor with a left hand side expression and the name of the property to access. The target-name strategy is used to confirm that the name from the property access actually matches the name we are looking for. This is required because adjust rules are called multiple times on different terms in the program. Without this check it could resolve to the name of another property access. The type is retrieved using the type-of strategy, which given an expression returns the type of that expression. If it is successful, a class type is returned. The index-lookup-children strategy is then used to retrieve all definitions in namespace Property() that start with given prefix inside the retrieved class type. A prefix is used here so that both code completion and exact matching works with this adjust rule. Definitions returned from index-lookup-children are returned and the lookup uses these definitions instead of the definitions that would be found with a regular lexical lookup.

An adjust rule for the case where variable references can point to both local variable and class fields looks like this:

```
VarRef(v) => [[Variable()|path], [Field()|path]]
  where
    <target-name> v
```

This adjust rule matches variable references and adjusts the lookup to look for both local variables and class fields. Instead of returning definitions, this adjust rule returns a list of URIs. The difference is that when a definition is returned from an adjust rule, the lookup assumes that the returned definitions are looked up and correct. If a URI is returned, the lookup will recursively do a lookup inside the namespace and path of the URI. In this case, we do not know if a variable or field exists so we just delegate the lookup by returning URIs instead of definitions.

This adjust rule is generated from the following NBL definition:

```
VarRef(v) :
  refers to Variable v
  otherwise refers to Field v
```

Nested references are another common occurrence that need an adjust rule. The following NBL rule uses a nested reference to refer to a definition inside the reference:

```
ModHolderRef(m, ug):
  refers to Mod m in Group g
  where ug refers to Group g
```

The adjust rule generated from this looks like this:

```
adjust-index-lookup(target-name|namespace, path, prefix):
  ModHolderRef(m, ug) => [[Mod(), Group()|path], [Group()|path]]
  where
    ug refers to Group g
```
4. LANGUAGE-PARAMETRIC NAME RESOLUTION

\[ \text{ModHolderRef}(m, ug) \rightarrow \text{defs} \]

where
\[ \text{<target-name>} \ m; \]
\[ \text{Def}(g) := \text{<index-lookup>} \ ug; \]
\[ \text{defs} := \text{<index-lookup-children} (\text{|Group()}, \text{prefix}) \ ug\{g\} \]

This is very similar to the field access adjust rule where a lookup into the type is performed. In this case a lookup is performed instead of an interaction with the type system.

**Imports** Imports are processed after adjust rules and simply add extra URIs that the lookup will search in, like the variable reference adjust rule does. The strategy for imports is generic, it uses \text{Import} entries in the index that are added during the data stage:

\[ \text{adjust-index-import}(\text{|namespace}, \text{path}): \]
\[ \_ \rightarrow \text{allImports} \]

where
\[ \text{uri} := \text{|namespace|path}; \]
\[ \text{entries1} := \text{<imports}(\text{|uri})>; \]
\[ \text{entries2} := \text{<partition-imports}(\text{|namespace})>; \]
\[ \text{allEntries} := \text{<conc} \text{ (entries1, entries2)} \]
\[ \text{allImports} := \text{<mapconcat (import-uris (| namespace, \text{|uri})>) allEntries} \]

\[ \text{imports}(|\text{uri}) = \]
\[ \text{<index-get > Import (\text{uri}, (), (})} \]

\[ \text{partition-imports(|namespace)} = \]
\[ \text{<index-get > Import (\text{|namespace, <current-partition>}, (}, (})} \]

\[ \text{import-uris(|namespace, seen}): \]
\[ \text{Import (|_|path, key, type) \rightarrow [uri]} \]

where
\[ \text{<not (?Trans( ))> type} \]
\[ \text{path} := \text{<index-uri-path} \ key; \]
\[ \text{uri} := \text{|namespace|path} \]

The \text{adjust-index-import} strategy is called by the lookup algorithm with the current namespace and path it is looking into. A prefix or name is not required here, since imports only change the namespace and path to look into. All \text{Import} entries that are active at the given uri and partition are retrieved from the index using \text{import} and \text{partition-imports}. Then, \text{import-uris} is applied to each import entry to change the namespace of the import to the namespace of the definition we are looking for. For example, a class imports its parent (inherited) class. The namespace of the import would then be \text{Class()}. However, the lookup is looking for fields inside that class, so the namespace is changed to \text{Field()}. This supports normal imports such as class inheritance, and partition scope imports such as C# namespace imports.

To support transitive imports, such as a class inheritance chain, a recursive version is added:

\[ \text{import-uris(|namespace, seen)}: \]
\[ \text{Import (|_|path, key, Trans()) \rightarrow <conc} \text{ (\text{|uri}, imports)} \]

where
\[ \text{path} := \text{<index-uri-path} \ key; \]
\[ \text{uri} := \text{|namespace|path}; \]
Name Resolution Algorithm

if not <elem> (uri, seen)) then
  entries := <import-uris(|uri)>;
  imports := <mapconcat{|import-defs(|namespace, [uri|seen])}> entries
else
  imports := []
end

Transitive imports could cause loops, for example class A : B class B : A. To prevent loops, the seen term parameter contains a list of URIs that were already visited. If an URI is found that has already been visited, the recursion stops. Otherwise, it recursively applies itself until no more transitive imports are found and the first (non-recursive) import-uris is applied.

Resolving the AST All references in the AST are resolved to their definition using the lookup algorithm. Because this is a traversal over an AST again, Aster is used.

The nbl-use attribute only fires on use sites. It produces either a Use key with a resolved uri to the definition, or a BadUse indicating that the reference could not be resolved.

The resolve stage is wrapped up similar to the definition stage, references are stored in the index and the AST is annotated with resolved and unresolved references:

def nbl-use =
  namespace := id.nbl-use-ns;
  path := id.nbl-use-site.nbl-path(|namespace);
  uri := [namespace, id|path];
  term := id(uri)
  if Def(def-uri) := <index-lookup> term then
    !Use(def-uri)
  else
    !BadUse(uri)
end

The reason that the annotations are on names and not the terms is because of how reference resolution works in Spoofax. Users of the IDE want to use ‘go to definition’ on the name of a class to jump to the definition of that class. If the annotations were on the class term itself, users would have to click on the class itself which is not intuitive.

Besides resolved references, this stage also produces dependency information for the AST. Remember that index-get and index-get-children from the library in Section 3.4.3 store reads in the index. The lookup algorithm uses both these strategies.
4.4.4 Dependency stage

The dependency stage computes the partitions that have to be re-analyzed by changes made to the current partition. In Section 3.4.4 we defined a protocol for finding dependencies which we use to construct the algorithm. In the protocol, the difference is taken between definitions and data entries in the current partition before the algorithm started, and after the resolve stage ended. All partitions that depend on entries in the difference could be affected by the changes and have to be re-analyzed.

