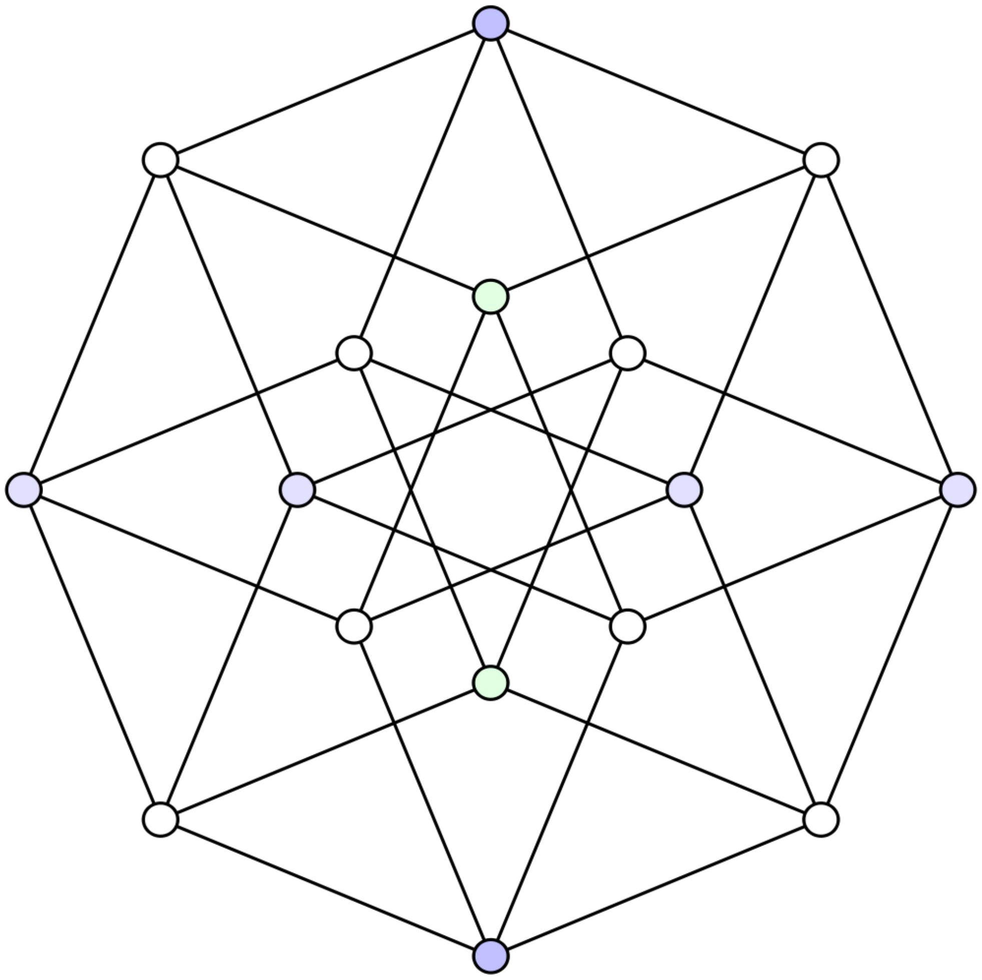
Composable Type System Specification using Heterogeneous Scope Graphs

Master's Thesis



Aron Zwaan

Composable Type System Specification using Heterogeneous Scope Graphs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Aron Zwaan
born in Bennekom, the Netherlands

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

Composable Type System Specification using Heterogeneous Scope Graphs

Author: Aron Zwaan
Student id: 4451031
Email: `a.s.zwaan@student.tudelft.nl`

Abstract

Static Analysis is of indispensable value for the robustness of software systems and the efficiency of developers. Moreover, many modern-day software systems are composed of interacting subsystems written in different programming languages. However, in most cases no static validation of these interactions is applied.

In this thesis, we identify the *Cross-Language Static Semantics Problem*, which is defined as "How to provide a formal and executable specification of the static semantics of interactions between parts of a software system written in different languages?" We investigate current solutions to this problem, and propose criteria to which an all-encompassing solution to this problem must adhere.

After that, we present a design pattern for the Statix meta-DSL for static semantics specification that allows to model loosely coupled, composable type system specifications. This pattern entails that the semantic concepts of a particular domain are encoded in an interface specification library, which is integrated in the type system of concrete languages. This allows controlled but automated composition of type systems. We show that, under some well-formedness criteria, the system provides correct results.

A runtime, executing composed specifications, is implemented using PIE pipelines for partial incrementality, and integrated in the command-line interface and Eclipse IDE platforms, using the Spoofax 3 Framework. This allows using multi-language analysis in concrete projects.

The design pattern, and the accompanying runtime are validated using two case studies. These case studies show that the approach is effective, even in a case where there is an impedance mismatch in the data models of the involved languages.

Thesis Committee:

Chair, Supervisor:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Finavaro Aniche, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. P.D. Mosses, Dept. of Computer Science, Swansea University
Committee Member:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft

Preface

My history with Computer Science probably started when I watched my dad working from home as a young boy. Despite I did not grasp what he was doing at all (his explanation let me think he was a cryptographer), I was highly fascinated. This fascination revived in the second half of my secondary school period, and culminated just before final exam. Then I wrote my first 'compiler', which translated plain text summaries to TI BASIC programs on my graphical calculator. The success made the school forbid graphical calculators for physics and chemistry the year after...

Now, more than five and a half years later, I am approaching the end of my studies. In this period, a world opened up. Expressing my thankfulness to everyone that played a role in that process is unattainable, hence I restrict to prominent people. First, I feel indebted to Eelco Visser, who guided me through this thesis project. Your critical but equally supportive feedback, the discussions that followed, and the pointers to relevant research helped me to improve the project result vastly. Next, I want to thank Gabriël Konat and Hendrik van Antwerpen for reflecting on one of the writings at the start of my thesis project, proofreading parts of this report, and navigating me into the depths of development with PIE/Spoofax 3 and Statix. I also want to thank Maarten Sijm, Ivo Wilms and Peter Mosses for providing their helpful feedback and Bert Roos for his insightful viewpoints.

An indirect, but equally important contribution to this result is companionship and support. In that respect, I am deeply grateful to my parents, for raising me the way you did. Compared to the love and effort you invested in me, this thesis was a sinecure. The same holds for my brothers. I am thankful for the strong bond we have. I also want to thank family Vlot, whom I lived with during my studies, for your openness, patience and support. Living so close with you has shaped me in numerous ways. Further, I am indebted to Joost and Diederik Geuze, for the broad experience I could gain at VitalHealth. In addition, my study years have been enriched intensely by the friends I met. Thank you for all the deep, late-night (or early morning) conversations and activities during our holidays and weekends. Life would be barren without you. Finally, I am deeply and sincerely grateful to the Creator of heaven and earth, whom I believe has given me the strength and ability to do my work every day. From your fullness, I received grace upon grace (John 1:16).

Aron Zwaan
Terschuur, the Netherlands
January 20, 2021

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

1.1 Contributions 2

1.2 Thesis Outline 3

2 Multi-Language Programming Environment 5

2.1 Problem Statement 5

2.2 Goal 7

2.3 Alternative approaches 9

2.4 Solution Criteria 12

3 Case Studies Introduction 15

3.1 Mini-SDF and Mini-Stratego 15

3.2 Mod and Mini-SQL 23

4 Multi-Language Type System Definition in Statix 33

4.1 Introduction to Statix 33

4.2 Defining Composable Type Systems in Statix 40

4.3 Case studies 43

5 Implementation in Spoofax 3 57

5.1 Background: Architecture of the Statix Implementation 57

5.2 Background: Spoofax 3 Architecture 61

5.3 Implementation of Multi-Language Analysis 65

5.4 Conclusion 78

6 Evaluation 79

6.1 Correctness of the SCI Design Pattern 79

6.2 Composability 82

6.3 Revisiting The Multi-Language Programming Environment 85

6.4 Durability 85

6.5 Usability 85

6.6 Performance 86

6.7 Possible Improvements 88

6.8 Conclusion 90

7 Related Work **91**
7.1 Taxonomy 91
7.2 Attribute Grammars 92
7.3 Other Approaches 93

8 Conclusion **95**
8.1 Future Work 95

Bibliography **97**

A Statix Specification of Mini–SDF and Mini–STR Case Study **103**
A.1 Interface 103
A.2 Mini–SDF 109
A.3 Mini–STR 112

List of Figures

2.1 Example of Cross Language Consistency Validation 7
2.2 Current Programming Environment Organization 7
2.3 Envisioned Future Programming Environment 7
2.4 SQL String recognition in IntelliJ 11

3.1 Syntax definition for Mini-SDF 16
3.2 Syntax Example 16
3.3 Shadowing 17
3.4 Double Import 17
3.5 Double Sort name 17
3.6 Diamond import 17
3.7 Duplicate Constructor name 18
3.8 Constructor Overloading 18
3.9 Undeclared Sort 18
3.10 Forward Reference 18
3.11 Unresolved Import 18
3.12 Syntax definition for Mini-STR 19
3.13 Mini-STR Example 19
3.14 Matching a list 20
3.15 Building a list 20
3.16 Pass list by variable 21
3.17 Rewrite rules 21
3.18 Errors in rewrite rules 22
3.19 Mini-STR integration with Mini-SDF 23
3.20 Name resolution from Mini-STR to Mini-SDF 23
3.21 Error on invalid import 23
3.22 Partial Syntax definition for Mod 24
3.23 Example Mod program 25
3.24 Mod program with errors 25
3.25 Syntax definition for Mini-SQL 26
3.26 Mini-SQL program 27
3.27 Duplicate columns 27
3.28 Mini-SQL resolution errors 28
3.29 Mini-SQL Primary Key errors 29
3.30 Mod – Mini-SQL example 29
3.31 Record initialization errors 30
3.32 Record in Foreign Key field 30
3.33 Procedure invocation 31

3.34 Procedure without Parameters . . . 31
3.35 Procedure `Date` Parameter . . . 31
3.36 Mod – SQL: Procedure Foreign Key return . . . 31

4.1 Expression language grammar . . . 34
4.2 Expression signature . . . 34
4.3 Rule signature . . . 34
4.4 Integer typing . . . 34
4.5 Plus typing . . . 35
4.6 Less-than typing . . . 35
4.7 If-then-else typing . . . 35
4.8 Top and Bottom constraints . . . 35
4.9 Let and Var typing . . . 36
4.10 Scope Graph notation . . . 37
4.11 Reference resolution in scope graphs . . . 38
4.12 Relations and Labels . . . 38
4.13 Let typing rule . . . 38
4.14 Var typing . . . 38
4.15 Statix modules . . . 39
4.16 Project Type-checking . . . 40
4.17 Abstract signature interface (summarized) . . . 41
4.18 Interface usage (declaration) . . . 42
4.19 Interface usage (resolution) . . . 42
4.20 Examples of wrapper predicates . . . 42
4.21 Interface adaption . . . 43
4.22 Incorrect label usage . . . 43
4.23 Constructor resolution . . . 45
4.24 Module interface . . . 45
4.25 Module interface usage . . . 45
4.26 Import Example . . . 46
4.27 Abstract signature interface . . . 46
4.28 Mini-SDF interface usage . . . 47
4.29 Mini-STR interface usage . . . 47
4.30 Duplicate import checking . . . 48
4.31 Small Mini-SDF – Mini-STR project . . . 49
4.32 Scope graph of Figure 4.31 . . . 49
4.33 Relational Data Interface . . . 50
4.34 Relational Data Interface (2) . . . 50
4.35 Scopes representing types . . . 51
4.36 Nullability . . . 51
4.37 Stored Procedures . . . 51
4.38 Mod Types . . . 51
4.39 Type Translation . . . 53
4.42 Transformed Procedure Type . . . 53
4.40 Mod – Mini-SQL Integration example . . . 54
4.41 Scope Graph of Figure 4.40 . . . 54

5.1 Ordered rules . . . 58
5.2 Rules without order . . . 58
5.3 More overlap . . . 59
5.4 Solver API . . . 60
5.5 Example PIE pipeline for creating a Jar file from a Java project . . . 62

5.6 Spoofax 3 Core API . . . 63
5.7 Task Definitions in LanguageInstance . . . 63
5.8 Spoofax 3 Default Pipeline . . . 64
5.9 Analysis Metadata . . . 66
5.10 Specification definition . . . 67
5.11 Specification Management . . . 68
5.12 Transformation Task definitions . . . 69
5.13 Transformation Integration in LanguageMetadata . . . 69
5.14 Transformation Task definitions . . . 70
5.16 Configuration . . . 71
5.15 Configuration Task definitions . . . 72
5.17 Analysis Execution Task definitions . . . 74
5.18 Check Task definition . . . 75
5.19 Summary of Task Definitions . . . 75

6.1 Extension Permission not composable . . . 81
6.2 Overlap Detection not composable . . . 81
6.3 Precedence Interference . . . 82
6.4 Size of Benchmark Source sets . . . 86
6.5 Benchmark Results . . . 87

Chapter 1

Introduction

Static Analysis is automated analysis of a computer program that is performed without actually executing the program. Static Analysis comes in many forms, including, but not limited to, type-checking (Pierce 2002), data-flow analysis (Khedker, Sanyal, and Sathe 2009), automated program verification (P. Cousot and R. Cousot 1977), automated bug finding and program repair (Cadar, Dunbar, and Engler 2008; Long and Rinard 2015), metrics calculation (Fenton and Bieman 2014) and linting (García-Munoz, García-Valls, and Escribano-Barreno 2016). Static Analysis provides a programmer with feedback about the system he is developing in early stages of the development cycle. This enables navigation, understanding and improvement of the system under development quickly. This improves the productivity of a developer and the quality of the system he is developing. Therefore, static analysis is of indispensable value to software engineers.

A particular form of static analysis is type checking. Type checking validates that a program is well-formed. That is, the type checker validates that each operation (for example an addition, field access or method call) is applied to runtime values that are consistent with its definition. Although type systems differ in the guarantees they provide, they are generally used to prevent type errors or undefined behavior at runtime.

Many contemporary software systems are engineered using multiple languages (Mayer, Kirsch, and Le 2017). Often, the main logic of the system is written in a General Purpose Language, which delegates sub-tasks to sub-programs that are written in other languages. Those secondary languages are chosen because they fit the domain of the sub-task better. For example, many web-applications use a dedicated query language for data storage and retrieval. Other reasons to use secondary languages include efficiency and interoperability with the operating system or legacy software.

Using a software component written in another language is an operation, similar to a regular method call. However, the type-checker of a language is usually not capable of validating the well-formedness of such an operation, nor subsequent operations applied on the returned data. This is caused by the fact that the type systems of the languages are not designed and implemented with possibilities for interoperability.

As a result of this lacuna, systems with multiple languages are more difficult to understand and improve. Moreover, inconsistencies might not be noticed at compile-time, resulting in runtime errors, which are often not mapped back to the original source location, and hard to interpret. This hampers the stability and robustness of the system. A recent survey conducted by Mayer, Kirsch, and Le (2017) illustrates this problem. Out of a group of 139 professional software developers, with on average 8 years of experience, 92% has encountered errors in cross-language linking, while only 25% has tooling available that assists in detecting cross-language linking errors. As a result, ”many indicated avoidance of multi-language development, cross-language linking, or changing cross-language identifiers”. Therefore, the role of type systems should be extended to validate interactions in the system that cross language boundaries as well. The integrity checks a type system provides should not be confined to the source set of a particular languages within a project, but should provide comprehensive validation of the complete workspace.

In this thesis, we investigate the requirements on type system design and type checker implementations that enable reliable integration of type systems of multiple languages. We analyze several existing partial solutions to this problem, and list their key limitations. Based on the findings of that survey, we propose four criteria a generic system for multi-language type checking should adhere to. These criteria guarantee correctness, consistent behavior in different environments and feasibility.

Statix (Antwerpen, Poulsen, Rouvoet, and Visser 2018) is a meta-DSL for declarative type system specification. It allows to write typing rules in a style that is close to formal inference rules. For name resolution, a set of primitives based on the concept of scope graphs (Néron, Tolmach, Visser, and Wachsmuth 2015) is integrated in the language. Moreover, type systems expressed in Statix are executable, which means that they can be used directly to type-check concrete programs.

Based the criteria for type system composition, we propose a design pattern for Statix type system specifications, that enables implementation of composable type systems. We suggest to abstract the semantic concepts of a particular domain in a specification library, and use such libraries to model type systems of concrete languages. Type systems that share such a library can then be composed automatically. This enables flexible integration of the type systems of multiple languages.

The existing Statix runtime is not capable of creating composed specifications, and verifying their integrity. Neither does it allow to combine analysis results from sources of different languages. Hence, we implemented a runtime that addresses these concerns. This runtime accurately manages specification composition and execution in any composition setting.

Currently, Statix does not fully support composition of arbitrary specifications. During the evaluation, we analyzed the compositional properties of Statix in-depth, and assessed their impact on type systems designed using the proposed design. Based on this analysis, we propose adaptions to the language that improve its compositionality. Until then, the checks that are not performed at compilation time are integrated in the runtime, to ensure that executed specifications are reliable.

Finally, the proposed design pattern and the implementation of the runtime are validated with two case studies. These case studies show that the system allows modeling meaningful interactions between type systems, even when the underlying data models are different.

1.1 Contributions

In this work the following contributions are made:

- We formulate the '*Cross-Language Static Semantics Problem*', and investigate which key aspects still require solving.
- We propose a set of criteria that a generic solution to that problem should adhere to. Full compliance with these criteria will enable fully integrated analysis of systems written in multiple languages.
- We present two case studies illustrating how type systems of multiple languages can be integrated.
- We propose a design pattern for Statix specifications that allows modeling composable type systems.
- We present a runtime that executes composed type systems.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. In chapter 2, we investigate the opportunities and challenges of multi-language analysis. In chapter 3, we discuss two case studies that illustrate how type systems can be composed. After that, we introduce the Statix meta-language, and propose a design pattern that enables composable type system specification. This design pattern requires an adapted compilation process and execution runtime. We present an implementation of such a runtime in chapter 5. The resulting system is evaluated in chapter 6 and compared to other work in chapter 7. Finally, chapter 8 concludes this thesis.

Chapter 2

Multi-Language Programming Environment

In this chapter, we introduce the Cross-Language Static Semantics Problem, which deals with the concern how to define and implement static consistency checking over language boundaries. In the first section, we give some background information regarding type systems, their properties and their evolution. Additionally, we explain why it would be desirable that type systems validate consistency over language boundaries. In the next section, we sketch the hypothetical programming environment that would emerge when the introduced problem would be solved. Subsequently, in section 2.3, we discuss four approaches that have been taken to solve the problem. Those approaches are: (1) defining a monolithic language aggregating several sub-languages, (2) extending another type system, (3) using code generation and (4) implementing an editor plugin. By assessing these approaches, we find a number of issues that still need to be solved. In the last section, we propose four criteria that a solution to the Cross-Language Static Semantics Problem must adhere to, based on the observed issues.

2.1 Problem Statement

A type system is a part of a programming language specification that describes how to ”prove the absence of certain program behaviors by classifying phrases according to the kinds of values they compute” (Pierce 2002). It consists of rules that check the ‘consistency’ or ‘well-formedness’ of a program. Together, these rules ensure that programs that adhere to them have particular guarantees about their execution behavior. Examples of these guarantees include: ‘a method which is called always exists’, or ‘references always point to allocated memory’. Such guarantees rule out undesirable behavior, making programs more robust and programmers more focused on the actual problem they want to solve.

Further classifications of a type system can be made. In this paragraph, we consider two characteristics that are particularly relevant for this thesis. Firstly, the validation described by the type system can be performed at compilation time or at execution time. The former approach, denoted as ‘statically typed’, prevents execution of a malformed program altogether, forcing the programmer to solve the type errors before executing it. This guarantees that actually compiled programs are well-formed with respect to the typing rules. On the other hand, the latter group, denoted as ‘dynamically typed’, allows execution of (possibly) inconsistent programs, but will handle a type-error gracefully at run-time. Hence, dynamically typed programs are considered less robust, because a type error can occur at any time. Some languages combine features of both by performing most validations statically, but inserting run-time checks for validations that cannot be performed at compile-time (e.g. checked downcasting in Java).

Secondly, when the run-time value of an expression is guaranteed to have the type that the type system computed, the language is called type-safe (sometimes denoted as 'strongly typed'). Other languages have constructs (e.g. unchecked casts or untagged unions) that allow a run-time value of a particular type to be handled as if it has another type. Again, the trade-off between those two is stronger guarantees about program robustness versus flexibility for the programmer.

While there is a long-standing debate about the question whether the guarantees that static type checking and type safety provide are worth the restrictions they impose (Cartwright and Fagan 1991; Meijer and Drayton 2004; Siek and Taha 2007), a large group of programmers consider static type checking and type safety as valuable, increasing their confidence in the stability of their program. As will follow from the line of reasoning in the remainder of this section, we focus on static type-checking of type-safe languages in this thesis.

In the 1960s, when the first programming languages that were largely type-safe, such as ALGOL (Tanenbaum 1978), were designed, projects were usually written in one language. In those settings, validating *project-level* consistency is equivalent to checking whether the *program* source is well-formed with respect to the typing rules of the language that is used. However, the software development scene today is rather different. Most industrial projects implement their business logic in a general-purpose language, but use sub-programs written in other languages for secondary tasks. Examples include: query languages, configuration languages and other programs called using foreign function interfaces. Additionally, contemporary web standards force web application developers to write their frontends in several languages: HTML for data representation, CSS for styling and JavaScript for interactivity.

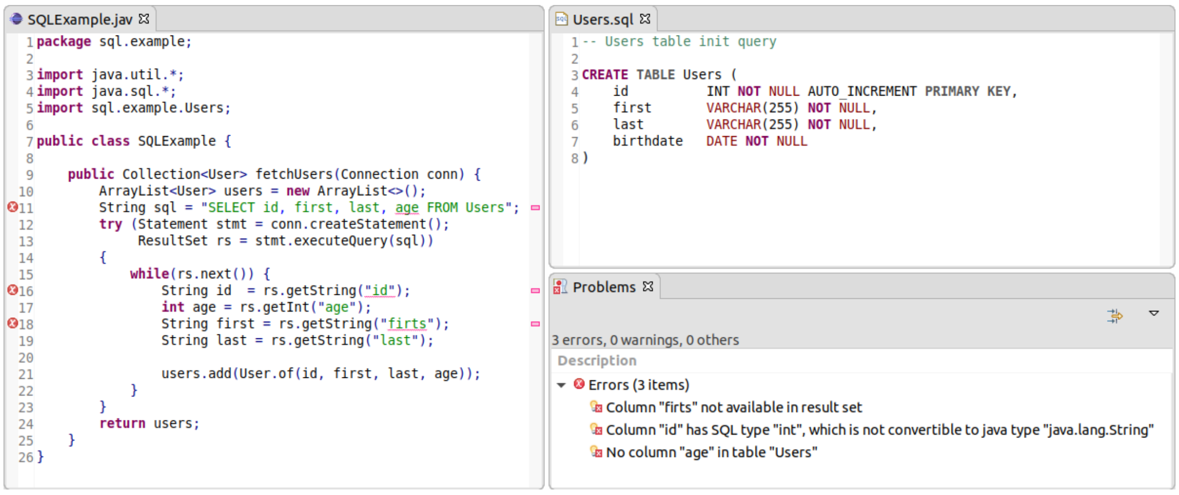
As a natural extension of the single-language project consistency checking by type systems, one would expect multi-language projects to use some form of cross-language consistency checking mechanism. Yet, with some exceptions (of which a few are discussed in section 2.3), this is not uniformly the case. Neither has science developed a principled, language-independent approach to leverage static analysis to cross-language projects. Therefore, we introduce the *'Cross-Language Static Semantics Problem'*, which can be defined as:

> How to provide a formal and executable specification of the static semantics of interactions between parts of a software system written in different languages.