Because this stage requires a change to the entire name resolution algorithm, we also show how the algorithm looks like when all stages are combined.

```
name-resolution: (ast1, partition) → (ast6, defs, data, uses, reanalyze2, partition)
where
<set-partition> partition;
revision := <start-transaction>;
before := <get-partition-entries> partition;
clear;
ast2 := ast1.init;
(ast3, defs) := <nbl-definition-stage> ast2;
(ast4, data) := <nbl-data-stage> ast3;
(ast5, uses) := <nbl-resolve-stage> ast4;
ast6 := ast5.uninit;

// Find inconsistencies
entries := <get-partition-entries> partition;
partitions := <index-dependencies> entries;
revs := <map( get-partition-revision )> reads;
if <getfirst (r → (r, revision) | gt)> revs then
  inconsistent := [partition]
else
  inconsistent := []
end;

// Find dependencies
after := <filter(diff-filter)> <conc> (defs, data);
changed := <diff(entry-eq)> (before, after);
reanalyze1 := <remove-all(?partition)>
  <index-dependent> changed;
reanalyze2 := <conc> (reanalyze, inconsistent);
commit-transaction
```

Given an AST and partition, name-resolution returns an annotated AST, all definition, data and use entries and a list of partitions to re-analyze. It follows the protocol for thread safety, inconsistencies and dependency analysis. The only difference is that the diff strategy application is parameterised by the entry-eq strategy. This strategy decides whether a given tuple of 2 elements is equal. If they are not equal, the strategy must fail. By default it uses the builtin eq strategy, but this is not always a valid strategy in some cases. For example, if there is an unnamed scope, a unique identifier is generated to identify that scope. However, the generated identifier is randomly chosen and is not deterministic between name resolutions. In the first name resolution the definition would be Def([Variable(), Anon("1"), "a"]), but in the second one Def([Variable(), Anon("2"), "a"]). This would mark local variable definitions inside a scope as being different every time.
To resolve this, entry-eq defines custom equality checks for Def, Type and Import entries:

```haskell
entry-eq:
  (Def(u1), Def(u2)) → <id>
  where
    <uri-eq> (u1, u2)

strip-uri:
  u1 → u2
  where
    u2 := <remove-all(?Anon(_)) <+ ?Subseq(_)) <+ ?Uniq(_))> u1

uri-eq:
  (u1, u2) → <id>
  where
    <eq> (<strip-uri> u1, <strip-uri> u2)

entry-eq:
  (Type(u1, v1), Type(u2, v2)) → <id>
  where
    <uri-eq> (u1, u2);
    <eq> (v1, v2)

entry-eq:
  (Import(u1, v1, t1), Import(u2, v2, t2)) → <id>
  where
    <uri-eq> (u1, u2);
    <eq> (v1, v2);
    <eq> (t1, t2)
```

Checking equality of URIs is done using uri-eq which simply compares the URI without non-deterministic terms. A custom equality strategy also has the benefit that other entries such as Use(x) are filtered out automatically since entry-eq is not defined for a tuple of Use(x). We are only interested in Def, Type and Import entries because it is only possible to look up definitions and data about definitions. Thus it is only possible to depend on definitions and data about definitions.

### 4.5 Parallel Name Resolution

Both the infrastructure and name resolution algorithm are made in such a way to support parallel analysis. These preparations make the actual parallel analysis relatively simple.

Stratego contains a library called Stratego-parallel for high-level concurrency with strategies. It adds a new strategy called parallel-unordered that parallelizes the builtin all strategy. The parallelized all strategy applies a strategy in parallel on each term in a list of inputs, collects all results and returns them when they have all finished.
The following Stratego code performs parallel name resolution:

```stratego
def parallel-analyze : files → allResults
  where
  asts := <map{do-parse-file}> files;
  results := <parallel-all(name-resolution)> asts;
  reanalyze := <make-set> <mapconcat (get-reanalyze)> results;
  if not ([] := reanalyze) then
    newResults := <parallel-analyze> reanalyze;
    allResults := <conc> (results, newResults)
  else
    allResults := results
  end

def do-parse-file : file → (ast, file)
  where
  ast := <parse-file> file

def parallel-all(s) = parallel-unordered(all(s))

def get-reanalyze : (_, _, _, _, reanalyze, _) → reanalyze
```

The `parallel-analyze` strategy takes a list of files and returns the analyzed results of these files. All files are first parsed from their textual form into ASTs. Invoking the parser is not thread safe, so this cannot be done in parallel. Then, `parallel-all` is invoked with the `name-resolution` strategy, which is thread safe. This performs name resolution for multiple files in parallel and returns a list of name resolution results. Inconsistencies and changed dependencies are resolved by recursively applying the strategy until there are no more files to reanalyze.

It is also possible to perform editor services such as constraint checking in parallel. Constraint checks typically only depend on data from the index, which is thread safe in transactions. The `parallel-analyze` strategy can easily be modified to include extra steps which are run in parallel automatically.

### 4.6 Evaluation

The name resolution algorithm and name-dependent analyses are evaluated by applying them to several case studies and measuring the results. We evaluate the editor services, overall performance and performance gains (or losses) by incremental and parallel analysis.

#### 4.6.1 Case Studies

The case studies from Chapter 2 are used in the evaluation, in particular the Entity language. Because we are not using languages that are use in practice, we have no existing programs for use in evaluation. A source file generator is created for the Entity language to generate programs. The generator can generate a variable number of source files with (or without) dependencies on each other. The number of files, definitions and references can be adjusted to evaluate how this influences the name resolution.
During development a lot of changes to the infrastructure and name resolution algorithm were made. Therefore we created unit tests for each language using the Spoofax testing language [23]. Editor services such as reference resolution, code completion and error checking are tested to produce the expected results. For example, a reference is always resolved to a certain definition, an incomplete reference always completes to a certain reference and a piece of code produces a number of (or no) errors. If any changes cause tests for a language to fail it is an indication that the change is incorrect or not generic enough.

4.6.2 Analysis Performance

We measure the overall speed of the analysis under varying conditions such as different number of files, definitions, references and dependencies. A baseline set of files are generated that have on average 150 lines of code consisting of entities, properties, functions, statements and aspects. To measure the effect that the number of files has, 4 sets are generated with varying numbers of files; 10, 50, 100 and 250 files.

First, the performance of the baseline and influence of extra definitions is measured. Another two sets of files are generated, one with 5 times the number of definitions compared to the baseline, and one with 10. The performance is measured by adding the entire set of files to a clean project, triggering a batch analysis of all files, and taking the time it took to analyze all files. The time taken includes the name resolution algorithm from this chapter, parsing, type analysis, error checking and updating error markers on source files. In these benchmarks, the same computer from the benchmarks in Section 3.5.3 is used. The analyses in this subsection are executed in parallel according to Section 4.5. The results are plotted in Figure 4.12.

With 100 files, the baseline set of files is parsed, analysed and updated in about 13 seconds. However, with 250 files, 74 seconds are required. This is caused by the performance of the generic lookup algorithm depending on the number of entries in
The lookup algorithm must work with both exact lookups and prefix lookups for code completion. To support this, instead of performing hash lookups on the index, it looks up all children and filters those instead, which is a linear operation. We think this performance issue could be resolved by constructing a separate efficient lookup algorithm for exact matches. In any case, the analysis of all 250 files would only have to be done once when the project is opened for the first time. Subsequent changes are incremental and only take time proportional to the change.

Comparing the baseline to files with more definitions, only a small decrease in speed is incurred. More definitions incur more calculations in the definition stage, and slightly more in the resolve stage because of the mentioned lookup problem. We expect more references to incur a larger hit on performance, since lookups are more expensive than calculating keys for definitions. In Figure 4.13 the baseline is compared to files with 5 and 10 times the number of references.

As expected, the performance hit is much larger. Files with 10 times the number of references take more than double the time to parse, analyze and update than the baseline.

To measure the impact of dependencies, a new set of files is generated that has no dependencies across files and only very minimal dependencies within the file. The results of this measurement are found in Figure 4.14.

The graph makes it clear that dependencies incur a large performance hit. This is not unexpected, dependencies between files (partitions) cause reanalysis and possible inconsistencies which also case reanalysis. Not having dependencies means that only one pass over all files is performed, which is faster.
Figure 4.14: Benchmark results for baseline and no dependencies.
4.6.3 Parallel Performance

We also measured the performance of parallel analysis in relation to sequential analysis. The same files from the analysis performance section are used, but are also performed sequentially. First, the performance of the baseline in parallel and sequence is measured. The results are plotted in Figure 4.15.

![Graph showing performance comparison between parallel and sequential analysis.](image)

Figure 4.15: Benchmark results for parallel and sequential baseline.

Parallel analysis is roughly twice as fast as sequential analysis up to 100 files. Interestingly, the normalised difference becomes smaller when the number of files increases. For 100 files, parallel analysis is 99% faster, for 250 files it is only 60% faster. The results of the files with an increased number of definitions in Figure 4.16 are identical. However, the results of files with increased number of references in Figure 4.17 are different.