This is a new name for an old problem. For example, Hemel, Groenewegen, Kats, and Visser (2011) write in their abstract:

> Modern web application development frameworks provide web application developers with high-level abstractions to improve their productivity. However, their support for static verification of applications is limited. Inconsistencies in an application are often not detected statically, but appear as errors at run-time. The reports about these errors are often obscure and hard to trace back to the source of the inconsistency. A major part of this inadequate consistency checking can be traced back to the lack of linguistic integration of these frameworks. Parts of an application are defined with separate domain-specific languages, which are not checked for consistency with the rest of the application.

However, more has been done in this research area. For example, research by Pfeiffer and Wasowski (2012a) shows that cross-language support mechanisms, which include static checking, navigation and refactoring, are highly beneficial for developer efficiency. This work, among others (e.g. (Kullbach, Winter, Dahm, and Ebert 1998; Pfeiffer and Wasowski 2011)) show that the Cross-Language Static Semantics Problem is prevalent in contemporary software development, and well worth solving.



```
 1 package sql.example;
 2
 3 import java.util.*;
 4 import java.sql.*;
 5 import sql.example.Users;
 6
 7 public class SQLExample {
 8
 9     public Collection<User> fetchUsers(Connection conn) {
10         ArrayList<User> users = new ArrayList<>();
11         String sql = "SELECT id, first, last, age FROM Users";
12         try (Statement stmt = conn.createStatement();
13              ResultSet rs = stmt.executeQuery(sql))
14         {
15             while(rs.next()) {
16                 String id  = rs.getString("id");
17                 int age = rs.getInt("age");
18                 String first = rs.getString("firts");
19                 String last = rs.getString("last");
20
21                 users.add(User.of(id, first, last, age));
22             }
23         }
24         return users;
25     }
26 }
```

```
1 -- Users table init query
2
3 CREATE TABLE Users (
4     id          INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
5     first       VARCHAR(255) NOT NULL,
6     last        VARCHAR(255) NOT NULL,
7     birthdate   DATE NOT NULL
8 )
```

3 errors, 0 warnings, 0 others

Errors (3 items)
- Column "firts" not available in result set
- Column "id" has SQL type "int", which is not convertible to java type "java.lang.String"
- No column "age" in table "Users"

Figure 2.1: Example of Cross Language Consistency Validation

2.2 Goal

Before specifying precisely what criteria a solution to the Cross-Language Static Semantics Problem needs to adhere to, we sketch our ideas of a programming environment for which this problem is solved.

2.2.1 Project-level consistency

Foremost, we have argued that type systems are used to rule out programs that are not composed correctly. However, 'rule out' can have two different meanings: not compiling a program or giving error messages in the editor. In a multi-language programming environment, we envision both of these. First, we want the compiler to validate composition across language boundaries for the same reasons as people use static type-safe compilers: increased robustness and programmer efficiency. Additionally, programmers should get early feedback about type errors. A fictional example about what that could look like is given in Figure 2.1. This figure shows an example of a database query, executed from Java code using the JDBC API. The use of this API is validated against the table definition that would be created by the query in `Users.sql`. This table definition is resolved by the `import sql.example.Users;` statement at line 5. For the invalid parts of the query and the processing of its results,

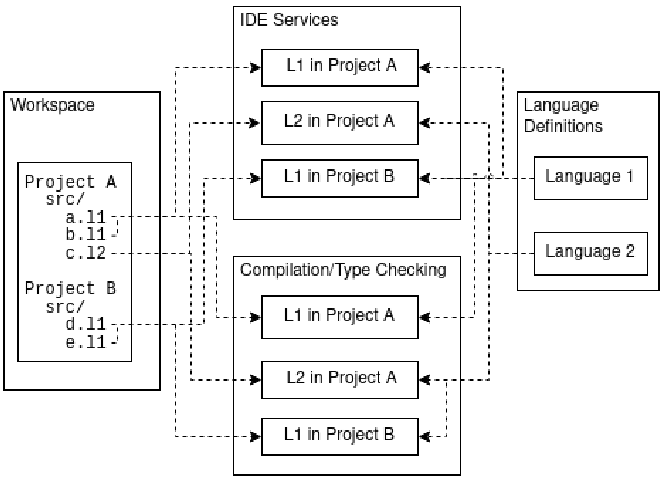


Figure 2.2: Current Programming Environment Organization

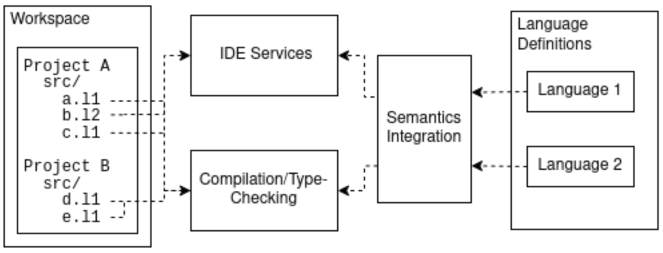


Figure 2.3: Envisioned Future Programming Environment

which involve unresolved names and type mismatches, detailed semantic error messages are provided.

To enable this level of integration, the general architecture of the programming environment needs to be reconsidered. In Figure 2.2, the current organization of a programming environment is shown. Within a project, the sources in a particular language are analyzed and compiled together, but isolated from the source sets in other projects or other languages. In Figure 2.3, we visualize how we think it should be organized in the future. The different IDE services and compilers are collapsed into a single one, performing their tasks integrated for the complete workspace. Those components are not powered by individual language definitions anymore, but by a new component, here called *Semantics Integration*. This thesis provides a implementation of such a component for the Spoofax 3 framework.

Thus, in short, from the developer perspective, the tools they work with (compiler and editor) should behave as fully 'understanding' the interactions between all the languages in the system, so that no manual reasoning or testing is necessary to be confident that the program will give no type or type-conversion related runtime errors.

2.2.2 Beyond consistency checking

The applications of regular static semantics that we want to leverage to multi-language programming environments go beyond consistency validation only. Many editor services, such as reference resolution, rely on static analysis. These use-cases could be leveraged to cross-language situations as well. For example, in the example in Figure 2.1, consider navigating to the `CREATE TABLE Users` statement in `Users.sql` from the query string in `SQLExample.java`, or to one of its column definitions from the `ResultSet` getter methods. Such options can make navigation of codebases more efficient, and help programmers in understanding the system they are working with.

Moreover, automated refactoring possibilities can be made more powerful and precise when cross-language analysis information is taken into account. Consider these two examples:

- Renaming: many IDEs provide options to rename code elements, such as methods, fields and variables. With cross-language name resolution information taken into account, these renamings can additionally change the references to the renamed element from sources in other languages. In addition, pairs of names that reference the same concept from different languages could be kept in sync. To illustrate those, consider the SQL example again. When an SQL column is renamed (e.g. `last` to `last_name`), cross-language reference resolution information could be used to update the query string. Moreover, when the type system is aware that the `User` entity class (not included in the figure) corresponds to that table, it can additionally update the field name that corresponds to the column. Such more advanced options make it easier to keep the naming in a codebase consistent, which makes it easier to understand for a developer.

- Safe deletion: many IDEs validate whether a component/file was not used by another component when it is deleted. However, it is either the case that the IDE only checks other files in the same language, or checks all sources in a project based on the occurrence of (a part of) the file name. In the former case, the IDE might not issue a warning when a file with a component from another language references is deleted, while in the latter case false positives may arise. Accurate cross-language name binding information can improve the accuracy and precision of this service.

Finally, code metrics, like the different variants of coupling (Anwer, Adbellatif, Alshayeb, and Anjum 2017; Yourdon and Constantine 1979) can be computed more accurately when cross-language name bindings are taken into account. It would even be possible to do empirical research about the effects of the usage of multiple languages in a project based on firm quantitatively substantiated metrics, such as the number of references that cross a language boundary.

In conclusion, we see that the possibility of jointly analyzing program sources opens up a new realm of possibilities for improving software robustness and developer efficiency.

2.3 Alternative approaches

Before specifying precisely what criteria our solution must adhere to, we shortly discuss several alternative approaches that address this problem. For each approach, we identify some weaknesses, which will guide the derivation of the solution criteria in the next section.

2.3.1 Monolithic Language

The first approach to whole-project validation is integrating all language features that are required in a particular domain into a single language. This is the approach taken by WebDSL (Groenewegen, Hemel, Kats, and Visser 2008), which can be seen as "a web language integrating a number of sub-languages for different concerns related to the construction of web applications" (Hemel, Groenewegen, Kats, and Visser 2011). Integrating these sub-languages enables static verification including "cross-aspect consistency checking".

This approach is quite feasible for well-defined domains, such as web applications. However, it is limited in at least two ways: first, only the concepts that are integrated in the language are actually available to the language user. When an application requires encoding a concept from another domain, the problem is reintroduced. Second, a large software project, and hence a complicated language project will usually incur more maintenance and distribution overhead than more modularized approaches.

2.3.2 Semantic Extension

Another approach is designing a language with a type system that is fully compliant with the type system of an existing language, including a well-defined foreign function interface to it. Often, such a language transpiles to the language it took its base type system from, although it does not need to be a syntactic superset of the source language. Several languages that transpile to JavaScript, like CoffeeScript and Dart, belong to this category.

A variation of this pattern occurs when the base type system is not provided by a particular language, but by a runtime platform. This is the case for the JVM language family, which includes (among others) Java, Scala and Kotlin, and the .NET family, including C#, F# and Visual Basic.NET. Both platforms include object-oriented, functional and scripting languages, showing that, on a well-designed platform, a rich variety of languages is possible.

The actual semantic integration that this pattern allows depends on the semantic information the target language or platform preserves. When transpiling to JavaScript, no static validation can be done, because the type system of JavaScript is very weak. On the other hand, languages such as Kotlin and F# actually provide static type-checking of references to components written in Java and C#, respectively.

This approach has at least three particular disadvantages: the languages are a priori limited by the constraints that the source language or platform imposes (especially regarding their execution). Additionally, this approach cannot be implemented for already existing languages. Finally, static type-checking of interoperating code may require compromises to the stronger guarantees of the source language. For example, nullability information is explicit in Kotlin and F#. However, the runtime platforms they operate with do not handle `null` values explicitly. Hence, the respective compilers or language users need to deal with incoming `null` values, for example by inserting runtime checks. When using Kotlin, the compiler inserts runtime assertions when assigning to non-null variables[1], while F# leaves nullability checking up to the user[2].

2.3.3 Code Generation

Yet another approach to bridge the gap between languages is by generating the sources of a language from sources of another language. Approaches to code generation vary a lot; it can be done by an external source processor, by a plugin to an extensible compiler, macro expansion (Chang, Knauth, and Greenman 2017), or by a runtime library. Examples that use code generation include:

- Rust BindGen[3] generates Rust sources that access C/C++ libraries.
- Hibernate[4] is an Object-relational mapping framework for Java, which generates SQL queries at runtime, based on annotations given to Java entity classes.
- In the J% Java preprocessor, a type-checked SQL embedding has been implemented (Karakoidas, Mitropoulos, Louridas, and Spinellis 2015). This preprocessor type-checks SQL queries with Java code, but allows validation against an existing database schema as well.

In general, these approaches can be divided into two categories, based on the sort of code they generate. First, bindings can be generated from heterogeneous references, as is the case for Rust BindGen and J%. Conversely, sometimes the bindings are specified, and the code of the target language is generated instead. This is the approach taken by Hibernate.

Extensible compilers can be used to extend type-checking to cross-language interactions as well. In this case, no code is generated, but it is verified that the bindings match the code of the target language. For example the `sqlx`[5] crate extends the Rust compiler to validate manually written SQL queries at compile-time.

Although this approach is the most powerful of all the approaches discussed in this section, four weaknesses can be named. At first, when using code generation, the consistency of the project relies on the correctness of the source generator. Because such code generators should be reasonably well tested, and are therefore less likely to introduce errors than unvalidated manually written code, project consistency will mostly be improved. However, a code generator does not provide a formal specification of the linguistic integration it introduces, and hence gives no strict guarantees about the consistency of a project. Moreover, transparency regarding the code that is generated is lost. This may cause confusion to the developer, especially when error messages are not traced back to the original source code. In addition to that, the number of features of the target language that can be used is limited by the features the source generator provides[6]. Finally, in many cases, editor support is rather limited.

[1]`https://kotlinlang.org/docs/reference/java-interop.html#null-safety-and-platform-types`

[2]`https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/values/null-values`

[3]`https://github.com/rust-lang/rust-bindgen`

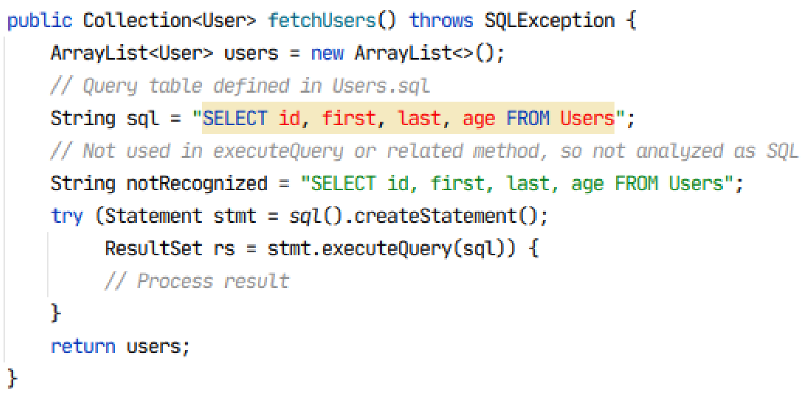
[4]`https://hibernate.org/`

[5]`https://github.com/launchbadge/sqlx`

[6]A funny illustration hereof can be found at the *Hibernate Formula and Transformer annotations*, which reintroduce string embeddings of SQL, the very language it abstracts over.

2.3.4 Editor Plugins

Lastly, several IDEs provide multi-language analysis. For example, IntelliLang plugin for IntelliJ has a Language Injection feature[7], which offers "comprehensive code assistance" for pieces of code embedded in literals of other languages. For example, as can be seen in Figure 2.4, it recognizes SQL queries in Java String literals. Similar features exist in Eclipse (Nagy and Cleve 2018) and (for C#) in Visual Studio[8].



```
public Collection<User> fetchUsers() throws SQLException {
    ArrayList<User> users = new ArrayList<>();
    // Query table defined in Users.sql
    String sql = "SELECT id, first, last, age FROM Users";
    // Not used in executeQuery or related method, so not analyzed as SQL
    String notRecognized = "SELECT id, first, last, age FROM Users";
    try (Statement stmt = sql().createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {
         // Process result
    }
    return users;
}
```

Figure 2.4: SQL String recognition in IntelliJ

For this approach, there are three prominent deficiencies as well: Firstly, the provided analysis is not integrated in the compiler pipeline, and is therefore not invoked at build-time. Secondly, only syntactical analysis of the string embedding is performed. Although most SQL integrations permit validation against a database scheme, the semantic interactions between the languages involved (e.g. SQL and Java in this example) are not taken into account. Thirdly, the detection of the embedding heavily relies on (configurable) heuristics, which make it prone to mistakes and incompleteness.

2.3.5 Evaluation

Summarizing, it can be seen that several approaches to solve the Cross-Language Static Semantics Problem have been explored in academia as well as in industry. However, each approach has limitations that prohibit it from fully realizing the goal we sketched in section 2.2. Before we make the criteria for a solution more precise, we address some more intricate problems that all approaches mentioned above share.

Generality. All solutions mentioned in the previous section have one thing in common: they solve the problem in a particular situation. They offer no conceptual foundation, nor a language-parametric framework, that can be used for other implementations as well. Such a framework would be desirable to have uniformity, and to reduce implementation effort.

Mutual Agnosticism. Furthermore, in all current situations except for the *Semantic Extension* of a platform, the integrated languages are not mutually agnostic. This causes scalability issues, because for any pair of languages for which consistency checking is desired, a new integration must be implemented. Therefore, to integrate n languages, $O(n^2)$ composition implementations are needed. To make cross-language analysis common-place, an approach that requires only $O(n)$ effort is needed.

Editor/Build Support. In addition to that, the *Editor Plugin* approach does not provide build integration, while the *Code Generation* and *Semantic Extension* approaches do not (trivially) provide editor support. It is important that analysis in the editor and at compile-time provides the same results, to increase developer efficiency and reduce confusion. Therefore we want a approach where one type system specification provides both services.

[7] https://www.jetbrains.com/help/idea/using-language-injections.html
[8] https://marketplace.visualstudio.com/items?itemName=PKochubey.VerifyRawSQL

Additional Component. In the *Code Generation* and the *Editor Plugin* approaches, the integration is not provided by the language specification itself, but rather by a third-party component. We think the cross-language capabilities should be included in the language specification itself, to foster reusability and uniformity, reduce development effort, and ensure that the integrations stay up-to-date with the language.

2.4 Solution Criteria

By assessing the desired programming environment, and the advantages and disadvantages of all current approaches, we derived a set of criteria to which a solution of the Cross-Language Static Semantics Problem should adhere. These criteria guarantee that the pitfalls discussed in the previous sections are avoided.

1. Derivability: cross-language type-checking should be derivable directly from the type system specifications.
2. Editor/Build Support: derived type checking should support both type-checking in a compiler and in an editor.
3. Loose Coupling: cross-language type-checking does not require a dependency between type system specifications.
4. Correctness: the type systems of individual languages give correct results in any composition setting.

In the following sections, we explain each criterion, and elaborate on how these solve the aforementioned problems.

2.4.1 Derivability

The Derivability criterion states that cross-language type-checking should not be specified *upon* the type system specification of a language, but *in* it. The type system specification should be designed for cross-language analysis in the first place. By designing a language this way, no additional effort needs to be taken to add cross-language analysis to it. Moreover, the integration with other languages stays up-to-date and has stronger correctness guarantees.

2.4.2 Editor/Build Support

As explained in the previous sections, we want the type-checking in the editor and in the compiler to be guaranteed to give the same results. When the compiler gives errors that the editor does not mention, the developer may be notified rather late, which reduces efficiency. On the other hand, when the editor gives an error the compiler does not emit, either the type-checking in the compiler is invalid or the editor reports false positives. By deriving both from the same specification, their outputs are forced to be correct and consistent.

2.4.3 Loose Coupling

Furthermore, a solution to the Cross-Language Static Semantics Problem should not require type system specifications to depend on each other to do combined analysis. Instead, it should provide a mechanism through which mutually agnostic type systems can interact. This approach guarantees $O(n)$ type system development effort, and guarantees language users that they do not incur the overhead of a type system for a language they do not use.

2.4.4 Correctness

Finally, the type systems should behave well in all possible use cases. That is: type systems that are type-safe should remain type-safe in all usage scenarios. The different scenarios for which this criterion must hold can be categorized in the following three groups:

- Isolation: this situation happens when the language in question is the only language used in a project. In that case, static analysis should only provide results regarding the sources in that language.
- Disjoint Analysis: in this situation, the project is implemented in multiple languages, but no cross-language analysis is desired. In that case the type systems of the different languages should be guaranteed not to have any accidental interaction, because that can invalidate analysis results.
- Joined Analysis: in this situation, the type systems of the languages in question do interact. This interaction should be specified unambiguously, and the implementation should adhere to the specification. Furthermore, the parts of the individual type systems that are language-specific should be guaranteed *not* to interact, like in the *Disjoint Analysis* environment.

Besides the fact that *Correctness* is needed for a type system to have real value, combined with the *Loose Coupling* criterion it provides considerable flexibility.

Now that we have defined precise conditions to which a solution of the Cross-Language Static Semantics Problem should adhere, we present our solution for defining multi-language type systems in chapter 4, and the implementation of its runtime in chapter 5. But before that, we explore the problem more precisely by presenting two case studies we performed in chapter 3.

Chapter 3

Case Studies Introduction

In this chapter, we introduce the case studies we performed. These case studies serve two goals. First, they serve as more detailed examples underlining the vision we provided in chapter 2. Second, we use them as means to validate the implementation of our framework.

The first case study, discussed in section 3.1, is an integration of a syntax definition language and a term rewrite language. First, the syntax and semantics of both languages are discussed independently. After that, the way their distinctive features interact is explained. In section 3.2, we present the second case study, which integrates a small mod/record language with SQL. The structure of this section is similar to that of the first case study: we first discuss both languages individually, and subsequently present their integration.

Note that this chapter describes the syntax of the languages involved in the case studies, and gives an informal description of their static semantics. An explanation about the implementation of these semantics in our framework is given in section 4.3, while an evaluation is provided in chapter 6. Readers familiar with any of the involved languages are recommended to skim through the sections and the examples about the individual languages, and only read about the way they integrate.

3.1 Mini-SDF and Mini-Stratego

The first case study we performed was integrating a syntax definition and a rewrite system. In this section we will first introduce both languages, and then describe how their integration works. For reference, its full specification is included in Appendix A.

3.1.1 Syntax of Mini-SDF

The first language in this case study is Mini-SDF. Mini-SDF is a small subset of the SDF3 Syntax Definition Formalism (Souza Amorim 2019). SDF3 is a versatile meta-language that can be used to define context-free grammars. In addition it offers features to add constructor names, disambiguation, layout constraints and formatting rules (Souza Amorim and Visser 2020). In Mini-SDF, only the features that are required to define context-free grammars with template productions are included.