In this graph, the differences in measurement between parallel and sequential analysis are larger. For 100 files, parallel analysis is 135% faster, for 250 it is 111% faster. The speedup decreases as more files are added, but is initially much higher. In any case, the speedup should stay roughly constant if the number of files increases, with an exception for a low number of files. Also, on a machine with 4 cores, a higher speedup should be attainable.

The Spoofax-parallel library is used to run the semantic analysis in parallel. It uses a thread pool in Java to execute strategies in parallel on different threads. We inspected the run-time behaviour of the library using the jvisualvm JVM profiler. It includes a thread view where all threads are listed and if a thread is waiting, running or blocking. When performing the analysis, not all threads seem to be working all the time, but there are almost no threads blocking. In fact, a lot of threads are waiting.

This is partially caused by the Stratego code that performs the parallel analysis. The algorithm analyses all files in parallel and waits until all files have completed. After the first set of files have completed, the files that have to be reanalyzed are analysed again. This is done to decrease the overhead and the total number of reanalyses that
Figure 4.16: Benchmark results for parallel and sequential baseline with increased number of definitions.

Figure 4.17: Benchmark results for parallel and sequential baseline with increased number of references.

are performed. For example, if file A depends on 10 files, analyzing those 10 files would trigger a reanalysis of A. If re-analysis is done immediately, A is analysed 10 times. Instead we wait until analysis is completed and only reanalyze the set of files, preventing duplicates. However, this waiting causes some threads to stall as they run out of work. The speed of the parallel analysis is then bound by the thread that has the longest completion time for all its jobs. Furthermore, parsing is executed sequentially and precedes the parallel analysis, causing more threads to wait.

A solution is to have a parallel job queue that behaves like the current job queue in
Stratego, but executes jobs in parallel. Files that have to be re-analyzed can be added to the job queue immediately. If the file is already scheduled for analysis, a new job is not added. This is not possible with the Spoofax-parallel library because it is not possible to add strategies to the thread pool on the fly.

In conclusion, we are satisfied with the performance gains from parallel analysis. It shows that it is indeed possible to perform name resolution and name-dependent analyses in parallel. Changes to the parser to allow parallel parsing and a parallel job queue will improve the performance gains even more. However, these changes are outside of the scope of this thesis, but are interesting topics for future work.
Related work

There are many other approaches for specifying and implementing incremental and parallel name resolution. We give an overview of such approaches and discuss how they relate to our approach. The main point of difference compared to other approaches is that we make abstractions for the name binding domain using NBL, thus hiding the low level details of writing name resolution implementations. Only the roles of language constructs have to be declared, the resolution mechanics are left to code generation and generic name resolution algorithm. The index infrastructure and generic name resolution algorithm serve as an implementation of NBL. Because of these abstractions over the name binding domain, we can generate editor services which would otherwise require substantial effort to manually implement.

In this chapter, related work on NBL, the index infrastructure and the generic name resolution is discussed.

Symbol Tables

In classic compiler construction, symbol tables are used to associate identifiers with information about their definition sites. This typically includes type information. Symbol tables are commonly implemented using hash tables where the identifiers are indexed for fast lookup. Scoping of identifiers can be implemented in a number of ways; for example by using qualified identifiers as index, nesting symbol tables or destructively updating the table during program analysis.

The type of symbol table influences the lookup strategy. When using qualified identifiers the entire identifier can be looked up efficiently, but considering outer scopes requires multiple lookups. Nesting symbol tables always requires multiple lookups but is more memory efficient. When destructively updating the symbol table, lookups for visible definitions are very efficient, but the symbol table is not available after program analysis. This approach also only works when name resolution never looks further down into the file, like in C, where all definitions must be declared before use.

The index we use is a generic symbol table that uses qualified identifiers. We map qualified hierarchical identifiers (URI’s) to information such as definitions, types and uses.
Inheritance Graph

The Inheritance Graph Model \cite{10} is an infrastructure for name resolution. Vertices in the graph represent visibility regions that contain definition and other data such as types. Visibility regions usually correspond to scopes in the language, such as a function, class or statement block. Edges represent inheritance between visibility regions and are used to model nesting of scopes.

While the graph representation is very different from our hierarchical URI representation, the idea is roughly the same. A URI can be an identifier, for example a URI to a variable, or a visibility region and identifier at the same time, for example a URI to a class. Symbol table entries hierarchically nested under a block URI are scoped by the declaration at their parent URI, similar to visibility region inheritance.

In both approaches, disallowing use before definition is accomplished by opening a new scope for every definition.

Attribute Grammars

Attribute Grammars \cite{28} (AG’s) are a formal way of declaratively specifying and evaluating attributes for productions in formal grammars. Attributes can either be specified synthesised or inherited. Synthesised attributes can use information in the node itself or in any of its children and move information upwards the AST. Inherited attributes can use information in the parents of the node it is specified on, parents then have the responsibility to provide a value for this attribute. These attributes can move information downwards.

Eli

In \cite{21}, an attribute grammar specification language with modular and reusable attribute computations that is part of Eli is proposed. Abstract computations of attributes can be specified. These computations are generic in the sense that they do not depend on a language. Abstract computations can then be reused in many languages by letting symbols from the target language inherit these computations.

For example a Range computation would calculate a scope, IdDef definitions and IdUse references. A FunDef could then inherit from Range and IdDef, because it defines a function and opens a scope. FunCall inherits from IdUse because it references a function.

This is not unlike NBL, instead of symbols we have ATerms that are pattern matched and instead of abstract computations we have naming concepts such as scopes, definitions, references and imports. However, the expressiveness differs. Where Eli can be used to specify general (and reusable) computations on trees, NBL is restricted to just name binding computations. This has the advantage that it is easier to understand and write NBL definitions, but as disadvantage that it is less flexible.

Silver

Silver \cite{34} is an extensible attribute grammar specification language. Their language can be extended with new general purpose and domain specific features. As an exam-
ple they extend their language with features such as auto-copying, pattern matching, collection attributes and support for data-flow analysis. However, name resolution is mostly done the traditional way; an environment with bindings is passed down the tree using inherited properties.

Reference Attribute Grammars

Reference Attribute Grammars (RAG’s) extend AG’s by introducing reference attributes; attributes that can reference any node. This makes it possible to reference nodes without strictly having to follow the tree structure, simplifying name resolution using AG’s.

JastAdd [13] is a meta-compilation system for generating language-based tools such as compilers, source code analyzers and language-sensitive editing support using RAG’s and object orientation. Evaluation of attributes is specified by declaring equations that can use information in the current, children, parent or referenced node, depending on the type of the attribute. Besides the reference attributes extension, JastAdd also extends attributes with parameters, allowing attributes to act as functions where the value depends on the given parameters.

In JastAdd, a typical name resolution as seen in [8,13,27] is specified by defining inherited lookup attributes parameterised by an identifier on nodes that have to be looked up, such as variable references. All nodes that can have a variable reference as a child node, such as a function body, then have to provide an equation for performing the lookup. These equations also have to take into account scoping and ordering. For example, the following code performs name resolution for a state machine language in JastAdd:

```java
aspect NameAnalysis {
    syn State Transition . source () = lookup (getSourceLabel ());
    syn State Transition . target () = lookup (getTargetLabel ());
    inh State Declaration . lookup (String label);
    eq StateMachine . getDeclaration (int i) . lookup (String label) {
        for (Declaration d : getDeclarationList ()) {
            State match = d . localLookup (label);
            if (match != null) return match;
        }
        return null;
    }
    syn State Declaration . localLookup (String label) = null;
    eq State . localLookup (String label) =
        (label . equals (getLabel ())) ? this : null;
}
```

When comparing the specification of name resolution, the JastAdd specification has more low level details than the NBL. Each lookup has to be written with attribute equations using Java code. However, this provides a lot of flexibility to handle all sorts of languages. We also think that our NBL definitions are easier to understand. For example it is very easy to see what a certain program element scopes by going to the corresponding rule and looking for a `scopes` definition. In JastAdd, this information is encoded within a set of equations and is usually implemented by returning early or late.

83
5. Related Work

Dynamic Rules

The environment passing style does not compose well with generic traversals, requiring much more complicated traversals and rewrite rules. As an alternative, Stratego allows rewrite rules to create dynamic rewrite rules at run-time [5]. The generated rules can access variables available from their definition context. Rules generated within a rule scope are automatically retracted at the end of that scope. Hemel et al. [16] describe idioms for applying dynamic rules and generic traversals for composing definitions of name binding, type analysis, and transformations without explicitly staging them into different phases. Our current work builds on the same principles, but applies an external index and provides a specialized language for name resolution.

Name resolution with scoped dynamic rules is based on consistent renaming, where all names in a program are renamed such that they are unequal to all other names that do not correspond to the same definition site. Instead of changing the names directly in the tree, annotations can be added which ensure uniqueness. This way, the abstract syntax tree remains the same modulo annotations. Furthermore, unscoped dynamic rewrite rules can be used for persistent mappings [24].

Language Workbenches

Xtext

Xtext [9] is a framework for developing programming and domain specific languages. It provides tools for parsing, name resolution, compilation, interpretation and also generates Eclipse editors with error checking, reference resolving, code completion and more. The Xtext Grammar Language is used to specify grammars for languages, but also the name resolution. By using name binding and cross-referencing in the grammar, references to other language elements are made. Use sites are then automatically resolved to their definition when a source file is analysed. For example, the following Grammar Language definition defines a grammar and simple name binding for classes that contain fields and methods.

```
Model : elements += Class*;
  Class :  'class' name = ID '{' 
     members += Field*
       members += Method*
             '}'

Field : type = [Class] name = ID ";";
Method : name = ID "()" "{" 
       statements += Assign*
              "";
Assign : left = [Field] "=" right = [Field] ";";
```

The `name = ID` part specifies that an identifier is used as a name for that language construct. It is equivalent with a `defines Class x` clause in NBL. References are defined using `[Class]`, which specifies that the name of a class is expected. This is equivalent with a `refers to Class x` clause in NBL.

However, scoping, import or other visibility rules cannot be defined in the Grammar Language, these have to be programmed using Xtend or Java code. A scoping API with some default resolvers are provided to assist with writing scoping procedures. For example field access, method calls, inheritance and block scopes would need custom
Xtend or Java code to function properly. Only package imports have special support and can be specified directly in the Grammar Language. Common constraint checks such as duplicate definitions, use before definition and unused definitions also have to be specified manually. This increases the amount of boilerplate code that has to be reimplemented for each language.

While a part of name binding is encoded in the Grammar Language in Xtext, NBL definitions are separate from SDF definitions in Spoofax. This separation allows us to specify more advanced name binding concepts without cluttering the grammar with these concepts. On an implementation level, we use a similar approach to Xtext with regard to infrastructure: a global index is used to store summaries of files. Summaries are made available through an API to write scoping definitions, editor services and transformations to other languages.

**EMFText**

EMFText [14] is another framework for developing domain specific languages. Like Xtext, it is based on the Eclipse Modeling Framework [32] and relies on metamodels to capture the abstract syntax of a language. While in Xtext this metamodel is generated from a concrete syntax definition, EMFText takes the opposite approach and generates a default syntax definition based on the UML Human-Usable Textual Notation [30] from the metamodel. Language designers can then customize the syntax definition by adding their own grammar rules. In the default setup, reference resolution needs to be implemented in Java. Only simple cases are supported by default implementations [15]. JastEMF [6] allows to specify the semantics of EMF metamodels using JastAdd RAGs by integrating generated code from JastAdd and EMF. EMFText further provides a prototypical support for EMF Attribute Grammars (EAGs) which are interpreted.

**Incremental Name Resolution**

In [12], an infrastructure and several techniques for language-independent incremental name resolution are shown. The infrastructure is a simplified version of the Inheritance Graph Model [10].

To do incremental analysis, a distinction between 4 kinds of changes to declarations are made: adding a declaration, removing a declaration, changing a declaration (for example changing the type data of a variable declaration) and changing the viability of declarations (for example changing the superclass of a class declaration).

For each of these kind of changes, there is a set of references that have to be rebound, or for which the information from the declaration has changed. They acknowledge that detecting which references have changed when a declaration is removed or changed are comparatively easy to handle, since they only affect directly their references. However, adding a declaration is a lot harder to handle. Affected sites can either be references for which there was no declaration, or references bound to declarations that are shadowed by the new declaration. Handling changing the visibility of declarations is also harder, because affected sites do not have a direct relation with the changed declaration.
5. Related Work

In our approach, changing the visibility does not occur since these changes either change the (data of a) declaration, or remove and add a declaration. For example changing the superclass will change the import data of that declaration and will trigger a change for every reference that depends on the import data. Changing a field from public to private will either remove the old field declaration and add a new one, because the namespace changed, or will trigger a change because the constraint checking rules depend on the field declaration.

Several approaches to finding the minimal set of changed references are given. The most powerful method, that finds the minimal set of changed references, is the Maintain Traces method. When looking up a reference in this approach, traces are left in the symbol table. Traces contain information about the identifier and the location of the reference that is being resolved. These traces can be used to efficiently find which references have changed when adding or removing a declaration, or changing the visibility of declarations. However, storing traces incurs a significant space overhead, especially if there are a lot of references (and thus a lot of lookups).

The idea of this approach is very similar to the approach used in this thesis. Each query on the index infrastructure leaves traces about dependencies in the index. When looking up a reference, the index is queried multiple times until the reference is found, or the root scope is reached. These traces can then be used to find the minimal set of changed references when a definition is added. Because traces are left behind on each query, changes will also be properly detected when other editor services like error checking query the index. The error checking editor service will query the index and leave traces. When anything the error checking depends on changes, only the dependent partitions are reanalysed.

Concurrency

Traditional concurrent programming models provide low level primitives, such as locks, to guarantee mutual exclusion as a concurrency control method. However, locks are complex, error prone and do not scale well. Using locks in an incorrect way causes deadlocks, stalling the entire program. If many locks are required, the use of them becomes more complex as deadlocks need to be prevented. Low level locks do not scale well because when a resource is locked, trying to access that resource (and thus locking it) entails waiting. The time spent waiting could be used to do something more useful.

Software Transactional Memory

Software Transactional Memory (STM) [31, 11] is a more high level approach to concurrency control than locking. In STM, access to shared memory is controlled by transactions, similar to transactions in databases. Modifications in transactions are isolated from other transactions and transactions are atomic. Transactions can fail if there are conflicts, for example if two transactions tried to write to the same shared memory.

The index infrastructure implements a variation on STM. Transactions are atomic, but not isolated, since inconsistencies can occur. Conflicts between transactions do not occur since they cannot write to the same shared memory.
Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) \cite{2,3} is a method to implement STM in a consistent and efficient way. In MVCC, writes to shared memory do not overwrite data, instead a new version of the data is created while the old version is kept intact. This allows reads to data to always be consistent, albeit outdated.

The index borrows the idea for versioning in the form of revision numbers. Revision numbers are not used to provide consistency, but only to detect inconsistencies. It is not required to provide consistent data, detecting inconsistent (old) data and restarting the analysis suffices. This avoids keeping old data, which can be complex to manage and introduces space and computation overhead.

Concurrent Programming with Revisions and Isolation Types

In \cite{7}, an approach is described for simplifying parallel execution with shared data. Programmers can declare which data to share and how data is merged in case of conflicts using isolation types. Tasks are executed concurrently by forking and joining revisions. Revisions are isolated, they only read from and write to a private copy of the shared data.

This approach is similar to the approach of the index infrastructure. Instead of having to declare which data to share, all data is shared implicitly. Since no merge conflicts can occur because of the partition restriction, no explicit merging strategy is required. Forking and joining revision is identical to starting and committing transactions.