Essential to this case study is the fact that a signature can be derived from a context-free grammar. Such signatures describe the structure of abstract syntax trees, and as such can be used for validating rewrite systems. Therefore we extracted the features of SDF3 that contribute to the signature definition, which are sort declarations and template productions (Vollebregt, Kats, and Visser 2012), into Mini-SDF. The syntax of Mini-SDF is given in Figure 3.1.

$$
\begin{aligned}
\mathit{id} &::= \text{Any identifier}\\
\mathit{lit} &::= \text{Any string literal not containing angle brackets}\\
\mathit{module} &::= \textbf{module}\ \mathit{id}\ \mathit{section}^*\\
\mathit{section} &::= \textbf{imports}\ \mathit{id}^* \mid \textbf{sorts}\ \mathit{id}^* \mid \textbf{context-free syntax}\ \mathit{prod}^*\\
\mathit{prod} &::= \mathit{id}.\mathit{id}\ \textbf{=}\ \textbf{<}\mathit{term}^*\textbf{>}\\
\mathit{term} &::= \mathit{lit} \mid \textbf{<}\mathit{sort}\textbf{>}\\
\mathit{sort} &::= \mathit{id} \mid \mathit{id}\textbf{*} \mid \mathit{id}\textbf{+} \mid \mathit{id}\textbf{?}
\end{aligned}
$$

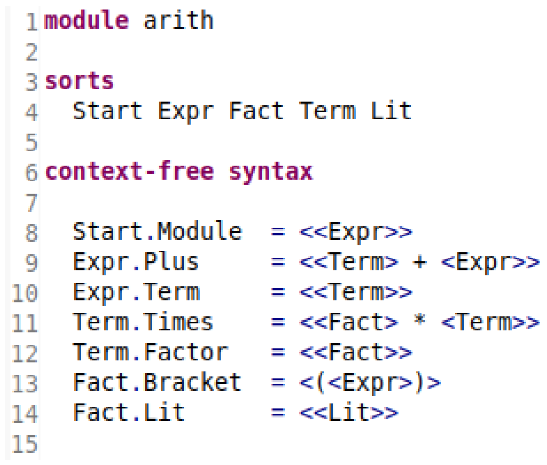
Figure 3.1: Syntax definition for Mini-SDF

A Mini-SDF module can contain three kinds of sections:

- *Sort declarations* declare (non-)terminal symbols.
- *Context-free Syntax declarations* declare productions. Each production consists of a sort name and a constructor name on the left-hand side, and a series of symbols on the right-hand side. A symbol can either some text literal (any sequence of characters not containing angle brackets or whitespace), or a reference to a non-terminal symbol, possibly decorated with a postfix arity operator ($?$, $*$ or $+$), enclosed in angle brackets.
- *Imports* make the sort and constructor declarations of the imported module visible in the importing module.

Note that Mini-SDF is not a complete syntax formalism, because it lacks features to describe lexical syntax and layout. However, it offers all features that are needed to derive a signature for an abstract syntax, and therefore it suffices for our purpose.

In Figure 3.2a, an example Mini-SDF module is shown. The syntax definition in this module, called `arith`, describes a left-factored grammar for arithmetic expressions.



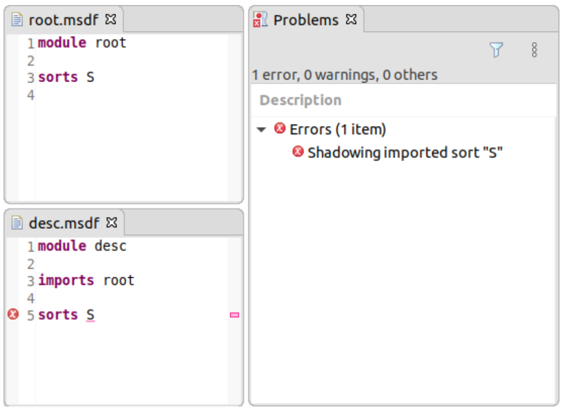
```
module arith

sorts
  Start Expr Fact Term Lit

context-free syntax

  Start.Module  = <<Expr>>
  Expr.Plus     = <<Term> + <Expr>>
  Expr.Term     = <<Term>>
  Term.Times    = <<Fact> * <Term>>
  Term.Factor   = <<Fact>>
  Fact.Bracket  = <(<Expr>)>
  Fact.Lit      = <<Lit>>

```

(a) Example Mini-SDF syntax definition

$$
\begin{aligned}
S = \{\ & \mathit{Start},\ \mathit{Expr},\ \mathit{Term},\ \mathit{Fact},\ \mathit{Lit}\ \}\\
\Sigma = \{\ & \mathit{Module} : \mathit{Expr} \to \mathit{Start}\\
& \mathit{Plus} : \mathit{Term} \times \mathit{Expr} \to \mathit{Expr}\\
& \mathit{Term} : \mathit{Term} \to \mathit{Expr}\\
& \mathit{Times} : \mathit{Fact} \times \mathit{Term} \to \mathit{Term}\\
& \mathit{Factor} : \mathit{Fact} \to \mathit{Term}\\
& \mathit{Bracket} : \mathit{Expr} \to \mathit{Fact}\\
& \mathit{Lit} : \mathit{Lit} \to \mathit{Fact}\ \}
\end{aligned}
$$

(b) Extracted Signature

Figure 3.2: Syntax Example


root.msdf

```
module root

sorts S

```

desc.msdf

```
module desc

imports root

sorts S
```

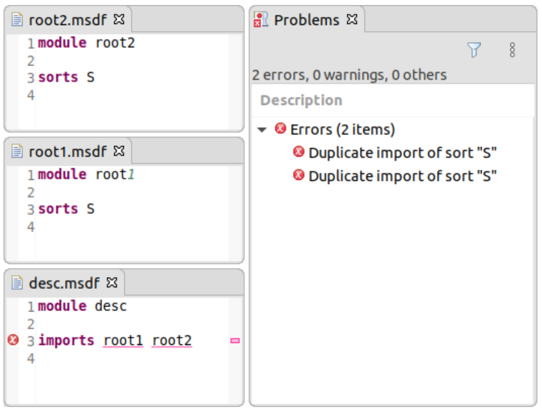
Problems

1 error, 0 warnings, 0 others

Description

- Errors (1 item)
  - Shadowing imported sort "S"

Figure 3.3: Shadowing


root2.msdf

```
module root2

sorts S

```

root1.msdf

```
module root1

sorts S

```

desc.msdf

```
module desc

imports root1 root2

```

Problems

2 errors, 0 warnings, 0 others

Description

- Errors (2 items)
  - Duplicate import of sort "S"
  - Duplicate import of sort "S"

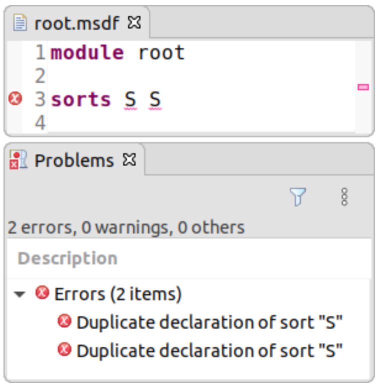
Figure 3.4: Double Import

To illustrate the extraction of signatures from a grammar, the multi-sorted signature (S, Σ) that would be extracted is shown in Figure 3.2b. In this figure, it is visible that every sort declaration translates directly into a sort in the derived signature. Similarly, for each production, a named constructor for its output sort is created. The parameter types of these constructors are the references to the sorts within the angled brackets, such as `<Expr>`. Literals and layout in the productions, such as the arithmetic operators and the brackets, are not present in the constructors, because they do not contribute to the abstract structure of a program. Finally, the `Lit` sort, which represents literals has no constructors, because it is lexical by nature, and lexical syntax is omitted from the Mini-SDF specification.

3.1.2 Semantics of Mini-SDF

In this section, we describe the static semantics of the Mini-SDF language. In short, the following constraints hold: sort names are unique, but constructor names can only be overloaded by arity. Regarding visiblity: every sort that is referenced must be visible, but forward references within a module are allowed. Imported modules must exist, and imports are transitive. In the remainder of the section, we will elaborate on these in more detail.

Unique Sorts First of all, sort symbols should be unique. This prevents accidental merging of two different syntactic categories, when they would both be imported in a different module. There are several patterns in which this constraint can be violated. These patterns include: declaring the same sort in the same module (Figure 3.5), shadowing an imported sort (Figure 3.3) and importing a sort from two different modules (Figure 3.4). On the other hand, it is allowed to import a particular sort via multiple paths, as Figure 3.6 demonstrates.


root.msdf

```
module root

sorts S S

```

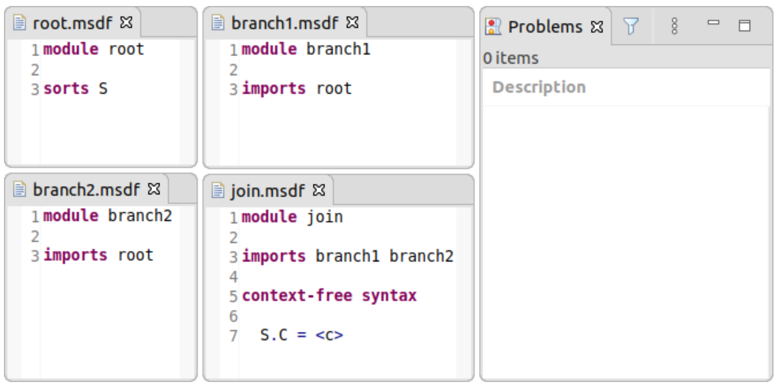
Problems

2 errors, 0 warnings, 0 others

Description

- Errors (2 items)
  - Duplicate declaration of sort "S"
  - Duplicate declaration of sort "S"

Figure 3.5: Double Sort name


root.msdf

```
module root

sorts S
```

branch1.msdf

```
module branch1

imports root
```

branch2.msdf

```
module branch2

imports root
```

join.msdf

```
module join

imports branch1 branch2

context-free syntax

  S.C = <C>
```

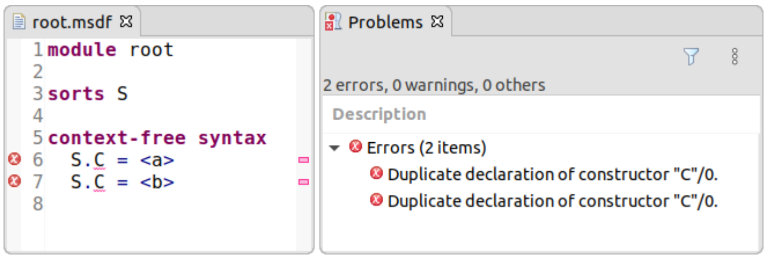
Problems

0 items

Description

Figure 3.6: Diamond import

Unique Constructors. For constructors, the same constraints as for sorts hold: duplicate declarations are not allowed, even not for different sorts (Figure 3.7). However, even when the constructors have a different number of arguments, overloading is allowed (Figure 3.8). The rationale behind this behavior is that it is always possible to derive syntactically what instance of the constructor was meant to be used.


root.msdf

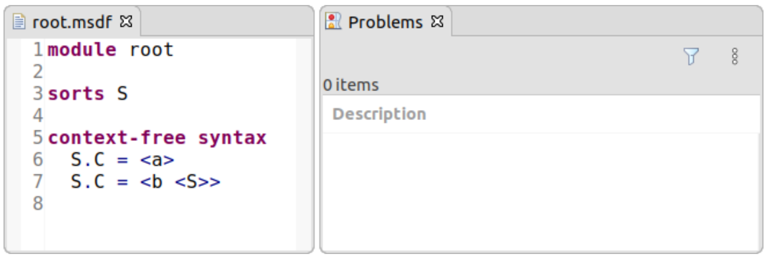
```
module root

sorts S

context-free syntax
  S.C = <a>
  S.C = <b>

```

Problems

2 errors, 0 warnings, 0 others

Description
- Errors (2 items)
  - Duplicate declaration of constructor "C"/0.
  - Duplicate declaration of constructor "C"/0.

Figure 3.7: Duplicate Constructor name


root.msdf

```
module root

sorts S

context-free syntax
  S.C = <a>
  S.C = <b <S>>

```

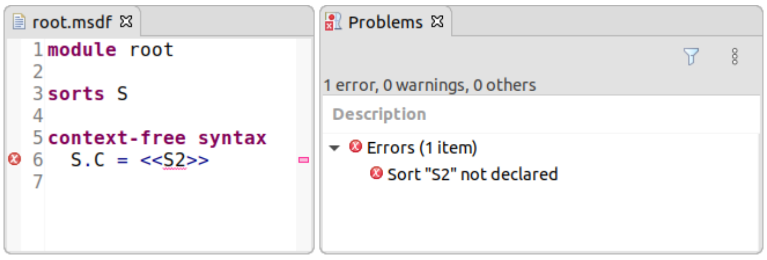
Problems

0 items

Description

Figure 3.8: Constructor Overloading

Transitive Imports. Further, imports are transitive in MiniSDF (which is shown in Figure 3.6 as well). This behaviour has two reasons. First of all, the uniqueness checks in the previous paragraph are only correct and useful when the occurrences they check for are transitively visible. Secondly, when imports are not transitive, situations where a constructor is visible, but its sort is hidden could occur. This would complicate type-checking of MiniStratego patterns (which we explain in subsection 3.1.4).


root.msdf

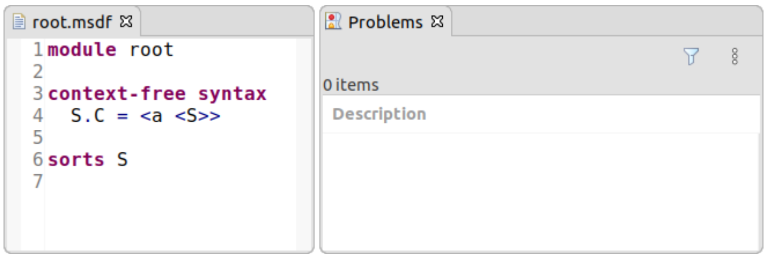
```
module root

sorts S

context-free syntax
  S.C = <<S2>>

```

Problems

1 error, 0 warnings, 0 others

Description
- Errors (1 item)
  - Sort "S2" not declared

Figure 3.9: Undeclared Sort


root.msdf

```
module root

context-free syntax
  S.C = <a <S>>

sorts S

```

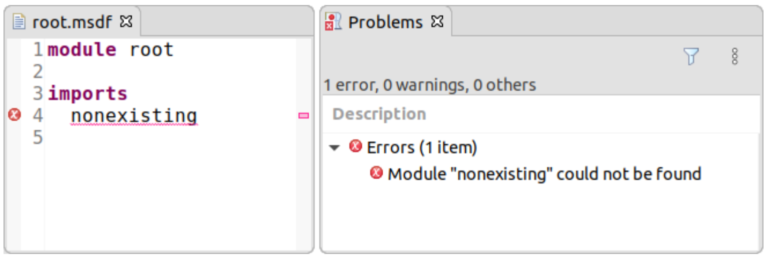
Problems

0 items

Description

Figure 3.10: Forward Reference

Sort References. Moreover, sorts that are referenced in productions must be declared. In order to validate that, they should be imported in the module that references them. When a referenced sort is not visible, an error is given (Figure 3.9).


root.msdf

```
module root

imports
  nonexisting

```

Problems

1 error, 0 warnings, 0 others

Description
- Errors (1 item)
  - Module "nonexisting" could not be found

Figure 3.11: Unresolved Import

Reference Direction. Fifth, there is no notion of declaration order within a module. Therefore it is possible to reference sorts that are declared later in a file, as Figure 3.10 shows.

$$
\begin{array}{rcl}
\mathit{id} & ::= & \text{Any identifier} \\[6pt]
\mathit{module} & ::= & \textbf{module}\ \mathit{id}\ \mathit{section}^* \\[6pt]
\mathit{section} & ::= & \textbf{imports}\ \mathit{id}^* \mid \textbf{signature}\ \mathit{sig}^* \mid \textbf{rules}\ \mathit{rule}^* \\[6pt]
\mathit{sig} & ::= & \textbf{sorts}\ \mathit{id}^* \mid \textbf{constructors}\ \mathit{cons}^* \\[6pt]
\mathit{cons} & ::= & \mathit{id} : \mathit{id} \mid \mathit{id} : \mathit{sort}\ {*}\ldots{*}\ \mathit{sort}\ \textbf{->}\ \mathit{id} \\[6pt]
\mathit{sort} & ::= & \mathit{id} \mid \mathit{id}{*} \mid \mathit{id}{+} \mid \mathit{id}? \\[6pt]
\mathit{rule} & ::= & \mathit{id} : \mathit{pattern}\ \textbf{->}\ \mathit{pattern}\ \mathit{with}? \\[6pt]
\mathit{pattern} & ::= & \mathit{id} \mid \mathit{id}(\mathit{pattern}, \ldots, \mathit{pattern}) \mid [\mathit{pattern}, \ldots, \mathit{pattern}] \mid \textbf{<}\mathit{id}\textbf{>}\ \mathit{pattern} \\[6pt]
\mathit{with} & ::= & \textbf{with}\ \mathit{strat} \\[6pt]
\mathit{strat} & ::= & \mathit{id}\ \textbf{:=}\ \mathit{pattern} \mid \mathit{strat}\textbf{;}\ \mathit{strat}
\end{array}
$$

Figure 3.12: Syntax definition for Mini-STR

Modules. Lastly, modules are globally visible. There is no notion of nested modules. On top of that, modules that are referenced in an `imports` statement have to exist, as shown in Figure 3.11.

3.1.3 Syntax of Mini-Stratego

The second language that is incorporated in this case study is Mini-Stratego (Mini-STR). This language, which is a small subset of the Stratego/XT language (Bravenboer, Kalleberg, Vermaas, and Visser 2008), encodes rewrite rules for program transformation. Rewrite rules are encoded as rules that match a pattern, and subsequently build a term that might possibly be different than the input term. The syntax of Mini-STR is shown in Figure 3.12. In Figure 3.13, a Mini-STR example program is shown. This program encodes a signature for plus-expressions and brackets, and two rewrite rules that simplify the left- and right-hand side of a plus-expression, respectively.

Just as in Mini-SDF, a Mini-STR module consists of three types of sections. First, the `signatures` section declares the signatures of the data that is transformed by the rewrite rules. It can contain `sorts` subsections, which declared sort symbols, and `constructors` subsections, which declares constructors. Constructor declarations can have two forms: a nullary constructor is declared by its name, and the sort it belongs to. Non-nullary constructors are declared by listing their argument sorts, separated by an asterisk, followed by an arrow and the name

```
module trans

signatures
  sorts Expr Term Lit
  constructors
    Plus    : Term * Expr    -> Expr
    Term    : Term           -> Expr
    Bracket : Expr           -> Term
    Lit     : Lit            -> Term

rules

  simplify-expr: Plus(Bracket(Term(l)), r) -> Plus(l, r)
  simplify-expr: Plus(l, Term(Bracket(r))) -> Plus(l, r)
```

Figure 3.13: Mini-STR Example

of the sort it belongs to. A sort is a reference to a sort symbol, that can be decorated by an postfix arity operator.

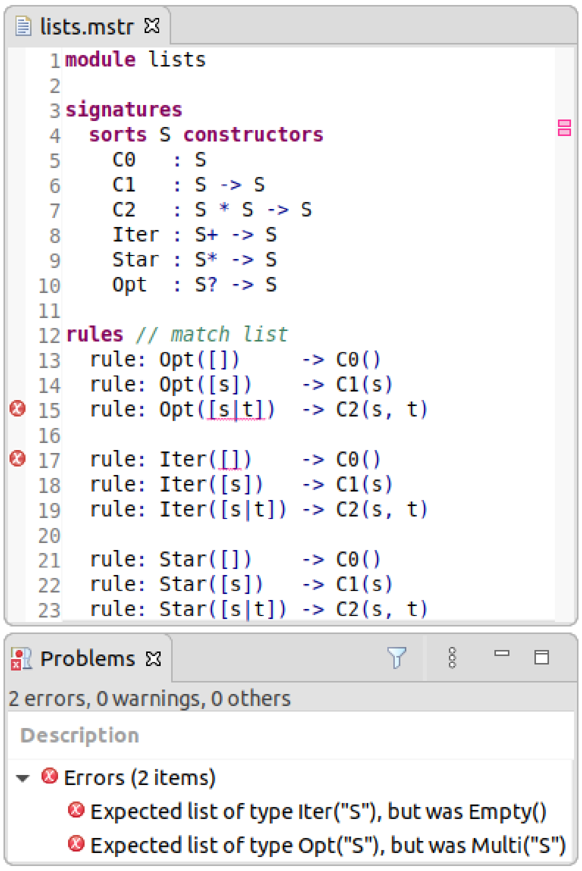
Second, a Mini-STR module contain `rules` sections, which define rewrite rules. A rule has a name, a match pattern, a build pattern and an optional `with` clause. The match patterns define the set of ASTs this rule should be applied to, while the build pattern specifies the output of the rewrite operation. A pattern can either be a variable, a constructor with a number of sub-patterns as arguments, a fixed-size list with sub-patterns as its elements, or a pattern transformed by another transformation rule. Finally, the optional `with` clause defines a series of sub-transformations, whose output terms are assigned to variables. The assignment operations in the with-clauses are executed sequentially, after matching the input, but before building the output.

Finally, the `imports` section makes the sorts, constructors and rules of an imported module visible in the importing module.

3.1.4 Semantics of Mini-STR

In this section, we will explain the semantics of Mini-STR. First, the constraints of Mini-SDF (as explained in subsection 3.1.2) naturally extend to, and actually hold for Mini-STR. However, in Mini-SDF, the constraints regarding visibility were only defined for sorts, because only sorts can be referenced. In Mini-STR, it is possible to reference constructors and other rules by name as well, and therefore the visibility and uniqueness constraints now hold for constructors and rules as well.