A key difference is that revisions are isolated and transactions are not. Since inconsistencies can occur when other transactions are committed, transactions are not completely isolated. We chose this because in our case, consistency is not a requirement. However, because their approach is more general, they do require isolation. They provide isolation by performing reads and writes on a private copy of the shared data. The data is not copied for every fork, because this would be very expensive. Instead they only copy lazily when a write to shared data would cause an inconsistency in another revision.

Another difference is that they allow revisions to be nested, with a restriction that nested revisions cannot be joined before the parent is joined. In the index we do not allow this, since it is not required.
Chapter 6

Conclusion

This thesis concludes with a summary in Section 6.1 and conclusions to the research questions in Section 6.2.

6.1 Summary

6.1.1 Declarative Specification of Name Binding and Scope Rules

Chapter 2 introduced the Name Binding Language (NBL), a declarative DSL in the domain of name binding. Instead of implementing name resolution, language engineers specify name binding rules in NBL. Specifications consist of concepts from the name binding domain; definition, namespace, reference, scopes and import rules.

Definitions are named elements in a program that define a concept from the language. Definitions can be unique, for example a C# field or method. Or they can be non-unique, for example C# partial classes, or Stratego strategies. Namespaces define a global storage for names. Different namespaces are used to distinguish between different groups of names. For example, field names are stored in the field namespace, method names in the method namespace, classes and partial classes in the class namespace, and so on.

References refer to definitions by name. Examples are variable or field references, method calls or class instantiations. Namespaces in the context of references specify what kind of definition this reference refers to.

Scopes restrict the visibility of definition sites. The main goal of scopes is to allow reuse of definition names in separate scopes, and to provide errors by disallowing referencing a definition outside of its scope. Scopes can be named, for example a class (which has a name) scoping all methods and fields inside that class. They can also be unnamed, for example a statement block (which is unnamed) scoping all local variables. Namespaces in this context specify which namespaces are scoped, by default no namespaces are scoped.

Imports introduce definitions from another scope into the current scope. Examples of imports are Java package imports or C# namespace imports, which import all classes from the package or namespace scope, into the current file. Class inheritance is also specified using imports. All fields and methods from the parent class, including any transitively imported fields and methods, are introduced in the base class.
6. CONCLUSION

NBL is evaluated against a number of case studies including subsets of programming languages such as C and C#, and DSLs such as the Entity language, Green-Marl and Mobil. We found that NBL is well suited for name binding specifications for DSLs. However, it is less suited for specifying entire programming languages, as these have much more complex name resolution requirements. For C# for example, method overloading with subtyping, dynamic typing and the this keyword could not be expressed in NBL, but ideas for supporting these language constructs are given.

Name resolution and editor services were traditionally created manually for each language, requiring a large implementation effort. NBL reduces the implementation effort for creating languages (in Spoofax) because generic name resolution and editor services based on NBL are provided. Furthermore, understanding name binding definitions for languages using NBL is improved over programmatic encodings of name binding.

6.1.2 Persistent Infrastructure for Incremental and Parallel Name Resolution

Chapter 3 described the requirements, design, implementation and evaluation of the index infrastructure. The previous infrastructure for persistent data in Stratego, dynamic rules, did not meet the requirements for incremental and parallel name resolution using NBL. Dynamic rules cannot be used in a parallel environment, since there is no concurrency control for dynamic rules. It is not easy to map one key to multiple values, which is a problem for non-unique definitions or detecting duplicate definitions for unique definitions in NBL. It is also not easy to do efficient incremental name resolution using dynamic rules.

A new infrastructure is required that has a symbol table with fast addition, retrieval and remove operations. The symbol table must be usable in incremental and parallel name resolution. The symbol table, called the index, consists of immutable entries that have a key and a value. Entries are partitioned into partitions, which can be files, but also more fine-grained units such as a class or method. An API is defined for setup, retrieval, addition, removal and persistence operations on the index.

Transactions are used as concurrency control. Modifications are collected in transactions and applied in bulk by committing the transaction. Transactions are restricted so that multiple transactions can be performed in parallel in a thread-safe way and to ensure that no merge conflicts occur. Inconsistencies can occur however, but a protocol is defined that handles inconsistencies and guarantees that access to the symbol table is thread safe. Incrementing revision numbers are assigned to transactions and partitions, retrieving data from a partition with a higher revision number signals that the data may be inconsistent. If data is inconsistent, the transaction has to be restarted. Using this protocol guarantees that the analysis is consistent and thread-safe, allowing parallel analysis.

A second protocol is defined that handles dependencies. Dependencies are traced by an extension of the API. Any data that is retrieved from the index leaves a trace in the index. The traces are later used to trace back dependencies from changes in data. Changes in data are addition or removal of definitions, or changes in data associated with definitions, such as types. If this protocol and API extension is used, it guarantees
that changes in data cause a reanalysis in all dependencies, resulting in an incremental analysis.

The functionality of the index is tested using unit tests. These tests were very useful during development as regression tests and will continue to be useful for future development. Since the non-functional requirement of high performance is also important, performance is also tested. The performance sensitive operations on the index are benchmarked. Benchmarks show that the initial implementation has a performance problem with removal operations. The second implementation does not have this performance problem and is overall a lot faster. The benchmarks also show that the persist operation is very slow for large indexes, since the entire index is serialized to disk. However, this did not pose a problem yet since persisting the index is not performed very often and is executed in the background. The performance benchmarks are set up in the same way as the unit tests, so they can be used for performance regression testing. Overall, the index infrastructure is a success because all requirements are fulfilled and performance is high, with the exception of the persist operation.

6.1.3 Language-Parametric Name Resolution

In Chapter 4 the language-parametric incremental and parallel name resolution algorithm was described. Name resolution algorithms are commonly written by hand for each language. Such algorithms require a large implementation effort, are hard to understand and cannot be reused. This is caused by a lack of abstraction of the name binding domain. NBL, the declarative DSL in the domain of name binding, provides these lacking abstractions. Because of these abstractions, a language-parametric (generic) name resolution algorithm can be constructed. A language-parametric algorithm based on abstractions from NBL can be re-used between any language that specifies name binding using NBL. The language-parametric name resolution algorithm can be incrementalized and executed in parallel by using the index, operations and protocols from the infrastructure. Any language with a NBL specification will then get an incremental and parallel name resolution, without the large implementation effort.

The main functional requirement of the algorithm is that it should be an implementation of NBL, following the semantics from Chapter 2. It should be reusable and support editor services from Spoofax such as reference resolution, code completion and error checking. The main non-functional requirement is that the algorithm should perform incremental name resolution and should work in a parallel environment.

The algorithm is a four-stage algorithm consisting of a definition, data, resolve and dependency stage. It takes as input an AST and NBL specification, and produces an annotated AST with name binding information, and persisted name binding information in the index. The definition stage is concerned with assigning a key to each definition such that references can be resolved to definitions in the resolve stage. This stage implements most of the logic regarding definitions and scopes from NBL. The data stage gathers additional information about definitions such as types and also stored data for imports.

In the resolve stage, each reference is resolved to their definition using a lookup algorithm. The resolve stage implements the logic of references and imports. Finally, in the dependency stage, any changes found during the algorithm are traced to dependent partitions. Any partition that depends on an index entry that was added, removed
or changed is reanalysed according to the dependency protocol from Chapter 3. This results in an incremental name resolution algorithm. Using the other protocol, the algorithm is protected against inconsistencies and made usable in a parallel environment. An implementation for parallel name resolution is also given.

The functionality and performance of the algorithm is evaluated by applying it to a number of case studies. Unit tests for case study languages ensured that changes in the algorithm resulted in the same behaviour. Benchmarks are used to evaluate the performance. Several configurations of test files were generated and used to measure the performance of the algorithm under certain conditions. The number of files, definitions, references and dependencies are varied.

The conclusion from these benchmarks is that the number of definitions does not have a large influence on the performance. However, references have a larger influence since the lookup algorithm from the resolve stage has a higher complexity than the definition stage. The number of files also has, as expected, a large influence on the performance. Also, the initial import of a project is not as fast as expected, for a 250 file project, an analysis of 74 seconds is required. But after the initial import, the algorithm is incremental, so the initial import is not a big problem.

The performance of parallel execution in contrast to sequential execution is also benchmarked. Parallel execution on average was twice as fast. In tests with an increased number of references there is a higher speedup from parallel execution. This has to do with the length of a single analysis. Longer jobs tend to gain more speedup than smaller jobs. This is caused by how the parallel execution works, and the library used to do parallel scheduling. Scheduling is not optimal and causes threads to wait and do nothing, which happens less often with longer jobs. The fact that parsing cannot be done in parallel also reduces the performance gains of parallel execution.

Overall, the raw performance of the algorithm is satisfactory. The initial analysis of a project when it is opened the first time may take a while, but after that it is incremental. There is room for improvement of the raw performance, especially in the resolve stage since this has the largest impact on performance. The speed improvement from parallel name resolution is also satisfactory. It shows that name resolution and name-dependent analyses such as error checking can be performed in parallel. The speed gains from parallel execution can be increased even more by providing a better parallel job scheduler and parallel parsing.

6.1.4 Integration

Previously, name resolution for a new language in Spoofax was either written from scratch, or using a library with few abstractions. By providing NBL, the index and a generic name resolution algorithm ‘for free’, the implementation effort of constructing a new language in Spoofax is reduced immensely. This is of course only true if NBL and the name resolution algorithm cover the language that is to be constructed. If only a small part of coverage is missing, reference lookup adjust rules, desugaring or other small transformations can be used to fill the gap. If NBL conceptually does not support a certain feature in a language, NBL and the name resolution algorithm will have to be changed to support the feature. However, in many cases, especially for DSLs, there will be full coverage. In most cases, writing an adjust rule or desugaring will be sufficient to cover a certain language feature.
The effort of maintaining and extending a language is also reduced. Changing the NBL specification for a language takes a lot less time than changing a custom name resolution algorithm. This is mainly because changing programmatic encodings is a lot harder than changing declarative specifications. The abstractions from NBL also help with understanding name resolution for a language. If someone is tasked with changing name resolution for a language, we think they would rather change a NBL definition than a name resolution algorithm. Where NBL has a small set of clear abstractions, a custom name resolution algorithm has no name binding abstractions, with binding rules being encoded in the algorithm. This encoding entails overhead in understanding the name resolution of a language, making it harder to maintain or extend.

6.2 Conclusions

**Research Question A**: What are the fundamental concepts of name binding?

The Name Binding Language introduced in Chapter 2 provides abstractions for the fundamental concepts of the name binding domain: namespaces, definitions, references, scopes and imports. We think that these abstractions are minimal if adjust rules are not taken into account (pure NBL without the algorithm). In other words, it is not possible to express a concept in terms of another. We also think the abstractions are clear and understandable. Combining these facts, NBL definitions are relatively easy to understand compared to other related approaches in Chapter 5.

The abstractions from NBL cover a wide range of features of (programming) languages, but not all. It is probable that we have not found all fundamental concepts yet, or that a concept could be generalized to cover more features. During the evaluation of NBL we found several language constructs that could not be expressed in NBL. Most of these language constructs could be expressed by extending or generalizing concepts. For example, `in surrounding scope` is valid for definitions (it is the default definition site), but not for references. Both the ‘this’ keyword and by name arguments issues could be solved by generalizing `in surrounding scope` to be valid for references as well.

A missing fundamental concept may be advanced interactions with the type system, or typing rules. It is still unclear which (or whether) typing rules should be introduced in NBL, or if they should go into a separate type system language. The method overloading and subtyping issue hinted that some type rules, such as `has type` are a good candidate for addition in NBL. Forcing those rules into a separate language would entail code duplication, since some `has type` rules depend on name binding. This is especially true for references, where the type system interacts with name binding and vice versa. For definitions, typing rules typically do not depend on name binding. Other type rules such as supertype relations also interact with name binding, but are different from the name binding rules of a language.
Research Question B: What are the formal semantics of the fundamental concepts of name binding?

In Chapter 4, we presented a language-parametric name resolution algorithm based on the abstractions from NBL. The algorithm represents the semantics behind the fundamental concepts from the previous research question. Namespaces are identifiers that represent a global data store for one kind of language construct, such as classes, methods, or fields. Definitions are unique or non-unique named elements in a program that define a construct from the language under a certain namespace. References refer to definitions by namespace and name. Scopes restrict the visibility of definition sites in certain namespaces. Imports introduce definitions from another scope into the current scope, under the same names or new names.

Part of the algorithm is encoded in Aster, an attribute grammar language. Aster is more than just an attribute grammar, but the features we used to construct the algorithm are from pure attribute grammars, or can be easily rewritten to pure attribute grammars. Since pure attribute grammars can be seen as a formal semantics, part of the algorithm is already a formal semantics for the fundamental concepts. The resolution and dependency part of the algorithm is written in Stratego instead of Aster to interface with the index infrastructure. The index infrastructure is used to make the algorithm incremental and parallel. However, this part of the algorithm can be encoded in an attribute grammar as well. Instead of storing definitions in the index, all definitions can be collected in an attribute on which lookup can be performed. For dependency analysis, lookups would produce trace attributes which can be collected as well. This would result in a full Aster specification of the algorithm, but the performance and scalability will suffer because the index cannot be used. So a full Aster specification will not be useful for production purposes, but it can serve as a formal semantics for the fundamental concepts.

Research Question C: How can name resolution and name dependent analyses be executed incrementally and in parallel?

In Chapter 3, we presented the index infrastructure, an infrastructure that supports incremental and parallel analyses. The infrastructure contains a persistent symbol table that is designed to support safe parallel access using transactions. Incrementality is supported by using a library that interacts with the index.

To ensure proper usage of transactions and the incremental library, two protocols are defined. If the protocols are followed, an analysis becomes incremental and executable in parallel. The protocol for incremental analysis tracks dependencies by storing traces if information is retrieved from the index. Traces are traced back to find which partitions (files) are affected by a change, which in turn are reanalyzed. The protocol for parallel analysis ensures safe concurrent access of shared data by proper use of threads and transactions.

Using the persistent symbol table and protocols, name resolution and name dependent analyses can be executed incrementally and in parallel.
Research Question D: How can incremental and parallel execution for name resolution and name dependent analyses be automatically derived using name binding definitions?

In Chapter 3 we defined two protocols that, if followed, make an analysis incremental and executable in parallel. The generic name resolution from Chapter 4 adheres to these protocols and thus is incremental and executable in parallel. In addition, if analyses created by the language engineer such as type analysis and error checking also adhere to these protocols, they too become incremental and parallel. Since these analyses can be performed directly after name resolution, the only conditions are that these analyses use the index operations so that dependencies are tracked and inconsistencies are prevented. Because these analyses depend on information from name resolution, they need to use the index operations anyway, since that is the only way to access this information. So while incremental and parallel execution for name resolution and name dependent analyses is not entirely automatically derived, the effort required is very low.
Chapter 7

Future work

In this chapter we show interesting future work regarding NBL, the index infrastructure and the generic name resolution algorithm.

Improved NBL Coverage

In the evaluation of NBL (Section 2.4) we discovered several language constructs that could not be expressed in NBL, and explored possible ways to express these language constructs in NBL. NBL and the name resolution algorithm should be extended to support these language constructs. It would also be interesting to try to model full name binding of popular programming languages such as C# and Java to improve the coverage of NBL.

Improved Index Persistence

Currently the entire index is persisted to disk in one operation. In the evaluation of the index infrastructure (Section 3.6) we measured that persisting the entire index to disk could take several seconds for high numbers of entries. A better approach would be to serialize partitions separately. This can be done incrementally during or after semantic analysis. It can also be used to reduce the memory footprint by unloading unused partitions from memory and load them from disk when needed. Some extra latency could be incurred, but if an index becomes too big to fit in memory this has to be done.