Patterns Match patterns as well as build patterns should be well-formed with respect to the signature. For constructors that means that a constructor with the same name and arity of



lists.mstr

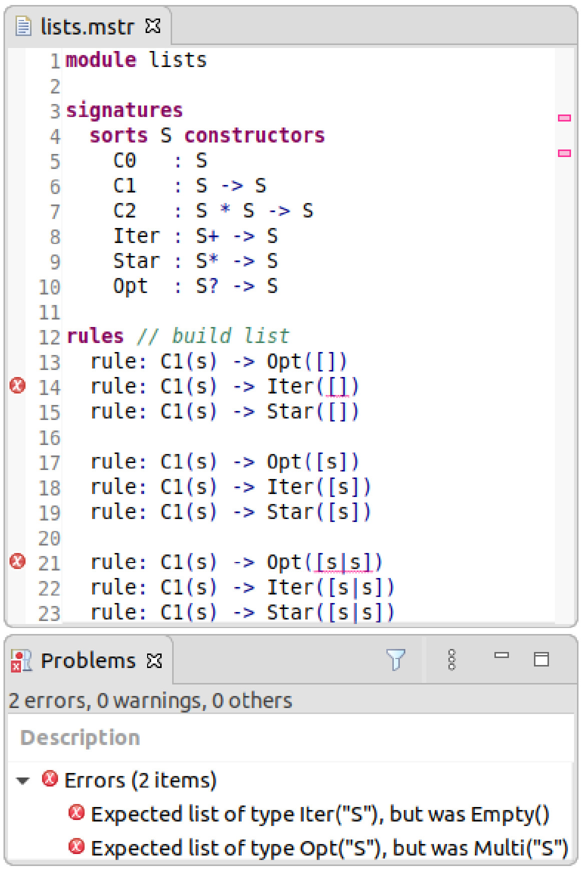
```
module lists

signatures
  sorts S constructors
    C0   : S
    C1   : S -> S
    C2   : S * S -> S
    Iter : S+ -> S
    Star : S* -> S
    Opt  : S? -> S

rules // match list
  rule: Opt([])     -> C0()
  rule: Opt([s])    -> C1(s)
  rule: Opt([s|t])  -> C2(s, t)

  rule: Iter([])    -> C0()
  rule: Iter([s])   -> C1(s)
  rule: Iter([s|t]) -> C2(s, t)

  rule: Star([])    -> C0()
  rule: Star([s])   -> C1(s)
  rule: Star([s|t]) -> C2(s, t)
```

Problems

2 errors, 0 warnings, 0 others

Description
Errors (2 items)
Expected list of type Iter("S"), but was Empty()
Expected list of type Opt("S"), but was Multi("S")

Figure 3.14: Matching a list



lists.mstr

```
module lists

signatures
  sorts S constructors
    C0   : S
    C1   : S -> S
    C2   : S * S -> S
    Iter : S+ -> S
    Star : S* -> S
    Opt  : S? -> S

rules // build list
  rule: C1(s) -> Opt([])
  rule: C1(s) -> Iter([])
  rule: C1(s) -> Star([])

  rule: C1(s) -> Opt([s])
  rule: C1(s) -> Iter([s])
  rule: C1(s) -> Star([s])

  rule: C1(s) -> Opt([s|s])
  rule: C1(s) -> Iter([s|s])
  rule: C1(s) -> Star([s|s])
```

Problems

2 errors, 0 warnings, 0 others

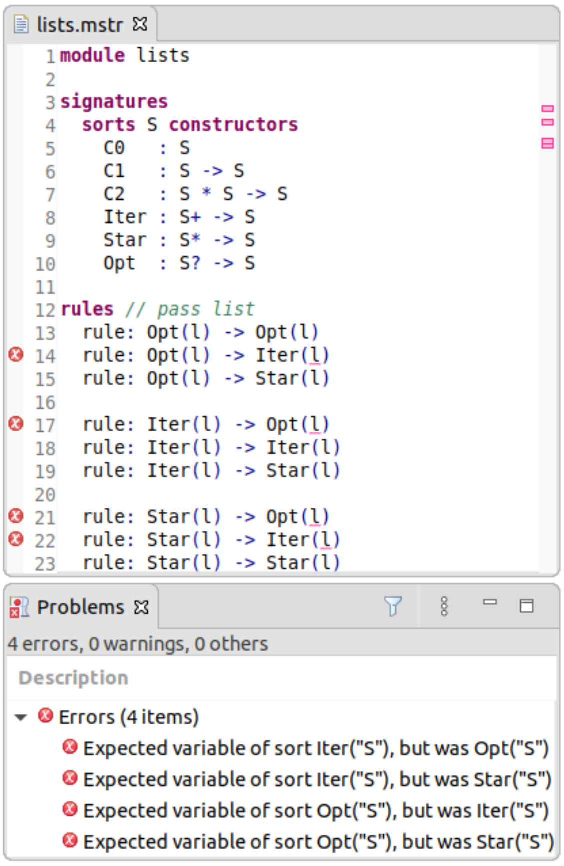
Description
Errors (2 items)
Expected list of type Iter("S"), but was Empty()
Expected list of type Opt("S"), but was Multi("S")

Figure 3.15: Building a list

the pattern should be visible, and the argument patterns should have the sort that is expected in that position. For example, the `Bracket(Lit(a))` pattern would be invalid with respect to the signature provided in Figure 3.13, because the sort of `Lit/1` is `Term`, but the `Bracket/1` constructor expects a term of sort `Expr` as its argument.

The validation of variables differs between match and build patterns. In match patterns, the variable is declared in the scope of the rule with the sort that is expected there. It is not possible to match on the same variable twice. When a variable is referenced in a build pattern, it should be declared with the sort that is expected at that position.

An interesting situation arises when type-checking lists. Lists in patterns correspond to positions in constructors that are decorated with an operator, where an optional operator (`?`) is interpreted as a list with zero or one element. However, a particular list pattern can map to multiple of those. For example, the pattern `[Lit(v)]` can match any of `Lit?`, `Lit*` and `Lit+`. In Mini-STR, we give errors on matches that cannot occur according to the type of the list, as can be seen in Figure 3.14, and we prevent building lists that do not adhere to the type of the list that is expected at that position. These validations are implemented for variables that have a list type as well, as can be seen in Figure 3.16.



```
module lists

signatures
  sorts S constructors
    C0   : S
    C1   : S -> S
    C2   : S * S -> S
    Iter : S+ -> S
    Star : S* -> S
    Opt  : S? -> S

rules // pass list
  rule: Opt(l) -> Opt(l)
  rule: Opt(l) -> Iter(l)
  rule: Opt(l) -> Star(l)

  rule: Iter(l) -> Opt(l)
  rule: Iter(l) -> Iter(l)
  rule: Iter(l) -> Star(l)

  rule: Star(l) -> Opt(l)
  rule: Star(l) -> Iter(l)
  rule: Star(l) -> Star(l)
```

4 errors, 0 warnings, 0 others

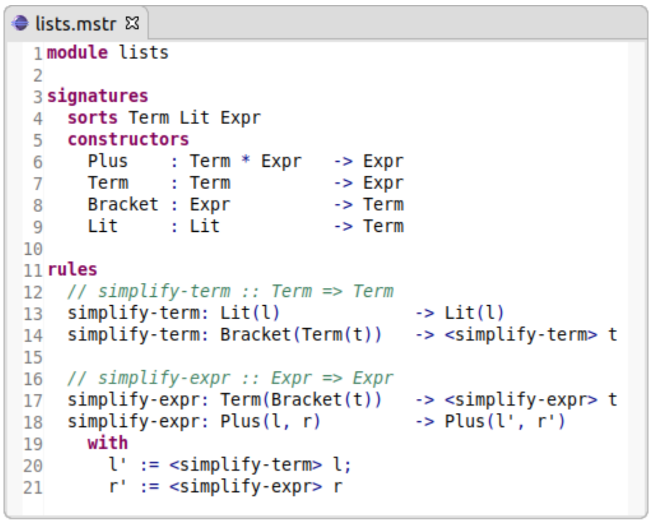
Description
Errors (4 items)
Expected variable of sort Iter("S"), but was Opt("S")
Expected variable of sort Iter("S"), but was Star("S")
Expected variable of sort Opt("S"), but was Iter("S")
Expected variable of sort Opt("S"), but was Star("S")

Figure 3.16: Pass list by variable

Finally, a transformation can be called on a sub-term. Such a pattern is well-formed when its input pattern is well-formed, and has the same sort as the transformation input. The resulting sort is the output sort of the transformation.

Rewrite Rules. A rewrite rule is a transformation of a term to another term. In Mini-STR, a rule is validated by checking that both the match pattern and the build pattern are well-formed. The type of the rule is the pair of the input type and the output type.

As an example, consider the system in Figure 3.17. The system in that example will remove all unnecessary brackets in an expression. The first rule (`simplify-term`) will do that for terms with sort `Term`, while `simplify-expr` does that for expressions. The first `simplify-term` instance (on line 13) serves as a base case: it



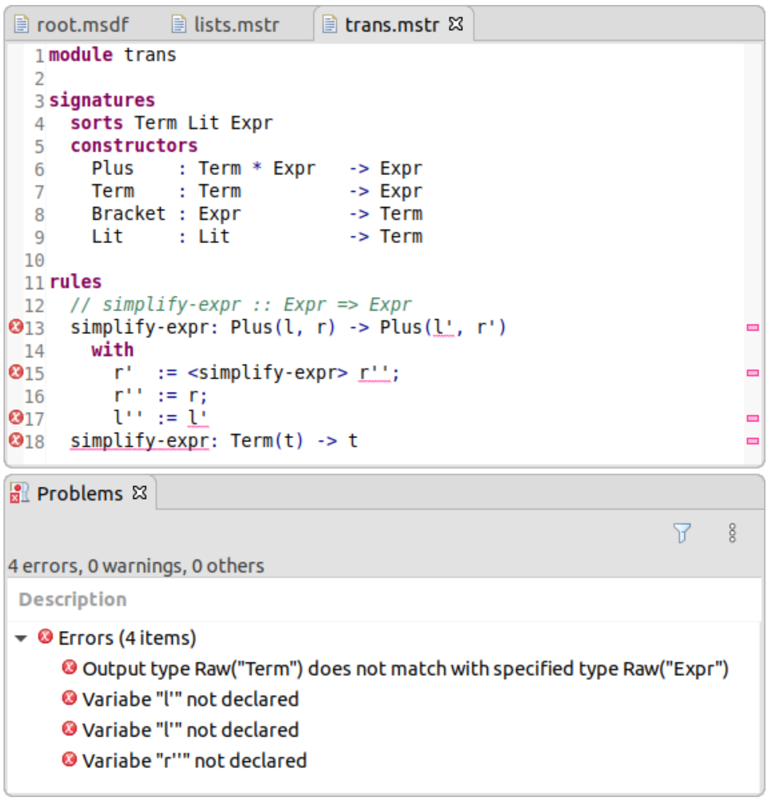
```
 1 module lists
 2
 3 signatures
 4   sorts Term Lit Expr
 5   constructors
 6     Plus    : Term * Expr    -> Expr
 7     Term    : Term           -> Expr
 8     Bracket : Expr           -> Term
 9     Lit     : Lit            -> Term
10
11 rules
12   // simplify-term :: Term => Term
13   simplify-term: Lit(l)              -> Lit(l)
14   simplify-term: Bracket(Term(t))    -> <simplify-term> t
15
16   // simplify-expr :: Expr => Expr
17   simplify-expr: Term(Bracket(t))    -> <simplify-expr> t
18   simplify-expr: Plus(l, r)          -> Plus(l', r')
19     with
20       l' := <simplify-term> l;
21       r' := <simplify-expr> r
```

Figure 3.17: Rewrite rules

matches on literals, but returns them unchanged. The rule on line 14 matches on all terms that have the form `Bracket(Term(t))` (`t` can be any term with sort `Term`), and recursively removes all brackets on its content.

As the aforementioned example shows, multiple rules with the same name can occur. However, in that case it is checked that all the instances of the rule have the same input and output sorts. This check enables the validation of patterns with sub-transformations, without introducing the complication of type-dependent name resolution on transformations that are open for extension. On line 18 of Figure 3.18, such an error is shown.

Finally, the first rule shows a `with`-clause. This clause consists of a sequence of build patterns that are bound to an identifier. This clause is executed sequentially after the match-pattern, but before the build pattern. This means that variables in a build pattern can reference to variables from the input pattern and to variables bound earlier in the sequence, but not to the destination variable of the pattern, to variables that are yet to be build, or to variables in the build pattern of the rule output. Conversely, variables in the build pattern can reference to identifiers bound in the with-clause, as the references to `l'` and `r'` show. Examples of erroneous binding patterns can be seen in Figure 3.18.


root.msdf lists.mstr trans.mstr

```
1 module trans
2
3 signatures
4   sorts Term Lit Expr
5   constructors
6     Plus    : Term * Expr   -> Expr
7     Term    : Term          -> Expr
8     Bracket : Expr          -> Term
9     Lit     : Lit           -> Term
10
11 rules
12   // simplify-expr :: Expr => Expr
13   simplify-expr: Plus(l, r) -> Plus(l', r')
14     with
15       r'  := <simplify-expr> r'';
16       r'' := r;
17       l'' := l'
18   simplify-expr: Term(t) -> t
```

Problems

4 errors, 0 warnings, 0 others

Description

Errors (4 items)
- Output type Raw("Term") does not match with specified type Raw("Expr")
- Variabe "l'" not declared
- Variabe "l'" not declared
- Variabe "r''" not declared

Figure 3.18: Errors in rewrite rules

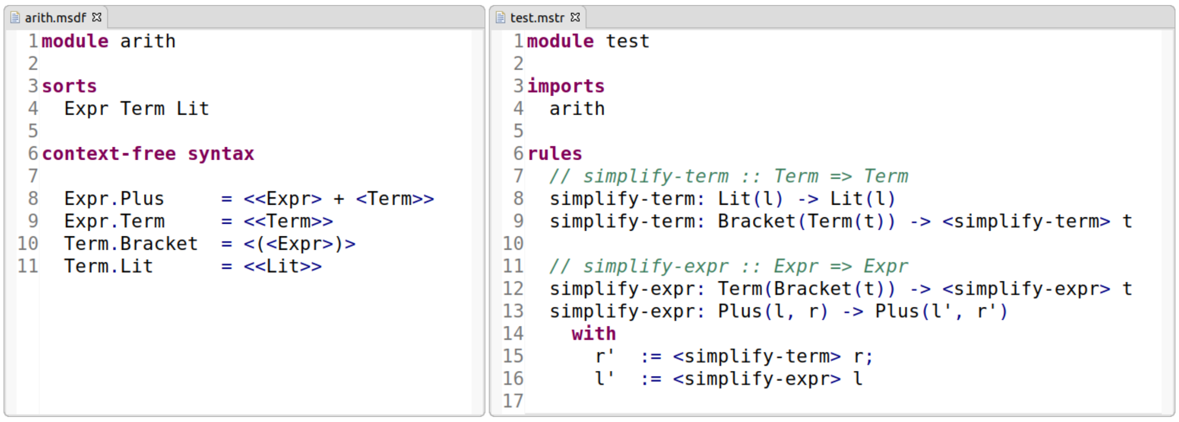
3.1.5 Integration of Mini-SDF and Mini-STR

Now that we have introduced both languages, we describe how their integration works. The core idea is to validate rewrite rules against the signatures that are provided by a syntax definition, instead of a signature provided by a `signatures` section.

In order to implement that, an import in a Mini-STR module can now resolve to Mini-SDF modules as well as to other Mini-STR modules. When a Mini-SDF module is imported, the sorts and constructors extracted from the syntax definition become visible in the Mini-STR module.

An example of what that could look like is depicted in Figure 3.19. At the right-hand side, we see the rewrite system that was introduced earlier. However, instead of a `signatures` section, there is an `imports arith` statement. This statements makes the signature of the grammar on the left visible. These signatures are then used to validate the rewrite system.

The constraints on sort and constructor visibility work the same for Mini-SDF modules imported in Mini-STR as they worked in the individual languages: Both sort symbols and constructor symbols with a particular arity should be defined uniquely. Figure 3.20 shows two Mini-SDF and two Mini-STR modules with a diamond import structure. The errors in `join.str` show that a duplicate import of a sort `S2` declared in both a Mini-SDF and a Mini-

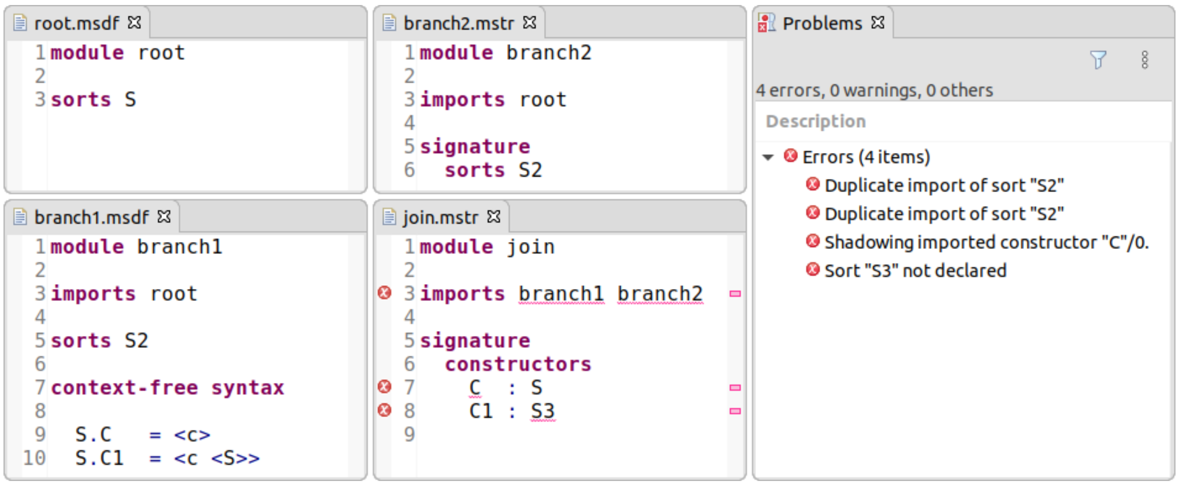

arith.msdf

```
module arith

sorts
  Expr Term Lit

context-free syntax

  Expr.Plus     = <<Expr> + <Term>>
  Expr.Term     = <<Term>>
  Term.Bracket  = <(<Expr>)>
  Term.Lit      = <<Lit>>
```

test.mstr

```
module test

imports
  arith

rules
  // simplify-term :: Term => Term
  simplify-term: Lit(l) -> Lit(l)
  simplify-term: Bracket(Term(t)) -> <simplify-term> t

  // simplify-expr :: Expr => Expr
  simplify-expr: Term(Bracket(t)) -> <simplify-expr> t
  simplify-expr: Plus(l, r) -> Plus(l', r')
    with
      r'  := <simplify-term> r;
      l'  := <simplify-expr> l
```

Figure 3.19: Mini-STR integration with Mini-SDF


root.msdf

```
module root

sorts S
```

branch2.mstr

```
module branch2

imports root

signature
  sorts S2
```

branch1.msdf

```
module branch1

imports root

sorts S2

context-free syntax

  S.C    = <C>
  S.C1   = <C <S>>
```

join.mstr

```
module join

imports branch1 branch2

signature
  constructors
    C  : S
    C1 : S3
```

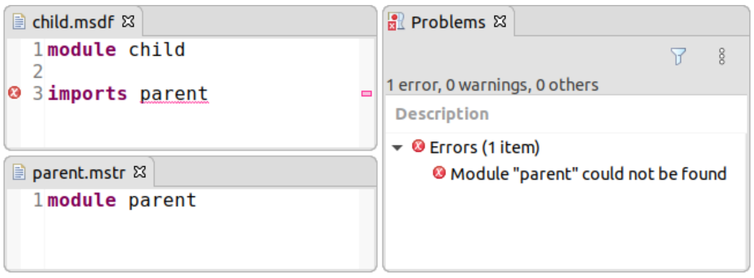
Problems

```
4 errors, 0 warnings, 0 others
Description
Errors (4 items)
    Duplicate import of sort "S2"
    Duplicate import of sort "S2"
    Shadowing imported constructor "C"/0.
    Sort "S3" not declared
```

Figure 3.20: Name resolution from Mini-STR to Mini-SDF

STR module gives an error. Neither is shadowing a constructor imported from Mini-SDF in a Mini-STR module allowed.

Finally, Figure 3.21 shows that an error is emitted when a Mini-SDF module imports a Mini-STR module. The reason for this behavior is that there would be no way to generate a parser from a signature imported from a Mini-STR module, and neither do rewrite rules have any meaning in Mini-SDF.


child.msdf

```
module child

imports parent
```

parent.mstr

```
module parent
```

Problems

```
1 error, 0 warnings, 0 others
Description
Errors (1 item)
    Module "parent" could not be found
```

Figure 3.21: Error on invalid import

3.2 Mod and Mini-SQL

In this section, we consider a more complicated case study: the integration of Mini-SQL and Mod, which is a language with expressions, records, functions and modules. This case study is more complicated than the previous one in two ways:

- For Mod we will adapt an existing type system implementation, where we aim to minimize the number of modifications to the existing specification.

- While Mini-SDF and Mini-STR operate over the same data model, the underlying data models of Mod and SQL are rather different.

This section has a similar structure as the section about Mini-STR and Mini-SDF: first both languages are introduced, and then their integration is described.

3.2.1 Syntax of Mod

The first language that is a part of this case study is Mod. This is an already existing language that is used to experiment with the Statix meta-language. Among its features are:

- Arithmetic and boolean expressions.
- Single-argument functions.
- Sequential, parallel and recursive let-bindings.
- Record types. Its members can be accessed using qualified names or a `with`-expression.
- Modules which can be nested. Its members can be accessed by importing or by qualified names.

For this case study, we extended the language with string literals. In order to understand the examples, a partial syntax specification for Mod is provided in Figure 3.22. In particular, it contains most of the syntax related to record type and expressions.

id ::= Any identifier

program ::= *decl**

decl ::= **$** *expr* | **def** *bind* | **record** *id* { *fdecl* **,** ... **,** *fdecl* }

expr ::= ... | *id* | *id* { *bind* **,** ... **,** *bind* } | *expr***.***id* | **with** *expr* **do** *expr* | *expr expr*

bind ::= *id* **=** *expr*

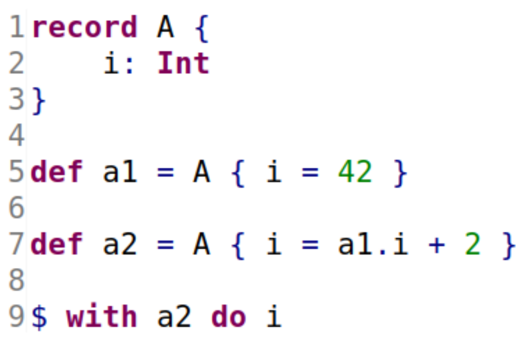
type ::= **Int** | **Bool** | **String** | *type* **->** *type* | *id*

fdecl ::= *id* **:** *type*

Figure 3.22: Partial Syntax definition for Mod

In Mod, a program is a sequence of definitions and expressions. A top-level definition can be a value bound to an identifier or a record type definition. A record type is bound to a type identifier, and contains the name-type pairs of all its fields. The first displayed production for an expression is a reference to a variable, which must be defined earlier using a `def` declaration. The second production initializes a record with the type that is bound to the given identifier, by providing bindings for each field. Subsequently, the . operator can be used to access a field. Additionally, a `with`-expression brings all the variables in a record in scope as top-level definitions, so that they can be accessed without qualifying. Finally, using a juxtaposition of two expressions, a function can be applied to a value.

An example of a program using most of these constructs is shown in Figure 3.23. In this example, lines one to three declare a record type with one field with name `i` and type `Int`. At lines five and seven, two records with that type are created. The initialization expression for the field at line seven contains an example of qualified access to the field `i` of `a1`. At line nine, a `with`-construct is used to access the field `i` of `a2`.

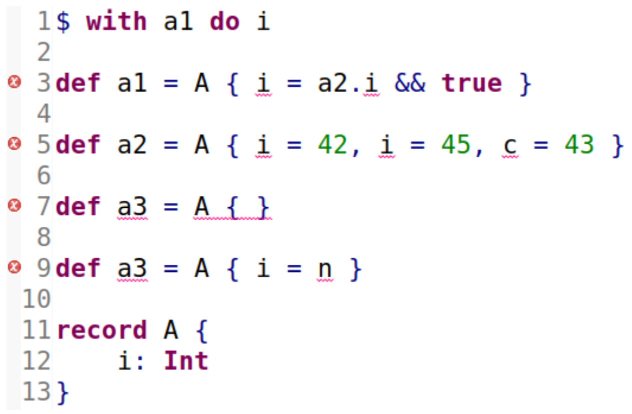


```
1 record A {
2     i: Int
3 }
4
5 def a1 = A { i = 42 }
6
7 def a2 = A { i = a1.i + 2 }
8
9 $ with a2 do i
```

Figure 3.23: Example Mod program

3.2.2 Semantics of Mod

Just as with the syntax definition, we will only provide the semantics of the mod language that are related to variables and records. Examples of all the typing features that are discussed in this section can be seen in Figure 3.24.



```
 1 $ with a1 do i
 2
 3 def a1 = A { i = a2.i && true }
 4
 5 def a2 = A { i = 42, i = 45, c = 43 }
 6
 7 def a3 = A { }
 8
 9 def a3 = A { i = n }
10
11 record A {
12     i: Int
13 }
```

Figure 3.24: Mod program with errors

First, forward references are allowed, as can be seen by the reference to `a2` at line one, and the references to the record type `A` at lines three to nine. Assignments and references should resolve, and have the correct type. For example, at line three, the assignment of a boolean value to an integer field and the reference of an integer field at a position where a boolean value is expected are invalid.

Second, the initialization of fields should be correct. The correctness conditions are:

- A field may not be initialized twice. This is demonstrated by the errors on the `i` fields at line five
- Fields that are not declared may not be initialized. This is shows by the error on the initialization of field `c` at line five.
- All fields that are declared must be initialized. Therefore, the expression at line seven causes an error.

This behaviour is slightly adapted in the integration of the languages, as will be explained in subsection 3.2.5.

Finally, variable names should be unique. This holds for top-level definitions as well as for field declarations. A violation of this rule is the redeclaration of the variable `a3` at line nine. Moreover, referenced variable must exist, as can be seen by the reference to the non-existing variable `n` at line nine.

3.2.3 Syntax of Mini-SQL

The second language incorporated in this case study is Mini-SQL: a subset of the SQL data definition and query language. The syntax of this language is shown in Figure 3.25.

In this language, it is possible to create tables and define stored procedures that perform select queries on these tables. Members of a table are columns, which have a name, data type and possibly some constraints. These column constraints can forbid `NULL` values, or mark a column as primary key. The data type is either integer, date or varchar, which is

```
         id ::= Any identifier

        num ::= Any integer literal

        str ::= Any string literal

    program ::= statement+

  statement ::= CREATE TABLE id (columndef+, tconstraint*);
              | CREATE PROCEDURE id paramdecl* AS select GO;

  columndef ::= id type cconstraint*

       type ::= INTEGER | VARCHAR | VARCHAR(num) | DATE

cconstraint ::= PRIMARY KEY | NOT NULL

tconstraint ::= CONSTRAINT id PRIMARY KEY (id)
              | CONSTRAINT id FOREIGN KEY (id) REFERENCES id(id)

  paramdecl ::= @id type

     select ::= SELECT projection FROM tablespec (WHERE condition)?

 projection ::= * | (columnref (AS id)?)*

  columnref ::= id | id.id

  tablespec ::= id (AS id)? | tablespec JOIN tablespec ON condition

  condition ::= condition AND condition | condition OR condition | value = value
              | value <> value | value < value | value > value | value IS NOT? NULL

      value ::= str | num | @id | columnref | CONVERT(type, value)
```

Figure 3.25: Syntax definition for Mini-SQL

the SQL variant of a string. The varchar type can have an optional maximum length, which has a default value of 80. The primary key constraint can alternatively be specified after the column definitions, as a constraint on a table. Another table-level constraint is the foreign key constraint, which declares that a particular column is a reference to another table.

Stored procedures are defined by a `create procedure` statement. A procedure can take arguments, which are declared by name – type pairs. A procedure consists of a `select` statement, of which the result set is returned. The `from` clause specifies the tables to query. A table can optionally be aliased by adding an `as` clause. On top of that, a table can be joined with another table. The projection clause determines the columns that will be included in the result set of the query. Similar to tables, columns can be renamed, using the `as` clause. Finally, the result can be filtered with a condition specified in the `where` clause. Such a condition can be a comparison of two values, or a conjunction or a disjunction of multiple conditions. A value is a literal, or a reference to a column or procedure argument. Furthermore, the `convert` function can be used to convert values to another type.

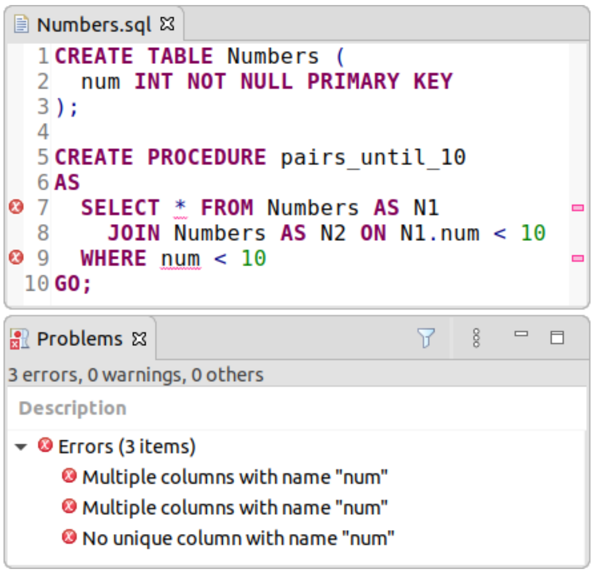
```
 1 CREATE TABLE Users (
 2     user_id      INT          NOT NULL PRIMARY KEY,
 3     username     VARCHAR      NOT NULL,
 4     email        VARCHAR(320)
 5 );
 6
 7 CREATE TABLE Logins (
 8     login_id     INT          NOT NULL,
 9     user         INT          NOT NULL,
10     timestamp    DATE         NOT NULL,
11     CONSTRAINT   pk_Logins    PRIMARY KEY (login_id),
12     CONSTRAINT   fk_Users     FOREIGN KEY (user) REFERENCES Users(user_id)
13 );
14
15 CREATE PROCEDURE user_logins @uid INT
16 AS
17   SELECT U.username, U.email, L.timestamp AS logindate
18     FROM Users  AS U JOIN
19          Logins AS L ON U.user_id = L.user
20     WHERE U.user_id = @uid
21 GO;
```

Figure 3.26: Mini-SQL program

Finally Figure 3.26 shows an example program in Mini-SQL as it could have existed in a web-application. Both tables contain three columns, with the various types that are included in Mini-SQL. The `email` column is the only column where null values are allowed. In the `Users` table, the primary key constraint is declared as a column constraint, while in the `Logins` table the constraint is declared as a table constraint at line 11. Furthermore, the foreign key constraint at line 12 indicates that the `user` column references a record in the `Users` table when its value matches with the `user_id` value of a record in the `Users` table.

3.2.4 Semantics of Mini-SQL

The type system of Mini-SQL contains versatile set of static consistency validations. In this section, we will explain these validations.


Numbers.sql

```
CREATE TABLE Numbers (
  num INT NOT NULL PRIMARY KEY
);

CREATE PROCEDURE pairs_until_10
AS
  SELECT * FROM Numbers AS N1
    JOIN Numbers AS N2 ON N1.num < 10
  WHERE num < 10
GO;
```

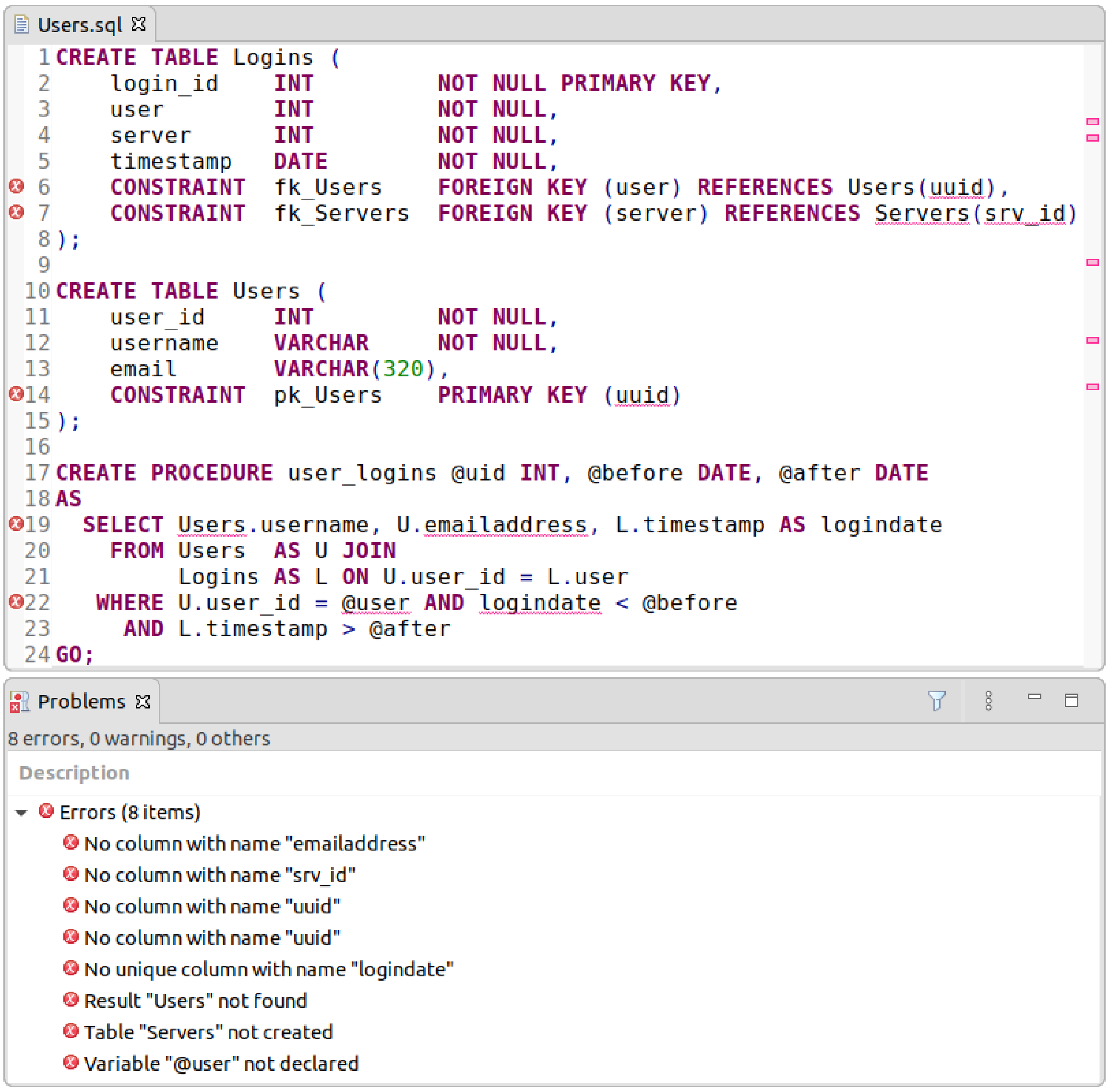
Problems

3 errors, 0 warnings, 0 others

Description
Errors (3 items)
Multiple columns with name "num"
Multiple columns with name "num"
No unique column with name "num"

Figure 3.27: Duplicate columns

Existence. Firstly, references to tables and columns in constraints should resolve. When either a table or a column does not resolve, a static error is given, as shown in Figure 3.28. Line six in this figure shows that forward references, in this case to the `Users` table, are allowed. Similar constraints hold for references to tables, columns and variables in `select` statements. However, the evaluation order of a `select` statement needs to be considered carefully to decide if an original name, or an alias should be used. First, the table selections and renames in the `from` clause are executed, after that the `join` conditions and `where` clause, and finally the column selection and renaming. For this reason, the conditions need to reference tables by their *new* name, as seen at line 19, while the columns



Users.sql

```
 1 CREATE TABLE Logins (
 2     login_id    INT         NOT NULL PRIMARY KEY,
 3     user        INT         NOT NULL,
 4     server      INT         NOT NULL,
 5     timestamp   DATE        NOT NULL,
 6     CONSTRAINT  fk_Users    FOREIGN KEY (user) REFERENCES Users(uuid),
 7     CONSTRAINT  fk_Servers  FOREIGN KEY (server) REFERENCES Servers(srv_id)
 8 );
 9
10 CREATE TABLE Users (
11     user_id     INT         NOT NULL,
12     username    VARCHAR     NOT NULL,
13     email       VARCHAR(320),
14     CONSTRAINT  pk_Users    PRIMARY KEY (uuid)
15 );
16
17 CREATE PROCEDURE user_logins @uid INT, @before DATE, @after DATE
18 AS
19   SELECT Users.username, U.emailaddress, L.timestamp AS logindate
20     FROM Users  AS U JOIN
21          Logins AS L ON U.user_id = L.user
22    WHERE U.user_id = @user AND logindate < @before
23      AND L.timestamp > @after
24 GO;
```

Problems

8 errors, 0 warnings, 0 others

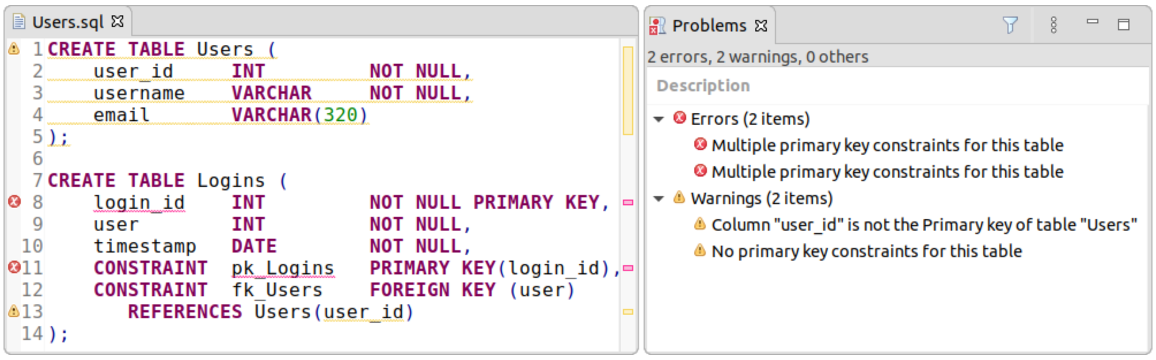
```
Description
Errors (8 items)
    No column with name "emailaddress"
    No column with name "srv_id"
    No column with name "uuid"
    No column with name "uuid"
    No unique column with name "logindate"
    Result "Users" not found
    Table "Servers" not created
    Variable "@user" not declared
```

Figure 3.28: Mini-SQL resolution errors

must be referenced by their *old* name, as shown by the reference to `L.timestamp` at line 23. Furthermore, errors are given when columns or variables cannot be found, as displayed by the references to `U.emailaddress`, `@user` and `logindate` on line 19 and 22.

Uniqueness. Secondly, names should resolve uniquely. Table names and procedure names must be globally unique, and column names, constraint names, parameter names and aliases should be unique within the scope of the table or procedure, respectively. This issue is particularly prevalent when a table is included multiple times in a result set, as demonstrated in Figure 3.27. In that case, an unqualified reference to a column can resolve to both of the result sets, and is therefore ambiguous. In such cases, a qualified reference must be used, as shown by the `N1.num` reference in the join condition.

Constraints. The type system of Mini-SQL validates whether a table has exactly one primary key. When no primary key is present, a warning on the table definition is issued. When there are multiple primary keys, errors on the primary key constraint declarations are given. Examples of these messages are shown in Figure 3.29.


Users.sql

```
CREATE TABLE Users (
    user_id     INT          NOT NULL,
    username    VARCHAR      NOT NULL,
    email       VARCHAR(320)
);

CREATE TABLE Logins (
    login_id    INT          NOT NULL PRIMARY KEY,
    user        INT          NOT NULL,
    timestamp   DATE         NOT NULL,
    CONSTRAINT  pk_Logins    PRIMARY KEY(login_id),
    CONSTRAINT  fk_Users     FOREIGN KEY (user)
        REFERENCES Users(user_id)
);
```

Problems

2 errors, 2 warnings, 0 others

Description

- Errors (2 items)
  - Multiple primary key constraints for this table
  - Multiple primary key constraints for this table
- Warnings (2 items)
  - Column "user_id" is not the Primary key of table "Users"
  - No primary key constraints for this table

Figure 3.29: Mini-SQL Primary Key errors

Additionally, Mini-SQL enforces that foreign key constraints are well-formed. Foreign keys are valid when the referencing column, and the referenced table and column exist, and have the same type. In the case of two `varchar` columns referencing each other, the referencing column must be at least as big as the referenced column. Finally, the referenced column must be the primary key of the table.

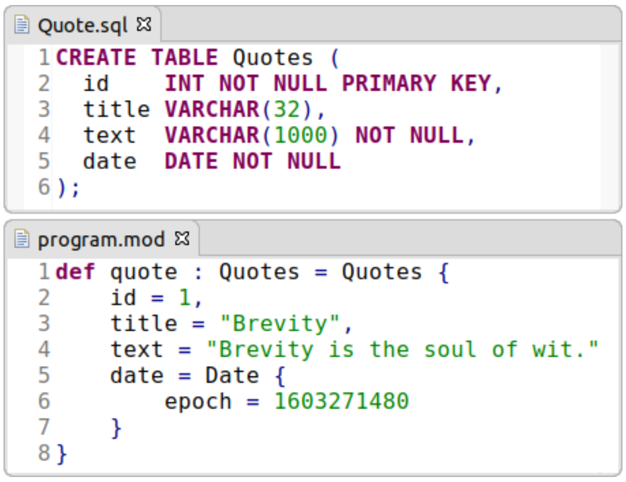
Condition Typing. Furthermore, the expressions in the join condition and the `where` clause should be well typed. These well-typedness conditions entail that only two values of the same type can be compared. When it is needed to compare values of different types, they must be explicitly converted using the `convert` function. However, when the input and the target type of a call to `convert` are equal, a warning is given. In the same way, a warning is given when `is null` or `is not null` is checked for a column with a `not null` constraint, or a literal.

Result Set Type. A `select` statement returns a Result set, which can be seen as an anonymous table without constraints. The *projection* clause determines the columns that it includes. In Mini-SQL, it is not possible to access it directly, but we will see how result sets work in the next section.

3.2.5 Integration of Mod and Mini-SQL

Now that we explained how the languages in this case study are defined, we will explain their integration. This integration contains two parts: using table definitions to create records, and calling stored procedures as if they were defined as functions. In the subsequent sections, we will discuss both integrations.

Record Creation. Firstly, it is possible to create a record using a reference to a table declaration, rather than a `record` type definition. Columns of the table definition are interpreted as record fields. An example of such a program is shown in Figure 3.30.

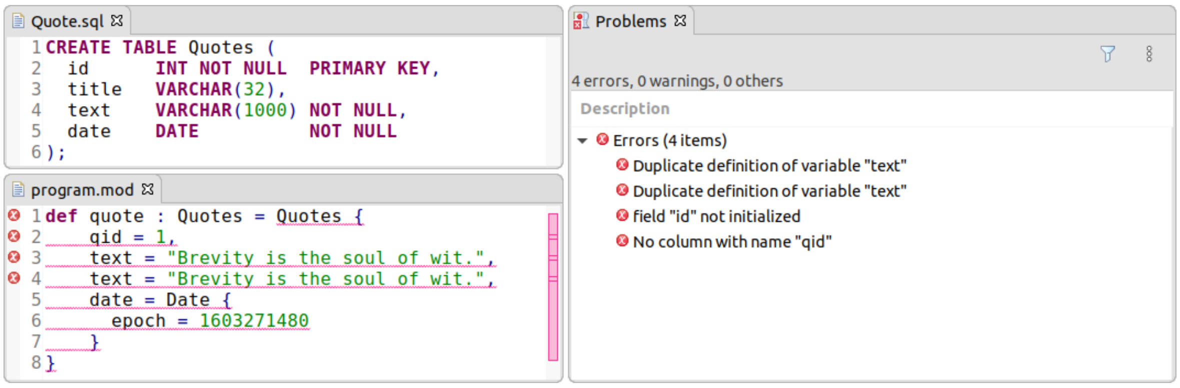

Quote.sql

```
CREATE TABLE Quotes (
  id    INT NOT NULL PRIMARY KEY,
  title VARCHAR(32),
  text  VARCHAR(1000) NOT NULL,
  date  DATE NOT NULL
);
```

program.mod

```
def quote : Quotes = Quotes {
    id = 1,
    title = "Brevity",
    text = "Brevity is the soul of wit."
    date = Date {
        epoch = 1603271480
    }
}
```

Figure 3.30: Mod – Mini-SQL example

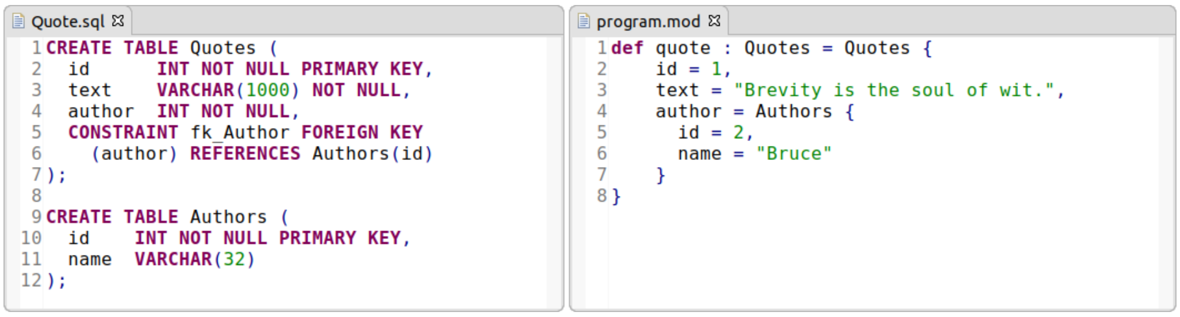


```
1 CREATE TABLE Quotes (
2   id      INT NOT NULL  PRIMARY KEY,
3   title   VARCHAR(32),
4   text    VARCHAR(1000) NOT NULL,
5   date    DATE          NOT NULL
6 );
```

```
1 def quote : Quotes = Quotes {
2     qid = 1,
3     text = "Brevity is the soul of wit.",
4     text = "Brevity is the soul of wit.",
5     date = Date {
6       epoch = 1603271480
7     }
8 }
```

```
4 errors, 0 warnings, 0 others
Description
Errors (4 items)
    Duplicate definition of variable "text"
    Duplicate definition of variable "text"
    field "id" not initialized
    No column with name "qid"
```

Figure 3.31: Record initialization errors



```
CREATE TABLE Quotes (
  id      INT NOT NULL PRIMARY KEY,
  text    VARCHAR(1000) NOT NULL,
  author  INT NOT NULL,
  CONSTRAINT fk_Author FOREIGN KEY
    (author) REFERENCES Authors(id)
);

CREATE TABLE Authors (
  id    INT NOT NULL PRIMARY KEY,
  name  VARCHAR(32)
);
```

```
def quote : Quotes = Quotes {
    id = 1,
    text = "Brevity is the soul of wit.",
    author = Authors {
      id = 2,
      name = "Bruce"
    }
}
```

Figure 3.32: Record in Foreign Key field

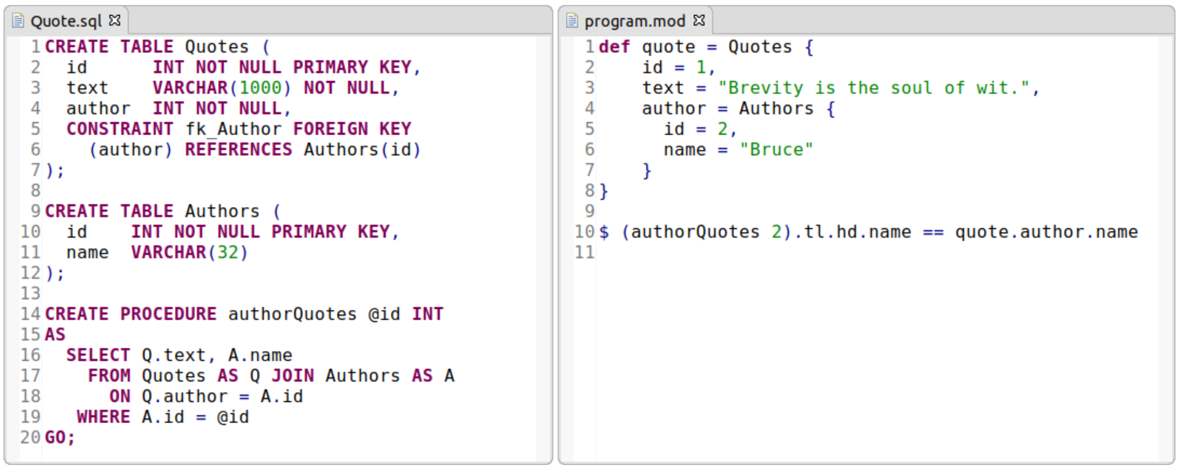
Similar to regular record initialization expressions, an error is given, as demonstrated by the errors on the `id` and `text` fields. Moreover, when an unknown column is initialized, an error is given as well, as shown by the `qid` field. However, there is a slight relaxation of these conditions: it is allowed to omit values of columns without a `not null` constraint. This is shown by the fact no `title` field is initialized in Figure 3.31, but no error is emitted for that.