Using a Database as Index

It is worthwhile to do further investigation into using an existing database engine as the index implementation, such as HyperSQL or H2. The (possible) benefits of using an existing database engine are:

- Fine grained persistence ‘out of the box’.
- Most database have some sort of concurrency control ‘out of the box’.
- Easy to change data model and keys without changing code.
7. Future Work

- Possible to use a client-server (possibly distributed) setup.

Some disadvantages are:
- Marshalling overhead for internal representations of database.
- Latency when using client-server model.
- Most databases use SQL. SQL is more suited for large operations on large datasets, not simple hash lookups.
- Less control over the implementation and how concurrency is handled.

Improved Parallel Scheduling

During the evaluation of the name resolution algorithm (Section 4.6) we discovered that scheduling parallel jobs using the Spoofax-parallel library was not very efficient. The available CPU cores are not constantly in use. A fixed number of jobs get spread over a number of threads, but all jobs must be finished before continuing. Parsing is also done sequentially, since using the parser in parallel is not supported in Spoofax. To efficiently fill up all (or most) CPU cores with work, a parallel job queue should be added to Spoofax. Using a parallel job queue, jobs can be added on the fly without waiting.

Incremental and Parallel Parsing

The parser in Spoofax does not do incremental parsing, the entire source file is parsed for every change. This in turn triggers an analysis for the entire file, because it is unknown which part of the file has changed. With incremental parsing it is known which part has changed and a more fine-grained analysis can be done. Analysis could be scoped to top-level units such as classes or methods. For example, a change in a method body would only trigger analysis for that method body. Changing the method name would trigger analysis for the class that contains the method. This will increase performance for large files with a lot of top-level units.

Furthermore, the parser in Spoofax cannot be used in parallel at this moment. It is most likely a small change that would allow this, since the parser only reads from one global parse table, but the parse table does not change.

Cross-Language References in NBL and Index Infrastructure

Currently, code generation is often (ab)used to make definitions in a language available in another language. For example, constructors from SDF are transformed to a Stratego file that contains signatures for use in Stratego. This introduces unnecessary overhead; a new file is generated, pretty printed, stored on disk and then analyzed (possible multiple times). Instead, NBL and the index infrastructure should support references to other languages. For example, to resolve to either Stratego signatures or SDF constructors from Stratego, the following syntax in NBL could be used:
The index infrastructure would also need to be changed to support easier use of entries from other languages. However, it is not easy to do in Spoofax yet, as there is no abstraction to register a dependency of a language on another.

**Incremental and Parallel Compilation**

Incremental compilation is in fact already possible. In the same way that editor services such as error checking are already incremental and parallel, compilation can be made incremental and parallel as well. However, analysis and compilation should be separated; dependencies during compilation do not have to be dependencies during analysis. For example, some aspects are only active during compile time, such as the `assigned` aspect in the Entity language. The `assigned` aspects changes the code generation to add a statement before or after an assignment statement. Dependencies from those aspects are only active during compile time and should be ignored during analysis. A prototype for doing this already exists, but needs to be generalized so that it can be used for every language.

Another problem is the compilation of partial definitions such as C# partial classes or Stratego strategies. Partial classes and strategies are merged in a desugar step before the actual compilation. This makes separate compilation of partial definitions harder, since these definitions always have dependencies on other definitions and can never be compiled separately. In that case either all partial definitions of the same name need to be compiled if any partial definition changes, or the results of compiling a partial definition needs to be cached. Compiling all partial definitions for every change requires more computation, caching the result requires more memory. Since caching the compilation results entails storing ASTs, a lot of memory may be required.

**Generating Command-line Analyses and Compilers**

Spoofax generates an IDE with editor services based on the semantic analysis specified using NBL and Stratego. However, command-line analyses and compilers have to be created by hand. Spoofax should also generate a command-line version of the analyses and compilation. A command-line version is required for integration in build scripts and build farms. It is also very useful for experimenting with a language in a terminal by creating a REPL. NBL, the index infrastructure and the name resolution algorithm are separate from Eclipse, so they can be re-used in command-line versions. Stratego code can compile to Java classes, so everything is in place to generate command-line versions.
Declarative Specifications for other Domains

The same principles of declarative specifications that were used to create SDF and NBL could be applied to other domains in language engineering as well. For example, a type language that specifies the type system for a language instead of implementing a type system in Stratego. It will be interesting to see if such abstractions can be made for each domain in language engineering, and what the impact on implementation effort is.

There is also the problem of balancing the concepts and managing the interactions between these DSLs. The domain of name binding and type systems are dependent on each other. We found several cases of this in the evaluation of NBL (Section 2.4). Resolving to a field in a class in name binding requires the type system, evaluating the type of a method call requires name binding to resolve the method. This causes NBL to contain some interactions with the type system and the type system language would have some interaction with name binding. An option would be to just add type rules to NBL. We could add all type rules to NBL, but that would introduce the entire type system domain into NBL. On the other hand, not adding any type rules to NBL and adding them only to type system language may cause duplicate code between NBL and the type system language. Which (or whether) typing rules should be added to NBL, and how a type system language should look like is another topic for further research.


BIBLIOGRAPHY


Appendix A

Extended Entity language specification
A. Extended Entity Language Specification

Figure A.1: Syntax definition in SDF for the extended Entity language. The names in annotations are term constructors.
namespaces
Module Aspect Pointcut Advice Entity Property Function Variable

rules // Module
Module(m, _, _) :
  defines unique Module m
scopes Entity, Aspect

WildcardImport(m) :
  imports Entity from Module m

EntityImport(m, e) :
  imports Entity e from Module m

rules // Aspect
Aspect(a, _) :
  defines unique Aspect a
scopes Pointcut, Advice

Pointcut(p, _, _) :
  defines unique Pointcut p
scopes Variable

Advice(a, _, _, p, _, _) :
  defines unique Advice a
  refers to Pointcut p
scopes Variable

rules // Entity
Entity(e, _) :
  defines unique Entity e of type Type(e)
scopes Property, Function

Type(e) :
  refers to Entity e
otherwise refers to Entity "Int"
otherwise refers to Entity "String"

rules // Property
Property(p, t) :
  defines unique Property p of type t

PropAccess(e, p) :
  refers to Property p in Entity t
  where e has type Type(t)

rules // Function
Function(f, _, _) :
  defines unique Function f
scopes Variable

Call(f, _) :
  refers to Function f

rules // Variable
VarDecl(v, t) :
  defines unique Variable v of type t in subsequent scope

VarDeclInit(v, t, _) :
  defines unique Variable v of type t in subsequent scope

Param(p, t) :
  defines unique Variable p of type t in subsequent scope

Var(v) :
  refers to Variable v
otherwise refers to Property v

rules // Control flow
Block(_) :
  scopes Variable

Figure A.2: Name binding and scoping definition in NBL for the extended Entity language.
Appendix B

MiniC# specification
B. MINIC# specification

Figure B.1: Syntax definition in SDF for MiniC#. The names in annotations are term constructors.
namespaces
Namespace Class Method Field Variable

rules // Namespaces
Namespace(x, _) :
defines non-unique Namespace x
scopes Namespace, Class

NsOrType(n1, n2):
refers to Namespace n2 in ns
where n1 refers to Namespace ns
otherwise refers to Class n2 in ns
where n1 refers to Namespace ns

Using(qname):
imports Class from Namespace ns
where qname refers to Namespace ns

Alias(alias, qname):
imports Namespace ns as alias
where qname refers to Namespace ns
otherwise imports Class c as alias
where qname refers to Class c

rules // Classes
Class(NonPartial(), c, _, _) :
defines unique Class c of type ClassType(c)
scopes Field, Method

Class(Partial(), c, _, _) :
defines non-unique Class c of type ClassType(c)
scopes Field, Method