Because the types of Mod and SQL do not fully align, some type mapping is needed. We have already seen the first example: mapping tables to records. Furthermore, both languages have an integer type, which therefore requires no special attention. To properly handle `varchar` types in Mod, and vice versa, we introduced string literals and types. The length constraint on `varchar` values is ignored in the mapping, because, in imperative languages, it is not possible to track string length in every situation. Finally, to handle `date` types in Mod, we introduced a built-in record type with name `Date`, and one field: `epoch : Int`. Initialization of such a field is demonstrated at lines five to seven of Figure 3.31.

Finally, columns with a foreign key constraint are handled in a special way. Instead of being initialized with the data type of the column, such a field must be initialized with a record of the type of the referenced table. For example, consider the program in Figure 3.32. Here, the `author` column is a reference to the `Authors` table, and hence it is initialized with a record of type `Authors`.

Procedure Calls. Secondly, it is possible to call stored procedures, defined in Mini-SQL, with the function application syntax of Mod. For example, Figure 3.33 shows a procedure that returns all the quotes of a particular author. In the Mod program at the right, this procedure is called, and part of its result is compared to the previously created record.

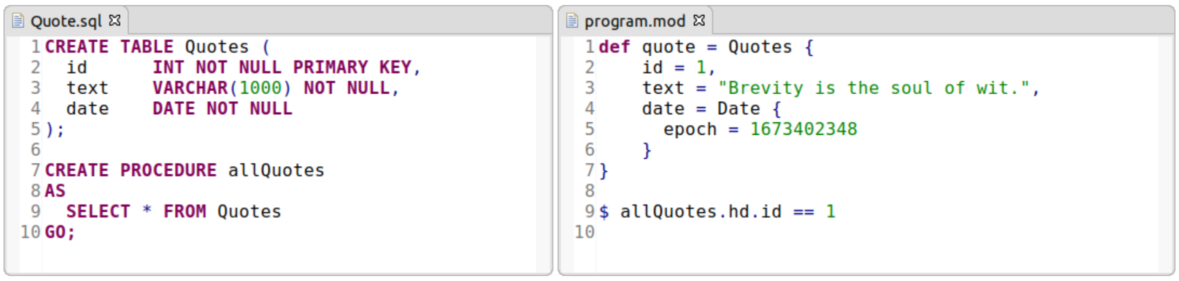
To make type-checking procedure calls work, some additional types must be mapped. First, an SQL procedure can take multiple arguments, while a Mod function can only take one. Therefore the procedure types must be curried before using them in a Mod context. This translation has the effect that procedures without parameters behave as values in Mod.



```sql
CREATE TABLE Quotes (
  id      INT NOT NULL PRIMARY KEY,
  text    VARCHAR(1000) NOT NULL,
  author  INT NOT NULL,
  CONSTRAINT fk_Author FOREIGN KEY
    (author) REFERENCES Authors(id)
);

CREATE TABLE Authors (
  id    INT NOT NULL PRIMARY KEY,
  name  VARCHAR(32)
);

CREATE PROCEDURE authorQuotes @id INT
AS
  SELECT Q.text, A.name
    FROM Quotes AS Q JOIN Authors AS A
      ON Q.author = A.id
   WHERE A.id = @id
GO;
```

```
def quote = Quotes {
    id = 1,
    text = "Brevity is the soul of wit.",
    author = Authors {
      id = 2,
      name = "Bruce"
    }
}

$ (authorQuotes 2).tl.hd.name == quote.author.name
```

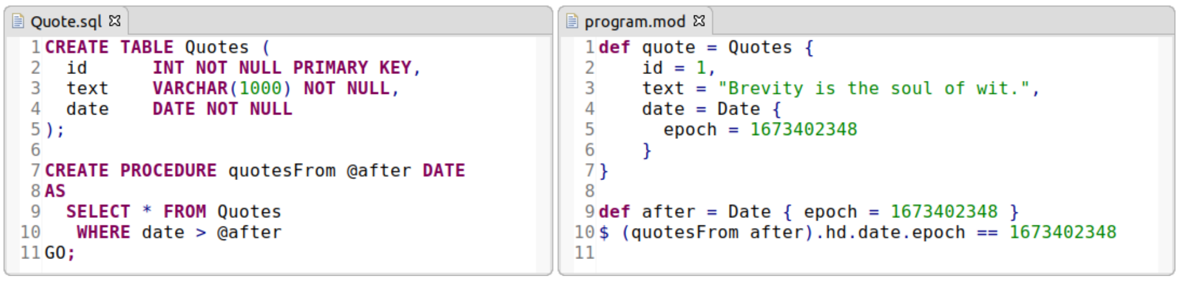
Figure 3.33: Procedure invocation



```sql
CREATE TABLE Quotes (
  id      INT NOT NULL PRIMARY KEY,
  text    VARCHAR(1000) NOT NULL,
  date    DATE NOT NULL
);

CREATE PROCEDURE allQuotes
AS
  SELECT * FROM Quotes
GO;
```

```
def quote = Quotes {
    id = 1,
    text = "Brevity is the soul of wit.",
    date = Date {
      epoch = 1673402348
    }
}

$ allQuotes.hd.id == 1
```

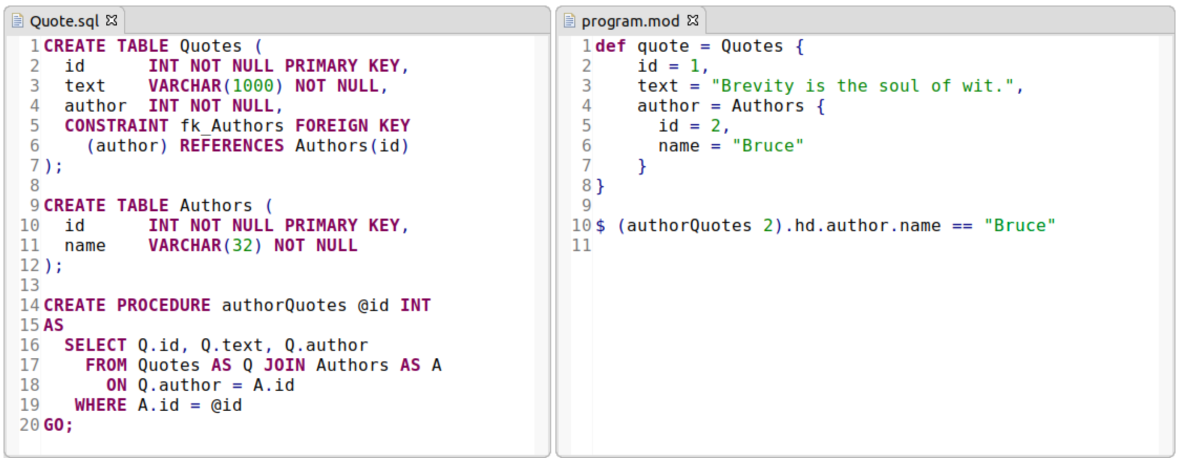
Figure 3.34: Procedure without Parameters



```sql
CREATE TABLE Quotes (
  id      INT NOT NULL PRIMARY KEY,
  text    VARCHAR(1000) NOT NULL,
  date    DATE NOT NULL
);

CREATE PROCEDURE quotesFrom @after DATE
AS
  SELECT * FROM Quotes
   WHERE date > @after
GO;
```

```
def quote = Quotes {
    id = 1,
    text = "Brevity is the soul of wit.",
    date = Date {
      epoch = 1673402348
    }
}

def after = Date { epoch = 1673402348 }
$ (quotesFrom after).hd.date.epoch == 1673402348
```

Figure 3.35: Procedure `Date` Parameter



```sql
CREATE TABLE Quotes (
  id      INT NOT NULL PRIMARY KEY,
  text    VARCHAR(1000) NOT NULL,
  author  INT NOT NULL,
  CONSTRAINT fk_Authors FOREIGN KEY
    (author) REFERENCES Authors(id)
);

CREATE TABLE Authors (
  id      INT NOT NULL PRIMARY KEY,
  name    VARCHAR(32) NOT NULL
);

CREATE PROCEDURE authorQuotes @id INT
AS
  SELECT Q.id, Q.text, Q.author
    FROM Quotes AS Q JOIN Authors AS A
      ON Q.author = A.id
   WHERE A.id = @id
GO;
```

```
def quote = Quotes {
    id = 1,
    text = "Brevity is the soul of wit.",
    author = Authors {
      id = 2,
      name = "Bruce"
    }
}

$ (authorQuotes 2).hd.author.name == "Bruce"
```

Figure 3.36: Mod – SQL: Procedure Foreign Key return

For example, in Figure 3.34, the `allQuotes` procedure can be referenced as a value at line nine of the Mod program.

Furthermore, there is no trivial way to represent SQL result sets in Mod, because Mod does not have built-in lists or other collection types. Therefore, we represent a result set as a record type. This type has two fields: `hd` for a result instance, and `tl` for the tail of the list, which contains the remaining results. Thus, the expression `tl.hd` at line ten should be interpreted as accessing the second element of the result set.

Moreover, the `Date` type we defined earlier is fully compatible with procedure parameters and return values as well. In Figure 3.35, there is a procedure that has a `date` parameter. Such a procedure can be called just by providing it a record with type `Date`. In the same way, the `epoch` field can be read from a value returned by a query.

Finally, when a column with a foreign key constraint are returned, its type is just the data type with which the column is declared, not a record representation of the table row it references. An example of this behavior is shown in Figure 3.36. The reason for this behavior is that standard SQL Semantics do not guarantee that all the fields of the referenced record are included in the result set. Hence, the fictional runtime would not be able to build this record without imposing new semantics on the query.

In conclusion, we have presented two case studies in which two languages are semantically integrated. These case studies show that, even for two rather different languages, it is still to define meaningful static semantics. However, we have not formalized the static semantics yet. An approach to do that in a modular fashion is therefore presented in chapter 4.

Chapter 4

Multi-Language Type System Definition in Statix

This chapter explains how to design composable type systems in Statix (Antwerpen, Poulsen, Rouvoet, and Visser 2018) that adhere to the criteria specified in chapter 2. First, we will introduce the Statix meta-language in section 4.1. Readers that already have some experience with Statix might want to skip this section. After that, we explain a design pattern that uses Statix to define composable type systems in section 4.2. Finally, in section 4.3 we validate this pattern by explaining how we used it to define the type systems of the case study languages we introduced in chapter 3.

4.1 Introduction to Statix

Statix (Antwerpen, Poulsen, Rouvoet, and Visser 2018) is a specification language with precisely defined declarative and operational semantics. Type systems are expressed in a notation that is close to formal inference rules. Name resolution is expressed using *scope graphs* (Néron, Tolmach, Visser, and Wachsmuth 2015), which provide a powerful, language-agnostic approach to encode name binding patterns. To execute specifications, a constraint solving approach is applied. In this section, we provide the background knowledge about Statix that is required to understand the approach introduced in section 4.2 and the examples. In this section, we focus on the *declarative* interpretation of Statix. Some relevant operational aspects will be discussed in chapter 5, where appropriate. For a more in-depth discussion on the operational semantics of Statix, readers are referred to the work of Rouvoet, Antwerpen, Poulsen, Krebbers, and Visser (2020).

4.1.1 Typing Rules in Statix

Type systems in Statix are specified using inference rules. Typing rules are generally written as a proposition under a bar (the *conclusion*) and a series of propositions above the bar (the *premises*). Such rules indicate that one can infer that the conclusion holds when all the premises hold. Usually, a typing judgment (which is a possible proposition) is written as $\Gamma \vdash t : T$, where t is a term from an AST, T is the type of this term, and Γ is a set of assumptions. In this thesis, when the assumptions are not relevant for a particular rule, the Γ symbol is omitted, but assumed to be implicitly threaded to all premises. Assuming the reader is familiar with such typing judgments, we will introduce the notation that Statix provides. As a running example, we use the small expression language in Figure 4.1

We must declare the signature of this syntax in Statix as well. This can be done with the same notation as in Mini-STR, which was introduced in subsection 3.1.3. For the grammar shown in Figure 4.1, a signature is provided in Figure 4.2.

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{int} \mid \mathit{str} \mid \mathit{id} \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} < \mathit{exp} \\
 & \mid & \textbf{let}\ \mathit{id} = \mathit{exp}\ \textbf{in}\ \mathit{exp} \mid \textbf{if}\ \mathit{exp}\ \textbf{then}\ \mathit{exp}\ \textbf{else}\ \mathit{exp} \\
\\
\mathit{Type} & ::= & \textbf{Int} \mid \textbf{String} \mid \textbf{Bool}
\end{array}
$$

Figure 4.1: Expression language grammar

```
signature
  sorts Exp constructors
    Int  : int        -> Exp
    Str  : string     -> Exp
    Id   : string     -> Exp
    Plus : Exp * Exp -> Exp
    Lt   : Exp * Exp -> Exp
    Let  : string * Exp * Exp -> Exp
    If   : Exp * Exp * Exp    -> Exp

  sorts TYPE constructors
    INT  : TYPE
    STR  : TYPE
    BOOL : TYPE
```

Figure 4.2: Expression signature

Moreover, we must explicitly declare the signature of typing predicates. In this example, we have only one functional predicate, shown in Figure 4.3[1]. This rule has the type `Exp -> TYPE`, which can be read as a regular function type: it computes a `TYPE` for an `Exp`.

```
rules
  typeOfExp: Exp -> TYPE
```

Figure 4.3: Rule signature

As a first example, consider the typing rule in Figure 4.4, which states that for any integer literal, it can immediately be derived that its type is *Int*, because there are no premises. Its Statix counterpart on the right encodes the same constraint: when `typeOfExp` is validated for an `Int(_)` literal, it is assigned the type `INT()`, without validating any premise.

$$\text{T-Int}\ \frac{}{\vdash \mathit{int} : \mathit{Int}}$$

```
typeOfExp(Int(_)) = INT().
```

Figure 4.4: Integer typing

However, most of the typing rules have premises that need to be satisfied before its conclusion holds. As an example, consider the rule to type plus-expressions in Figure 4.5. This rule states that a plus-expression has type *Int* if both its arguments have type *Int*. In Statix, these premises can be encoded with the `:-` operator, which corresponds to the inference bar of regular typing rules. In the right part of the figure, an equivalent rule is given in Statix notation. This rule indicates that the type of a `Plus` expression is `Int`, provided that its left and right argument have type `Int`.

[1]Alternatively, a predicate can be declared in a `signature constraints` section. Although this is more idiomatic, it seems not to be used anywhere. Joining established practice, declarations in the `rules` section are

$$\text{T-Plus}\ \frac{\vdash e_1 : \mathit{Int} \quad \vdash e_2 : \mathit{Int}}{\vdash e_1 + e_2 : \mathit{Int}}$$

```
typeOfExp(Plus(e1, e2)) = INT() :-
  typeOfExp(e1) == INT(),
  typeOfExp(e2) == INT().
```

Figure 4.5: Plus typing

This figure introduces two more constraint expressions that Statix provides. First, the `==` operator validates syntactical equality of terms. Second, the `,` operator denotes constraint conjunction.

Next, we consider the less-than operator from our expression language. We want this operator to type-check on any argument type, provided that the arguments on the left-hand side and on the right-hand side have the same type. An example of such a rule is provided at the left of Figure 4.6. In its Statix counterpart, we see that a new variable `T` is introduced with the `{T}` notation. In Statix, any meta-variable that is not part of the head of a rule must be introduced in this way. This variable represents the type the left and the right sub-expression have. By using a single variable, we encode that these types should be equal.

$$\text{T-Lt}\ \frac{\vdash e_1 : T \quad \vdash e_2 : T}{\vdash e_1 < e_2 : \mathit{Bool}}$$

```
typeOfExp(Lt(e1, e2)) = BOOL() :- {T}
  typeOfExp(e1) == T,
  typeOfExp(e2) == T.
```

Figure 4.6: Less-than typing

A slightly more complicated example is provided by the typing rule for if-then-else expressions, shown in Figure 4.7. In addition to checking that both branches have the same type, this type is returned as the type of the complete expression as well.

$$\text{T-If}\ \frac{\vdash c : \mathit{Bool} \quad \vdash e_1 : T \quad \vdash e_2 : T}{\vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : T}$$

```
typeOfExp(If(c, e1, e2)) = T :-
  typeOfExp(c) == BOOL(),
  typeOfExp(e1) == T,
  typeOfExp(e2) == T.
```

Figure 4.7: If-then-else typing

In this rule, we see that the type of the condition should be `Bool`, and the types of the expression in the then-branch should be equal to the type of the else-branch. On top of that, the type that those expressions have is returned by the predicate.

Finally, there are representatives of the $\top$ and $\bot$ constraints in Statix, which are shown in Figure 4.8. The `true` constraint, which unconditionally succeeds, is shown at the left side. This constraint can equivalently be written as `top().`, because having a premise that always holds is equivalent to having no premise at all. Similarly, the `false` constraint, which unconditionally fails, is shown at the right side of the figure.

```
top() :- true.
```

```
bottom(msg) :-
  false
  | error $[Error: [msg]] @msg.
```

Figure 4.8: Top and Bottom constraints

used throughout this thesis, including its case studies.

Additionally, the last line of the `bottom` constraint shows another Statix feature: the possibility to provide custom error messages to a constraint. When a constraint with a custom error message template fails, the Statix runtime will create an error message from the template. This template is a text literal in which variables can be inserted by wrapping them in square brackets, such as the `[msg]` part. The position of the message is determined by the last part of the message specification: a term preceded with an `@` symbol, which is assumed to be a node in the AST that is type-checked.

4.1.2 Introduction to Scope Graphs

Until now, we have discussed the typing rules that are independent of the outer context. That is, the type of the expressions discussed so far can be decided by traversing the subtree of its AST node. However, it is not possible to type variables and let-expressions in this way, because let-expressions need to pass the type of the newly introduced variable down to their subtrees, and variables need to lookup their type.

In traditional approaches, there is a set of assumptions Γ, sometimes denoted as 'environment', which consists of identifier – type pairs (denoted as $x : T$). These assumptions are passed down with the typing judgment. Using these assumptions, variable bindings can be introduced and queried. Typing rules for let-expressions and variables that use this principle are depicted in Figure 4.9. The rule for let expressions extends the environment with a binding for the introduced variable, which are looked up by the T-Var rule. Therefore, variables can only be typed if its identifier is introduced earlier by a let-expression.

$$\text{T-Let}\ \frac{\Gamma \vdash i : T_x \quad \Gamma, x : T_x \vdash e : T}{\Gamma \vdash \text{let}\, x = i \,\text{in}\, e : T} \qquad\qquad \text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Figure 4.9: Let and Var typing

However, Statix takes a different approach. Name binding patterns are encoded using *scope graphs* (Antwerpen, Poulsen, Rouvoet, and Visser 2018; Antwerpen, Néron, Tolmach, Visser, and Wachsmuth 2016; Néron, Tolmach, Visser, and Wachsmuth 2015). In this section, we give a high-level introduction to scope graphs. For an in-depth, formal treatment of the semantics of scope graphs in Statix, we refer to the work of Antwerpen, Poulsen, Rouvoet, and Visser (2018).

Scope graphs consist of three components:

- *Scopes* represent "a region in a program that behaves uniformly with respect to name resolution". These scopes are modeled as nodes in the graph. In text, we denote them with a sharp (e.g. #1). Its graph representation is shown in Figure 4.10a.
- *Labeled, directed edges* model visibility relations between scopes. For example, #1 $\xrightarrow{\mathsf{P}}$ #2 indicates that the graph contains an edge from #1 to #2 with label P. Its graph representation can be seen in Figure 4.10b.
- *Declarations* model a datum under a relation symbol in a scope. Its textual notation is #1 $\xrightarrow{rel}\blacksquare$ *d*, meaning datum *d* is declared in scope #1 under relation *rel*. Its pictorial equivalent is shown in Figure 4.10c. Note that in the pictures, we often use a colon, instead of a bracketed pair, to make the notation less cluttered. For example, instead of #1 $\xrightarrow{rel}\blacksquare$ `(x, INT)`, we write #1 $\xrightarrow{rel}\blacksquare$ `x : INT`.
- In Statix, it is possible to pass around scopes, just as regular data terms. This is used to model user-defined composite data types, such as records, modules or classes. The scope contains declarations for the members of the types. The textual notation for scopes in terms is `MOD(#1)` $\rightarrow$ #1, meaning that the #1 in `MOD(#1)` refers to scope #1. In a picture, we denote associated scopes as displayed in Figure 4.10d.

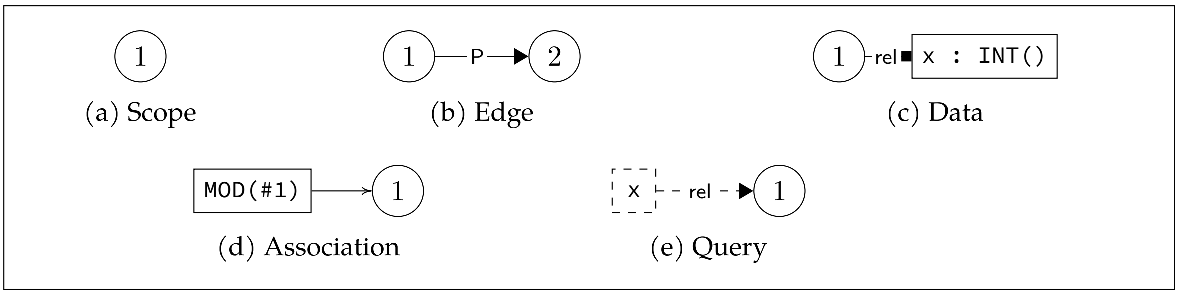


Figure 4.10: Scope Graph notation

Finally, information can be retrieved from a scope graph using *queries*. A query from a scope traverses the scope graph to find datums that match its conditions. A query has several parameters:

- The relation to query. Only datums declared under this relation will be returned by the query.
- Path well-formedness condition: a regular expression of labels that specifies which paths are well-formed. Only datums that are reached through a path whose edge labels are in the language described by the regular expression are included in the query result.
- A match predicate, taking a single datum as input. Only datums that satisfy this predicate will be included in the query result. Its default value is `true`, meaning that all datums under the specified relation that satisfy the path well-formedness condition are returned.
- Result comparison parameters:
  - A strict partial order on paths, described by less-than relations on labels. This relation defines a prefix-order on paths.
  - A 'shadow' predicate, which takes two datums as input. Its default value is `false`, meaning that no shadowing is performed.

  When the result set contains multiple datums, and it holds for a datum d that there exists another datum d' which path is strictly smaller that d, and the shadow predicate holds, then d is removed from the result set.

The result of a query is a list of path – datum tuples. In scope graphs, we depict queries as shown in Figure 4.10e. The value in the box represents an equality predicate with respect to that value, and the name on the arrow is the queried relation. Usually we do not depict the other parameters. These should be clear from the text, or even irrelevant at all. Especially shadowing is not widely used in this thesis.

To get a better understanding of scope graphs, we consider a small program in our expression language, shown in Figure 4.11. This example consists of two let-expressions, which introduce two variables, `x` and `y`, which are added in the inner expression. Its corresponding scope graph is shown on the right side.

In this scope graph, scope #1 corresponds to the global scope, scope #2 to the `in` expression of the outer `let` (lines two and three), and scope #3 corresponds to the `in` expression of the inner `let` (line three). The outer let introduces the variable `x` with type `INT()` in scope #2. That ensures that all expressions in scope #2 and its subscopes can see this variable. The same holds for the `y` variable in the inner let expression. In the graph, variable references, with their resolution paths are shown as well, with the correspondence between references and queries indicated by colors.

```
let x = 3 in
  let y = x + 2 in
    x + y
```

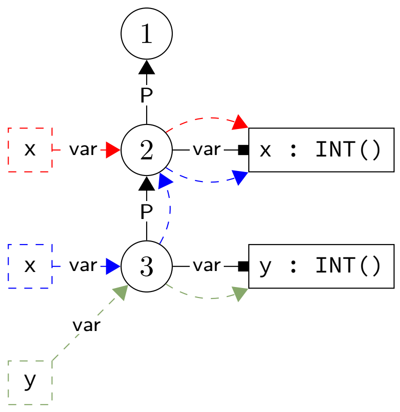


Figure 4.11: Reference resolution in scope graphs

```
signature
  relations
    var: string -> TYPE

  name-resolution
    labels P

rules
  typeOfExp : scope * Exp -> TYPE
```

Figure 4.12: Relations and Labels

Now, we discuss how these rules can be encoded in Statix. Because of the versatility of the scope graph model, these rules tend to be slightly more complicated. First, we must declare the relations and labels we use in our specification. In Figure 4.12, we declare a relation `var`, which maps identifiers to their types, and a label `P`. In this specification, #1 $\xrightarrow{P}$ #2 should be interpreted as "scope #2 is a lexical parent of #1". Furthermore, we adapt the `typeOfExp` predicate to accept a `scope` parameter, which represents the scope in which the expression is typed.

```
typeOfExp(s, Let(x, i, b)) = T :- {Tx s_body}
  typeOfExp(s, i) == Tx,
  new s_body, s_body -P-> s,
  !var[x, Tx] in s_body,
  typeOfExp(s_body, b) == T.
```

Figure 4.13: Let typing rule

Having provided these signatures, we show the Statix rule for let-expressions in Figure 4.13. Firstly, there is a new `s` parameter, which corresponds to the scope in which this expression is evaluated. At line 2, the type of the variable initialization expression is bound to the variable `Tx`. Then, at line 3, a new scope is introduced with the `new` keyword. The `s_body -P-> s` constraint should be read as $\#s_{body} \xrightarrow{P} \#s$. It indicates that `s_body` has an edge with label `P` to `s`. At line, 4, the variable is declared in newly introduced scope by the `!var[x, Tx] in s_body` constraint, which represents $\#s_{body} \xrightarrow{var}$ `(x : Tx)`. Finally, the type of the let body is evaluated in the newly introduced scope. The type of the body is then returned by the predicate as type of the complete expression.

```
typeOfExp(s, Var(x)) = T :-
  query var
    filter P*
      and { x' :- x == x' }
      min $ < P
      and true
       in s |-> [(_, (_, T))].
```

Figure 4.14: Var typing

Finally, the typing rule of a variable reference is shown in Figure 4.14. The type of a variable is looked up using a query. At line 2, we see that the queried relation is `var`, which corresponds to the declarations that are made in the rule for let expressions. Second, the path well-formedness predicate is `P*`, which indicates that a reference may resolve to a declaration in the scope in which the query started, or any parent scope. Third, the `{ x' :- x == x' }` parameter is an anonymous predicate, which is evaluated for every variable declaration `x'` that the query encounters. It is satisfied when it has a variable name that is equal to the reference `x`. This condition ensures that a variable only resolve to declarations with the same name. For example, when we would leave it out, the reference `x` in Figure 4.11 would resolve to the `y` declaration, which is obviously incorrect. Fourth, the `min $ < P` query parameter defines a path order. Here, the $ symbol represents the end of a path. Therefore, `$ < P` indicates shorter paths are preferred over longer paths, which corresponds to the regular notion of shadowing. Fifth, the `true` part indicate that all results shadow each other. This is correct, because the data well-formedness predicate already ensures that all results have the same name. Together, the last two parameters ensure that only the closest declaration of a particular name is included in the query result. Finally, the `in s` term indicates that scope `s` is the scope the resolution must start from. A Statix query returns a list of path – datum pairs, where a datum is an n-ary tuple containing the arguments to the relation. The pattern the query result is matched to indicates that a single result (one outer tuple in the list) is expected. The first position of this tuple, which is the path, is ignored using a wildcard. Similarly, we ignore the name of the declaration using the wildcard in the inner tuple. Eventually, we match the type of the declaration that is returned by the query with the `T` variable. This variable is then returned as the result of the rule.

4.1.3 Modules

The last Statix feature we discuss here are *modules*. Until now, we have only shown fragments of Statix specifications. In reality, these specifications must be organized in modules. A part of the Statix specification for the expression language we used so far is shown in Figure 4.15.

```
module type

signature
  sorts TYPE constructors
    INT : TYPE
```

```
module exp
imports type

signature
  sorts Exp

rules
  typeOfExp: scope * Exp -> TYPE
```

```
module arith
imports type
imports exp

signature
  constructors
    Int  : int -> Exp
    Plus : Exp * Exp -> Exp

rules
  typeOfExp(_, Int(_)) = INT().
  typeOfExp(s, Plus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

Figure 4.15: Statix modules

In this figure, we see three modules: `type`, `exp` and `arith`. The `exp` module imports the `type` using an `import` statement. Similarly, the `arith` module imports both other modules. Importing makes all declarations from the imported module visible. Moreover, the extensions to the `Exp` sort and `typeOfExp` rules in the `arith` module show that sorts and constraints can be extended. When executing a specification, the runtime will resolve modules, and combine the partial specifications each module provides into a complete specification. This specification can then be used to type-check actual programs.

4.1.4 Type-checking complete Programs

Using these constraints, complete type system specifications for a wide variety of languages can be expressed. Moreover, these specifications can also be executed for a particular program. In order to ensure that the solver knows how to use a specification to type-check a project[2], two special predicates must be provided. First, a `projectOk: scope` predicate must be present. This predicate, which is most often used to declare built-in types, is validated once for a project. Second, a `fileOk: scope * Start` predicate must be declared. This predicate is instantiated once for every file in the project, in order to ensure that all source files are well-typed. In Figure 4.16, the constraint set that represents this behavior is shown.

Worth mentioning is that the same scope is passed to all the predicates. In this way, the type-checker for every file can declare its top-level definitions in the global scope. Since queries of other files can resolve to this same global scope, these declarations become reachable by type-checkers of other files as well. This enables name resolution queries to cross file boundaries.

```
new s,
projectOk(s),
fileOk(s, file1),
fileOk(s, file2),
...
fileOk(s, fileN),
```

Figure 4.16: Project Type-checking

4.1.5 Summary

In summary, Statix allows modular encoding of formal and declarative type-checking rules. In order to model context-sensitive constraints, such as name resolution, scope graphs are used. In such graphs, scopes are modeled as nodes, in which declarations can be added. Using labeled edges and highly customizable queries, a broad range of name binding patterns can be encoded.

Now that we gave a short introduction to type system specification in Statix, we will show how the Statix features we explained so far can be leveraged to define composable, interoperating type systems.

4.2 Defining Composable Type Systems in Statix

In this section, we introduce the Shared Concept Interface (SCI) pattern, which is a design pattern for Statix specifications that can be used to define mutually agnostic, composable type systems. This design pattern is based on the property that queries in scope graphs need not be aware of the reason why their results are present in the scope graph. That is, scope graph declarations and queries are loosely coupled. Hence parts of a Statix specification that relate to each other at runtime, can be agnostic at compile time.

In short, this design pattern assumes that *analysis for multiple languages* is executed in *the same global scope*. When that is the case, a language can expose its declarations in the shared global scope, which makes them reachable by queries from other languages. To ensure that the relations and terms of different languages match, those are extracted in *interface modules*, which should be used in the specifications of the individual languages. In the remainder of this section, we explain the role of the interface module and the way languages interoperate with it.

4.2.1 Interface Module

First, the semantic concepts that should be shared with other type systems are encoded in a separate Statix interface modules. These concepts include:

- Sort and constructor declarations for the data terms that can be communicated.
- Relations that can be used to expose this data.

[2]In this thesis, we restrict to multi-file analysis only.

- The labels that manage the visibility of declarations.

Labels should be included in the interface as well, because Statix uses qualified names for labels internally. When labels would be defined in the individual languages, a language can-not define a path regex that queries the part of the scope graph defined by another language, because it can not reference its labels.

As an example of such a shared module, consider the interface of the integration between Mini-SDF and Mini-STR in Figure 4.17. This interface encodes a representation of sorts and constructors by respectively the `SORT` and `CONS` declarations, and their corresponding construc-tors. A `SORT` corresponds to a sort declaration. A `TYPE` is a sort with a multiplicity. The `CONS` constructor has three arguments. The first argument is the sort for which this constructor is a production. The second argument represents the arity of the constructor. Making the arity explicit makes queries to a constructor declaration easier. The third argument represents the types of the arguments that need to be supplied to this constructor. The `sort` and `cons` rela-tions allow declaring actual instances in a scope graph. Finally, the `P` and `I` labels (for lexical parent and import, respectively) can be used to encode a resolution policy.

```
module abstract-signature

signature
  sorts SORT constructors
    SORT    : string -> SORT

  sorts TYPE constructors
    SINGLE  : SORT   -> TYPE
    OPT     : SORT   -> TYPE
    ITER    : SORT   -> TYPE
    STAR    : SORT   -> TYPE

  sorts CONS constructors
    CONS    : SORT * int * list(TYPE) -> CONS

  relations
    sort: string -> SORT
    cons: string -> CONS

  name-resolution
    labels P I
```

Figure 4.17: Abstract signature interface (summarized). The complete interface is included in section A.1

4.2.2 Language Module

After having established an interface, a language implementer can use this interface to ex-pose definitions his language can share, and query for definitions that might be declared by another specification implementing the interface. For example, consider the code from the Mini-SDF specification in Figure 4.18 and Figure 4.19.

4.2.3 Conventions

Usage of this design pattern relies on strict adherence to particular conventions by the users of the interface. In this section, we discuss two groups conventions that accompany an inter-face: the usage of relations and the usage of labels (i.e. maintaining a particular structure in the scope graph).

```
sortOk: scope * SortDecl
sortOk(s, SortDecl(n)) :-
  !sort[n, SORT(n)] in s.
```

Figure 4.18: Interface usage (declaration)

```
typeOfSortTerm : scope * SortTerm -> TYPE
typeOfSortTerm(s, Ref(SortRef(n))) = SINGLE(S) :- {S}
  query sort
     filter P* I*
        and { n' :- n == n' }
        min $ < P, $ < I, P < I and true
         in s |-> [(_, (_, S))].
```

Figure 4.19: Interface usage (resolution)

Relation usage. When multiple languages are using this module to define queries, the results of these queries can resolve to declarations made by another language. However, such interfaces should be used in a correct way. When invalid data is provided in a declaration, a different language may interpret it incorrectly, or is not able to use it at all, breaking the interoperability.

To encourage proper use of the interface, we recommend to provide predicates for declaring and querying the relations in the interface. As an example, consider an excerpt from the integration between Mini-SDF and Mini-STR in Figure 4.20, that encode how to declare and query constructors.

```
rules
  declareSort: scope * string
  resolveSort: scope * string -> SORT

  declareSort(s, n) :-
    !sort[n, SORT(n)] in s.

  resolveSort(s, n) = S :-
    query sort
       filter P* I*
          and { n' :- n == n' }
          min $ < P, $ < I, P < I and false
           in s |-> [(_, (_, S)) | _].
```

Figure 4.20: Examples of wrapper predicates

Besides encouraging correct use of the interface, these wrapper interfaces can increase evolvability as well. For example, this interface does not encode the uniqueness constraints on sorts and constructors yet. Imagine that the language designer wants to implement these constraints in the interface, to ensure that any user of the interface can safely assume these constraints hold. He adapts the interface with the changes depicted in Figure 4.21.

Now, all type systems that manually created instances of the `SORT` constructor are broken, because the constructor signature changed. However, the signature of the predicates did not change, and hence all specifications that used the `declareSort` predicate do not need adaption, or even recompilation. Unfortunately, Statix has no features to hide constructors or relations to enforce this pattern, which would allow interface designers to carefully expose an interface to the type systems implementing it.

Label usage. The second convention that must be adhered to by the language designers is about the structure of the scope graph. As an example, imagine a situation where a language implementer needs an intermediate scope for imports (e.g. to declare sort renamings). When he implements it in the way depicted in Figure 4.22, the usual path regex (`P* I*`) cannot

```
signature
  sorts SORT constructors
    SORT : string * scope -> SORT

rules
  declareSort(s, n) :- {id}
    new id,
    !sort[n, SORT(n, id)] in s,
    query sort
       filter P* I*
          and { n' :- n == n' }
           in s |-> [_].
```

Figure 4.21: Interface adaption

```
rules
  import(s, n, rn) :- {s_mod s_int}
    resolveMod(s, n) == MOD(s_mod),
    new s_int,
    s -I-> s_int, s_int -P-> s_mod,
    renameSorts(s_int, rn).
```

Figure 4.22: Incorrect label usage

resolve imports, because there will be a `P` label *after* the `I` label, which does not match this regex.

Now, this particular language specification can use `P* (IP)*` as well-formedness condition in his queries, and his specification would give correct results. However, this way of using the interface breaks interoperability with other languages that use `P* I*` as their well-formedness condition. Hence, an invariant on the graph structure is an essential part of the interface. Unfortunately, there is no trivial way to force language implementers to create a well-formed subgraph, or even validate whether they do so.

4.2.4 Publish – Subscribe

When comparing this pattern to other design patterns, its similarity to the publish-subscribe design pattern in Object-Oriented programming is evident. In this analogy, values are 'published' by creating declarations in the scope graph, and 'received' by being returned in the result of a query. The interface acts as a broker, using predicates to prescribe publication strategies and using other predicates that manage the receiving of values.

Now that we have established our strategy to define multi-lingual type systems, we will explain how we used this pattern in the case studies we introduced in chapter 3.

4.3 Case studies

In this section, we return to the case studies we introduced informally in chapter 3, and discuss in more detail how we implemented their interaction using the pattern introduced in the previous section. For each study, we summarize the concepts the languages share, and discuss how these are encoded in an interface. After that, we discuss how the individual languages use the interface module. Then we illustrate the interoperation with examples, and explain in-depth how its works internally. Finally, we summarize the case study, and name the key findings.

4.3.1 Mini-SDF + Mini-STR

In the Mini-SDF – Mini-STR case study, we implemented an integration where Mini-STR rewrite rules can be validated against signatures that are derived from a Mini-SDF syntax specification.

Modules. In general, the Mini-SDF and Mini-STR languages share the concept of abstract signatures and the concept of modules. Because the resolution of signatures from other modules depends on the module import mechanism, we first explain the module interface, and after that the representation of abstract signatures.

In Figure 4.24, a summarized module interface is shown. This interface provides a `MOD` sort and constructor. The constructor takes a scope argument, in which all the declarations of that module are added, and a `ROLE` argument, which indicates whether this module supplies or consumes a signature. By exposing its scope, other modules can construct edges to this module, to model importing. Second, a relation `mod` is provided, which allows declaring and querying modules. To declare modules, the `declareMod` predicate is provided. This predicate takes a scope, which will usually be the global scope, a module name and a module scope, and creates a corresponding definition. Finally, the `import` predicate defines how to import a module. It takes a scope, a module name and a role, and then queries the `mod` relation to find a corresponding module declaration. From that declaration, it extracts the scope of the module, and constructs a corresponding edge to it. Using this edge, queries from $\#s$ can reach declarations in $\#s_{mod}$. Finally, the `itemsOk` predicate, of which the definition is discussed in section 4.3.1, validates that no duplicate sort and constructor names are imported.

Note that the resolution policy for modules is determined by the `filter P*` part of the query. In this case, nested modules are not supported. As the example in section 4.2.3 shows, it is important to have a consistent resolution policy between languages. Therefore it is very important for interface designers to anticipate the expectations of their future users, and manage them correctly.

In Figure 4.25, we see how this interface is used by Mini-SDF. An excerpt of its syntax is shown in the `signature` section. A Mini-SDF file consists of a module with a name and a list of sections. One type of sections is the `imports` section, which contains a list of module references. When a module is typed, a declaration for it is made using the `declareMod` predicate from the interface. Furthermore, when a module is referenced in an `imports` section, the `import` predicate is used to import it into the module scope.

The interface is used in a similar manner in Mini-STR. However, modules are declared with the `CONSUME()` role, and a wildcard is passed as to the role parameter of the `import` predicate, indicating that all module roles are allowed.

In order to understand this mechanism, consider the example provided in Figure 4.26. In this example, two modules `parent` and `child`, are provided, where the latter imports the former. In the corresponding scope graph on the right, we see that each of these modules corresponds to a declaration in the global scope. Moreover, we see how the import edge is established. First, a query from #3 reaches the declaration of the `parent` module. From this declaration, the scope that corresponds to this module is extracted. Finally, the import edge, labeled with `I`, is created.

Signatures. The second component of the interface module that enables interoperability between Mini-SDF and Mini-STR is the description of abstract signatures. For reference, a complete interface for abstract signatures in shown in Figure 4.27. This interface contains the regular sorts, constructors, relations and predicates for the definition of signatures.

The `scope` arguments to the `SORT` and `CONS` constructors do not contain any declarations. Instead, the fact that they are unique is used for name duplication checks, as discussed in section 4.2.3.

The TYPE sort corresponds with a position in a constructor. It can either be a single production, an optional production or a list of productions. List types build with the ITER constructor must contain at least one element, while list types build with the STAR constructor may be empty.

Finally, the second parameter to the CONS constructor denotes the arity of the constructor. Mini-STR allows overloading by arity, and therefore queries to constructor declarations need to be parameterized with a particular arity. To make these queries easier, we include the arity explicitly in the constructor. As an example, the definition of resolveCons is provided in Figure 4.23. In the match predicate, the CONS constructor is deconstructed, and it is validated that the provided name and arity of the constructor match with the declaration.

```
resolveCons(s, n, a) = C :-
  query cons
     filter P* I*
        and { c :- c == (n, CONS(_, a, _)) }
        min $ < P, $ < I, P < I
        and true
         in s |-> [(_, (_, C))].
```

Figure 4.23: Constructor resolution

The declareCons predicate always ensures that this parameter matches the length of the parameter list. In the graphs in this section, we will omit the scopes that serve as identity as well as the SORT and SINGLE constructors for brevity.