Base(c):
imports Field {transitive}, Method {transitive}
from Class c

ClassType(c) :
refers to Class c

rules // Fields
FieldDef(t, x) :
defines unique Field x of type t

FieldAccess(e, f) :
refers to Field f in Class c
where e has type ClassType(c)

rules // Methods
Method(t, m, p*, _) :
defines unique Method x of type (t*, t)
scopes Variable
where p* has type t*

Call(e, m, a*) :
refers to Method m of type (t*, _,) in Class c
where e has type Type(c)
where a* has type t*

Call(m, a*) :
refers to Method m of type (t*, _,) where a* has type t*

Param(t, p) :
defines unique Variable p of type t

rules // Variables
Var(t, v, _) :
defines unique Variable v of type t in subsequent scope

Var(t, v) :
defines unique Variable v of type t in subsequent scope

VarRef(v) :
refers to Variable v
otherwise refers to Field v

rules // Control flow
For(t, v, _, cond, stmt, body) :
defines unique Variable v of type t in cond, stmt, body

Foreach(t, v, _, body) :
defines unique Variable v of type t in body

Block(_)
scopes Variable

Figure B.2: Name binding and scoping definition in NBL for MiniC#.
Appendix C

C-like language specification
C. C-LIKE LANGUAGE SPECIFICATION

Figure C.1: Syntax definition in SDF for the extended C-like language. The names in annotations are term constructors.
namespaces
File Struct Type Field Function Variable

rules // Compilation unit
(f, CompilationUnit(_, _)) :
defines unique File f
scopes Struct, Type, Function

Include(f) :
  imports Struct, Type, Function
  from File f

rules // Struct
StructDecl(s, _) :
defines unique Struct s of type Type(s) in subsequent scope
scopes Field
TypeDef(_, t) :
defines unique Type t in subsequent scope
Type(t) :
  refers to Type t
  otherwise refers to Type "char"
  otherwise refers to Type "int"
  otherwise refers to Type "void"
StructRef(s) :
  refers to Struct s

rules // Variable
Field(t, f) :
defines unique Field f of type t
VarDecl(t, v) :
defines unique Variable v of type t in subsequent scope
Param(t, p) :
defines unique Variable p of type t
VarRef(v) :
  refers to Variable v
Access(e, f) :
  refers to Field f in Struct s
  where e has type Type(s)

rules // Function
FunDecl(t, f, _, _) :
defines unique Function f of type t in subsequent scope
scopes Variable
Call(f, _) :
  refers to Function f

Figure C.2: Name binding and scoping definition in NBL for the C-like language.
Appendix D

Green-Marl specification
namespaces File Type Procedure Function Variable Iterator Property

rules // Compilation unit
(f, CompilationUnit(_)):
defines File f
scopes Procedure, Function

rules // Function
Proc(ProcHeader(p, _, t), _):
defines Procedure p of type t
scopes Variable, Iterator, Property

Proc(FuncHeader(f, _, t), _):
defines Function f of type t
scopes Variable, Iterator, Property

Call(e, f, _):
refers to Function f in Type t
where e has type t

Call(f, _):
refers to Function f

rules // Variable
Param(p, t):
defines Variable p of type t

PropParam(p, t):
defines Property p of type t

VarDef(t, v):
defines Variable v of type t in subsequent scope

VarDefAssign(t, v, _):
defines Variable v of type t in subsequent scope

PropDef(t, p):
defines Property p of type t in subsequent scope

Iter(i, n, r):
defines Iterator i of type ElemType(n, r) in subsequent scope

RevIter(i, n, r):
defines Iterator i of type RevElemType(n, r) in subsequent scope

VarRef(v):
refers to Variable v
refers to Iterator v
refers to Property v

IterRef(i):
refers to Iterator i

PropRef(e, p):
refers to Property p in Type t
where e has type t

rules // Control flow
Block(_):
scopes Variable, Property, Iterator

Figure D.1: Name binding and scoping definition in NBL for Green-Marl.
Appendix E

Mobl specification
namespaces Application Module Type Screen Service Resource Style
   Control Method Property Variable

rules // Top
   Application[a, _] :
      defines Application a
      scopes Screen, Service, Style, Control, Type, Method, Variable
   Module[m, _] :
      defines Module m
      scopes Screen, Service, Style, Control, Type, Method, Variable
   Import(m) :
      imports Screen {current-file}, Service {current-file},
         Style {current-file}, Control {current-file},
         Type {current-file}, Method {current-file},
         Variable {current-file}
      from Module m

rules // Type
   Type(_, t, _, _) :
      defines Type t of type SimpleType(t)
      scopes Method, Property
   TypeNoSuper(_, t, _): 
      defines Type t of type SimpleType(t)
      scopes Method, Property
   ExternalType(_, t, _, _) :
      defines Type t of type SimpleType(t)
      scopes Method, Property
   ExternalTypeNoSuper(_, t, _): 
      defines Type t of type SimpleType(t)
      scopes Method, Property
   ExternalGenericType(_, t, _, _) :
      defines Type t of type SimpleType(t)
      scopes Method, Property
   Entity(_, t, _, _) :
      defines Type t of type SimpleType(t)
      scopes Method, Property
   EntityNoSuper(_, t, _): 
      defines Type t of type SimpleType(t)
      scopes Method, Property
   SimpleType(t) :
      refers to Type t
      refers to Type "Object"
      refers to Type "JSON"
      refers to Type "Dynamic"

rules // Screen
   ScreenNoReturnType(_, s, _, _) :
      defines Screen s
      scopes Variable

rules // Service
   Service(_, s, _, _) :
      defines Service s of type SimpleType(s)
      scopes Resource
   Resource(_, r, _, t, _): 
      defines Resource r of type t
      scopes Variable

Figure E.1: Name binding and scoping definition in NBL for Mobl, part 1.
rules // Style
Style(_, [StyleSelector(s)], _) :
defines Style s
scopes Variable

Control(_, c, _, _) :
defines Control c
scopes Variable

ExternalControl(_, c, _) :
defines Control c
scopes Variable

rules // Method
Function(_, f, _, t, _) :
defines Method f
scopes Variable

ExternalFunction(_, f, _, _) :
defines Method f
scopes Variable

ExternalSyncFunction(_, f, _, _) :
defines Method f
scopes Variable

ExternalSyncMethod(_, m, _, _) :
defines Method m
scopes Variable

ExternalMethod(_, m, _, _) :
defines Method m
scopes Variable

ExternalStaticSyncMethod(_, m, _, _) :
defines Method m
scopes Variable

ExternalStaticMethod(_, m, _, _) :
defines Method m
scopes Variable

MethodCall(e, m, _) :
refers to Method m in Type t
refers to Resource m in Service t
where e has type SimpleType(t)

Call(_, _) :
refers to Method m
refers to Screen m
refers to Type m

rules // Property
Property(_, p, t, _) :
defines Property p of type t

PropertyNoAnno(_, p, t) :
defines Property p of type t

FieldAccess(e, p) :
refers to Property p in Type t
where e has type SimpleType(t)

Figure E.2: Name binding and scoping definition in NBL for Mobl, part 2.
E. MOBL specification

rules // Variable
StyleVarDecl(StyleVar(v), _) : 
  defines Variable v
VarDecl(v, t, _) : 
  defines Variable v of type t
VarDeclInferred(v, _) : 
  defines Variable v
FArg(p, t) : 
  defines Variable p of type t
FArgOptional(p, t, _) : 
  defines Variable p of type t
ListInferred(l, _, _) : 
  defines Variable l
List(l, t, _, _) : 
  defines Variable l of type t
Var(v) :
  refers to Variable v
  refers to Method v
  refers to Type v
  refers to Service v
  refers to Style v
  refers to Control v
  refers to Type "Object"
  refers to Type "JSON"
  refers to Type "Dynamic"
  refers to Control "elements"
  refers to Variable "event"

rules // Control flow
ControlCallNoArgs(_, _) : 
  scopes Variable
ControlCallNoBody(_, _) : 
  scopes Variable
ControlCall(_, _, _) : 
  scopes Variable

Figure E.3: Name binding and scoping definition in NBL for Mobl, part 3.