```
module modules

signature
  sorts ROLE constructors
    SUPPLY  : ROLE
    CONSUME : ROLE

  sorts MOD constructors
    MOD: scope * ROLE -> MOD

  relations
    mod: string -> MOD

  name-resolution
    labels P I

rules

  declareMod: scope * string * scope * ROLE
  import: scope * string * ROLE
  itemsOk: string * scope * scope

  declareMod(s, n, s_mod, R) :-
    !mod[n, MOD(s_mod, R)] in s.

  import(s, n, R) :- {s_mod}
    query mod
       filter P*
          and { n' :- n' == n }
           in s |-> [(_, (_, MOD(s_mod, R))],
    s -I-> s_mod,
    itemsOk(n, s, s_mod).
```

Figure 4.24: Module interface

```
module minisdf

imports modules

signature
  sorts Start constructors
    Mod: string * list(Section) -> Start

  sorts Section constructors
    Import: list(ModRef) -> Section

  sorts ModRef constructors
    ModRef: string -> ModRef

rules

  fileOk: scope * Start

  sectionsOk: scope * list(Section)
  sectionOk: scope * Section

  importsOk: scope * list(ModRef)
  importOk: scope * ModRef

  fileOk(s, Mod(n, ss)) :- {s_mod}
    new s_mod,
    s_mod -P-> s,
    declareMod(s, n, s_mod, SUPPLY()),
    sectionsOk(s_mod, ss).

  importOk(s, ModRef(n)) :-
    import(s, n, SUPPLY()).
```

Figure 4.25: Module interface usage

parent.msdf
```
module parent
```

child.mstr
```
module child

imports parent
```

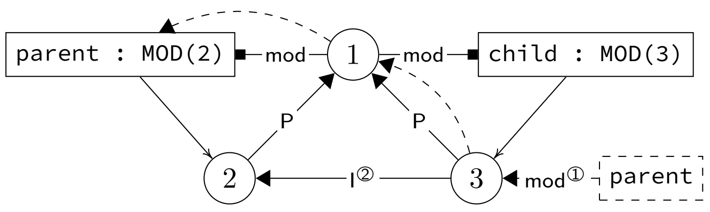


Figure 4.26: Import Example

```
module abstract-signature

signature
  sorts SORT constructors
    SORT    : string * scope -> SORT

  sorts TYPE constructors
    SINGLE  : SORT -> TYPE
    OPT     : SORT -> TYPE
    ITER    : SORT -> TYPE
    STAR    : SORT -> TYPE

  sorts CONS constructors
    CONS    : SORT * int * list(TYPE) * scope -> CONS

  relations
    sort: string -> SORT
    cons: string * CONS

rules

  declareSort: scope * string
  resolveSort: scope * string -> SORT

  declareCons: scope * string * list(TYPE)
  resolveCons: scope * string * int -> CONS
```

Figure 4.27: Abstract signature interface

Mini-SDF uses this interface at three places.

- To create declarations of sorts in the scope graph, analogous to the declarations of modules in Figure 4.25.
- To resolve the sort references in productions, as seen in the `typeOfType` predicate in Figure 4.28. Again, for brevity, only one instance of `typeOfType` is shown, but similar variants of the rule exist for the other constructors of the `Type` sort.
- To create declarations of constructors, which is done by the `prodOk` rule. The implementation of `typesOfSymbols` is left out. It filters away the literals, and translates `Type(t)` terms to their `TYPE` using `typeOfType`.

Mini-STR uses the interface in the same way, however, it resolves constructors for the validation of pattern expressions as well. This is shown in the `typeOfPattern` rule in Figure 4.29.

```
module minisdf/productions

imports abstract-signature

signature
  sorts Prod constructors
    Prod : string * string * list(Symbol)

  sorts Symbol constructors
    Lit  : string -> Symbol
    Type : Type -> Symbol

  sorts Type constructors
    Single : string -> Type
    Opt    : string -> Type
    Iter   : string -> Type
    Star   : string -> Type

rules
  prodOk: scope * Prod
  typesOfSymbols: scope * list(Symbol) -> list(Type)
  typeOfType: scope * Type -> TYPE

  typeOfType(s, Single(n) = SINGLE(T) :-
    resolveSort(s, n) == T.

  prodOk(s, Prod(sn, cn, smbls)) :- {T}
    typesOfSymbols(s, smbls) == T,
    declareCons(s, sn, cn, T).
```

Figure 4.28: Mini-SDF interface usage

```
module ministr/patterns

rules
  typesOfPatterns: scope * list(Pattern) -> list(TYPE)
  typeOfPattern: scope * Pattern -> TYPE

  typeOfPattern(s, Constr(n, args)) = SINGLE(S) :- {a}
    arity(args) == a,
    resolveCons(s, n, a) == CONS(S, _, Tp, _),
    typesOfPatterns(s, args) == Tp.
```

Figure 4.29: Mini-STR interface usage

In this rule, a constructor declaration is resolved, based on the name and arity. After that, the list of expected arguments (`Tp`) is compared to the actual types of the arguments.

Finally, we shortly address the duplicate import checking demonstrated in Figure 3.20. In Figure 4.24, we mentioned an `itemsOk` predicate that was supposed to do the duplicate checking. In Figure 4.30, we show a part of the implementation of this procedure for sort declarations. First, recall that `SORT` was defined as `SORT: string * scope`, where the `string` is the name, and the `scope` served as an identifier. The `itemsOk` contains two queries: the first one resolves all sorts visible in the *imported module*, which are hence imported to the importing module *via that module*. The second query resolves all sorts visible in the *importing module*. The results of these queries are compared pairwise using the `sortsOk` and `sortOk` predicates.

```
module import/overlap

rules
  itemsOk: string * scope * scope
  sortsOk: string * list(SORT) * list(SORT)

  sortOk(m, SORT(n, id1), SORT(n, id2)) :-
    // When name is equal, but id is different, give error
    id1 == id2 | error $[Duplicate import of Sort [s]]@n.

  itemsOk(m, s, s_mod) :- {S_all, S_mod}
    query sort
      filter P* I*
          in s |-> S_all,
    query sort
      filter P* I*
          in s_mod |-> S_mod,
    sortsOk(m, S_all, S_mod).
```

Figure 4.30: Duplicate import checking

The `sortOk` rule indicates that two sorts with the same name are accepted only if they have the same identifer. When a sort has the same name *and identifier*, it may be imported via multiple paths, but is not declared twice. However, when identifiers are different we create an error, because they indicate a duplicate declaration. Finally, pairs with different names are accepted in the second rule. The `sortsOk` predicate, of which the implementation is not shown, iterates the `S_mod` values. For each value, it filters sort declaration with the same name from `S_all`, and compares the with the `sortOk` predicate. Because there is no string comparison operator in Statix, these lists can not be ordered. Therefore, the filter operation is linear in the size of `S_all`. Hence the runtime complexity of this operation is at least $O(|S_{all}| \cdot |S_{mod}|)$. However, it is difficult to measure concrete running times for this constraint, which would be helpful for assessing its impact.

Example. Finally, we show a more complete example, featuring almost all language features we explained until now. In Figure 4.31, the sources for this project are shown. This source shows a syntax definition of a language with integer constants, plus expressions and brackets. In the `norm` module, a rewrite rule is defined that removes all unnecessary brackets from an expression.

A reduced scope graph for this project is shown in Figure 4.32. In this scope graph, $\#s_l$ is the scope of the `lit` module, $\#s_p$ is the scope of the `plus` module and $\#s_n$ is the scope of the `norm` module. The scopes $\#s_m$ and $\#s_b$ correspond with the match and build term of the first rule, respectively. The scope with the ellipsis signifies the scopes of the other rules, which are left out for brevity. For the same reason, the global scope, the module declarations and import queries are not shown. For all remaining relations, sample declarations and queries are shown, with the colors in the source code corresponding to the references.

Evaluation. In this case study, we found that the SCI pattern is well-suited to define composable type systems. Using just three shared relations and corresponding data types, it was possible to set up a highly integrated pair of languages. All the checks on the imports transferred to the multi-language semantics of imports without need for adaption.

A feature that was not very easy to implement in a neat manner was the restriction that Mini-SDF modules may not import Mini-STR modules. By assigning roles to the modules,

```
module lit

sorts Expr Term Lit

context-free syntax
  Term.Lit     = <<Lit>>
  Term.Bracket = <(<Expr>)>
```

```
module plus

imports lit

context-free syntax
  Expr.Plus = <<Term> + <Expr>>
  Expr.Term = <<Term>>
```

```
module norm

imports plus

rules
  norm: Plus(Bracket(Term(l)), r) -> <norm> Plus(l, r)
  norm: Plus(l, Term(Bracket(r))) -> <norm> Plus(l, r)
  norm: e -> e
```

Figure 4.31: Small Mini-SDF – Mini-STR project

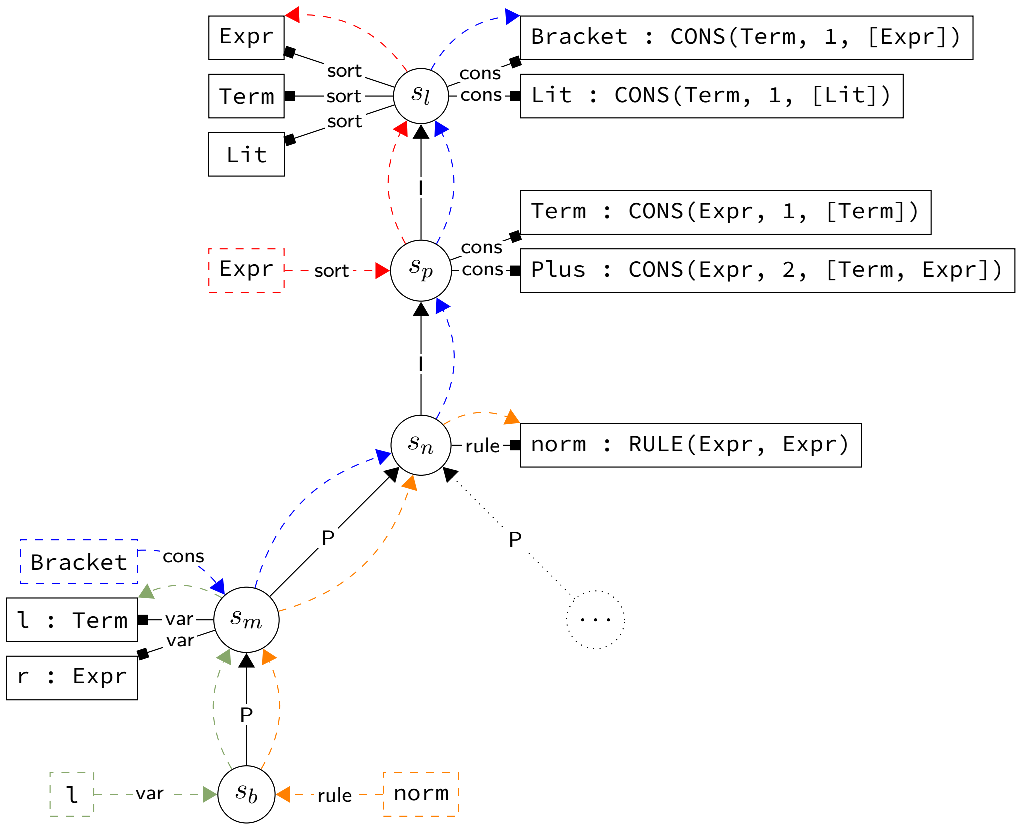


Figure 4.32: Scope graph of Figure 4.31

we could implement this check, but it is a rather ad-hoc feature that crept into the interface solely for this purpose. The core issue that is involved here is that it is not possible to define constraint that involve constructors for multiple languages, because there is no point in the process where the compilation of Statix specification can refer to both of these. One can think of a more idiomatic way to model this behavior by defining an `importOk : MOD * MOD` predicate, which can be implemented by all involved languages. However, as we explain in subsection 6.1.4, this pattern opens possibilities for inconsistent specifications, and is therefore prevented.

4.3.2 Mod + Mini-SQL

In this section we discuss the integration of the Mod language and Mini-SQL, which was introduced in section 3.2. This section has a similar structure as the previous one. First, we introduce the interface that is used to implement this case study. After that, we explain how this interface is used in both languages. Then, we discuss an example that shows all the ways the languages can interact. Finally, we evaluate our findings of this case study.

Interface. First, we describe the interface that manages the interaction between the two languages in this case study, which is shown in Figure 4.33. This interface describes relational data structures (Codd 1970). The primitive types are `INT` and `DATE`, which represent integer numbers and datetimes, respectively. The `VARCHAR` data type has an `int` as argument, which indicates its maximum length. The `FK` type represents foreign keys, which are references to other tables. Its first argument is the type of the column of the referencing table, and the second argument is the referenced table. The `TABLE` type represents a table. Its scope argument contains declarations of the columns, using the `var` relation, and the constraints on the table. The `RESULT` type, which represents query result sets, has a similar structure. However, as we explain when we discuss the integration with Mod, it has a different interpretation, and is hence given a different constructor.

```
module tables

signature

  sorts TYPE constructors
    INT       : TYPE
    VARCHAR   : int -> TYPE
    DATE      : TYPE
    FK        : TYPE * TYPE -> TYPE
    TABLE     : scope -> TYPE
    RESULT    : scope -> TYPE

rules

  declareTable: scope * string * scope
  resolveTable: scope * string -> scope

  declareVar: scope * string * scope
  resolveVar: scope * string -> scope
```

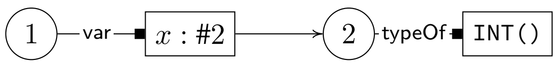
Figure 4.33: Relational Data Interface

```
signature

  relations
    var    : string -> scope
    type   : string -> scope
    typeOf : -> TYPE

rules

  withType: TYPE -> scope
  typeOf: scope -> TYPE

  withType(T) = s :-
    new s, !typeOf[T] in s.

  typeOf(s) = T :-
    query typeOf
      filter e
      in s |-> [(_, T)].
```

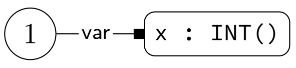
Figure 4.34: Relational Data Interface (2)

In contrast to the Mini-SDF – Mini-STR case study, this interface uses *scopes to represent types*. That is, the type that is returned by the various typing relations and predicates is not a data term of sort `TYPE`, but rather a scope, which contains a relation `typeOf` that contains the actual type. The `withType` and `typeOf` predicates can be used to convert between these representations. This representation was present in the Mod specification for historical reasons. In order to be as less invasive as possible, we maintained it. Because this encoding is used, the `declareTable` and `declareVar` take a `scope` as their third input parameter, instead of a `TYPE`. Similarly, the `resolveTable` and `resolveVar` return a `scope` that represents the type.

In Figure 4.35a, a scope graph fragment of a declaration of a variable `x` with type `Int` is shown. Because this notation is rather verbose, we will use the more compact notation with rounded corners, shown in Figure 4.35b, in the remainder of the section to denote types that are encoded with such an intermediate scope.



(a) Full



(b) Compact

Figure 4.35: Scopes representing types

```
signature

  sorts CNULL constructors
    CNULL    : CNULL
    CNOTNULL : CNULL

  relations
    nullable : CNULL
```

Figure 4.36: Nullability

```
module procedures

signature
  constructors
    PROC: list(TYPE) * TYPE -> TYPE

rules
  declareProc: scope * string * list(TYPE) * TYPE
```

Figure 4.37: Stored Procedures

Albeit not introduced for that reason, the intermediate scope is used to add additional data to a declaration. For example, column types have a `nullable` relation (shown in Figure 4.36), and table types can have constraint relations, which are not included in the interface.

Finally, the interface exposes a PROC type, which represents stored procedures. This part of the interface is depicted in Figure 4.37.The first argument represents the list of parameter types, while the second argument is the RESULT type that the procedure returns. This interface has a `declareProc` predicate, but no `resolveProc`. The reason for this is that procedures are in the same namespace as columns. Therefore procedures can be resolved with the `resolveVar` predicate. This requirement is mainly imposed by Mod, because in Mod functions are just regular variables.

Mini-SQL Integration. This interface is used by Mini-SQL at three places. First, for a `create table` statement, a new scope is created. In this scope, the columns are declared with the `declareVar` predicate. After that, this scope is passed to the `declareTable` predicate. Second, the `from` clause uses `resolveTable` to find the tables that are included in the result set. Third, the `select` clause joins all tables included in the `from` clause, and uses `resolveVar` to find the types of the columns that are included in the result set. Fourth, the `create procedure` statement uses the `declareProc` predicate to declare a procedure. Mini-SQL has no scoping mechanism. That is, tables and procedures are always declared in the global scope.

```
module types

imports tables

signature
  constructors
    STRING : TYPE
    REC    : scope -> TYPE
    FUN    : TYPE * TYPE -> TYPE
```

Figure 4.38: Mod Types

Mod Integration. The integration of the interface with the Mod language is more complicated. This increased complexity has two reasons. First, we adapted an already existing language definition. Second, the interface does match the relational model more closely than the data model Mod uses.

Initially, we tried to adapt the predicates in Mod to work with the newly introduced type constructors where appropriate. However, we found that this required rather pervasive changes to the whole type system specification. Moreover, this approach regularly resulted in stuckness in the `typeEq`, `subtype` and `lub` predicates, which Mod uses to implement its subtyping rules.

To mitigate these problems, we choose to translate the types from the interface to types that were already present in Mod. For this translation, a `strict : TYPE -> TYPE` predicate is introduced. This predicate is used at every place where a type is returned from a query. The translations that this predicate performs are:

- The `VARCHAR(_)` type is translated to `STRING()`. This looses the information regarding the maximum length of the string, which cannot be tracked in Mod anyway.
- The foreign key types are mapped to the type that *is referenced*. This mechanism ensure that tables with references can be instantiated with nested record expressions, as shown in Figure 3.32.
- `DATE()` types are translated to a record type that has a `epoch : INT()` field. This translation resembles a type provided by fictional runtime integration of Mod and SQL, which could translate SQL dates to this format.
- A `RESULT(s)` type is translated to a record type that has two fields. The `hd` field contains a record type that contains all the columns of the result set. The `tl` recursively points to the translated type. In this way, this type represents the result set as a list.
- Finally, the `PROC(Tp, T)` type is translated to a curried `FUN(Tin, T)` type.

Because the `TABLE(s)` type is almost analogous to the `REC(s)`, we have integrated this type in the type system of Mod using our initial approach. Therefore, the `strict` predicate does not translate `TABLE` types. We have deliberately chosen to maintain both types, and not replace `REC` with `TABLE`. When we would have done so, Mod would expose its top-level record type definitions. This would allow Mini-SQL to define foreign key references to Mod record types, which is certainly undesired behavior.

Example. Finally, we consider an example, depicted in Figure 4.40, which illustrates the concepts we have discussed so far.

First, we see two table definitions, `Quotes` and `Authors`, with a foreign key reference from the former to the latter. In the scope graph in Figure 4.41, we see the declarations of these tables, with most of their fields. In the `author` column, the foreign key type is reflected with the `FK(INT(), TABLE(#2))` type, which references #2, the scope of the `Authors` table. As explained earlier, this reference information is lost when this field is selected in a query. Therefore, the `author` field in #4, which represents the result set of the `select` statement, has type `INT()`.

However, in the Mod program on the right, the assignment expression to the `author` fields type-checks differently. It is assigned the value of the `kant` variable, which has type `TABLE(#2)`, retrieved by the query to the `Authors` table declaration, shown in blue. This type-checks correctly, because the type of the author field is transformed using the `strict` predicate, introduced in Figure 4.39. As can be seen there, this predicate extracts the second parameter from an `FK` type, which is indeed `TABLE(#2)`.

Second, in gray, we see the built-in declaration of the `Date` record type, which is different from the `DATE()` type from the interface. This type declaration is equivalent to the type that is created by the `strict` predicate call on an `DATE()` type. Therefore, the assignment to the `bdate` field in the first record type-checks correctly.

Third, the accesses to the fields of the `kant` variable work similar. For the `id` field, this translation is trivial, because Mini-SQL and Mod use the same `INT()` type. However, for the `text` and `bdate` fields, the transformation from `VARCHAR(1000)` and `DATE()` to their Mod counterparts is performed again. Hence these expressions type-check against the explicit type annotations on the `id`, `name` and `date` definitions.

```
module type-translation

rules
  strict : TYPE -> TYPE

  strict(VARCHAR(_)) = STRING().
  strict(FK(_, T)) = strict(T).

  strict(DATE()) = REC(s_date) :-
    new s_date,
    !var["epoch", withType(INT())] in s_date.

  strict(RESULT(s)) = T :-
    new s_rec,
    T == REC(s_rec),
    !var["hd", withType(REC(s))] in s_rec,
    !var["tl", withType(T)] in s_rec.

  strict(PROC([], R)) = strict(R).
  strict(PROC([H|T], R)) = FUN(Hs, Ts) :-
    Hs == strict(H),
    Ts == strict(PROC(T, R)).

  strict(T) = T.
```

Figure 4.39: Type Translation

Fourth, we discuss the call to the `quotes` procedure, marked in red. First, the corresponding query in the scope graph shows that the reference resolves to the `stored procedure` declaration. Note that this query resolves because the procedure declaration is in the `var` namespace.

Again, this reference is transformed by the `strict` predicate. In this case, it results in the type depicted in Figure 4.42. In this picture `REC(#1)` corresponds to the translated `DATE()` type, which is structurally the same as the built-in `Date` type we discussed above. Furthermore, `REC(#2)` is a linked-list like record, as created by evaluating `strict` on `RESULT(#3)`, the result type of the `quotes` procedure. By transforming, the input arguments match the types of the variables the procedure is called with, and in this way the function application expressions type-checks. The `result` variable does not have an explicit type, because a procedure result type cannot be referenced by name. However, the final expression shows it has the type `REC(#2)`, which would be expected, based on the signature of the (transformed) procedure.

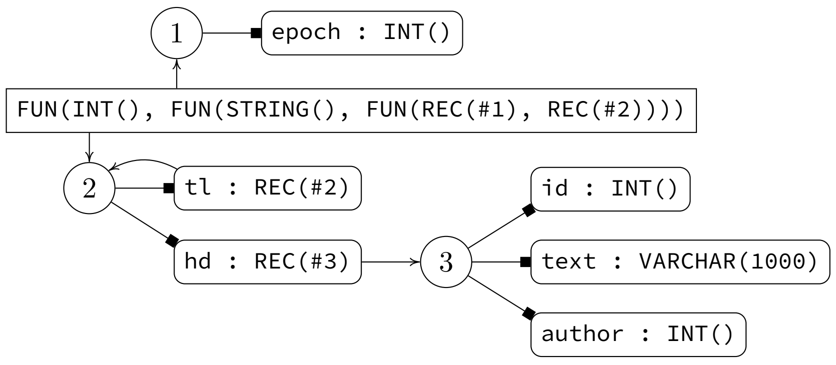


Figure 4.42: Transformed Procedure Type

```
CREATE TABLE Quotes (
  id       INT NOT NULL PRIMARY KEY,
  text     VARCHAR(1000),
  author   INT NOT NULL,
  CONSTRAINT fk_Authors FOREIGN KEY
    (author) REFERENCES Authors(id)
);

CREATE TABLE Authors (
  id       INT NOT NULL PRIMARY KEY,
  name     VARCHAR(32) NOT NULL,
  bdate    DATE
);

CREATE PROCEDURE quotes
    @id INT, @name VARCHAR, @bdate DATE
AS
  SELECT Q.id, Q.text, Q.author
    FROM Quotes AS Q JOIN Authors AS A
      ON Q.author = A.id
         AND A.name = @name
         AND A.bdate = @bdate
   WHERE A.id = @id
GO;
```

```
def kant = Authors {
  id = 1,
  name = "Immanuel Kant",
  bdate = Date {
    epoch = 1604419336
  }
};

def science = Quotes {
  id = 1,
  text = "Science is organized knowledge.",
  author = kant
};

def wisdom = Quotes {
  id = 2,
  text = "Wisdom is organized life.",
  author = kant
};

def id : Int = kant.id;
def name : String = kant.name;
def date : Date = kant.bdate;

def result = quotes id name date;

$ result.hd.id == 1
```

Figure 4.40: Mod – Mini-SQL Integration example

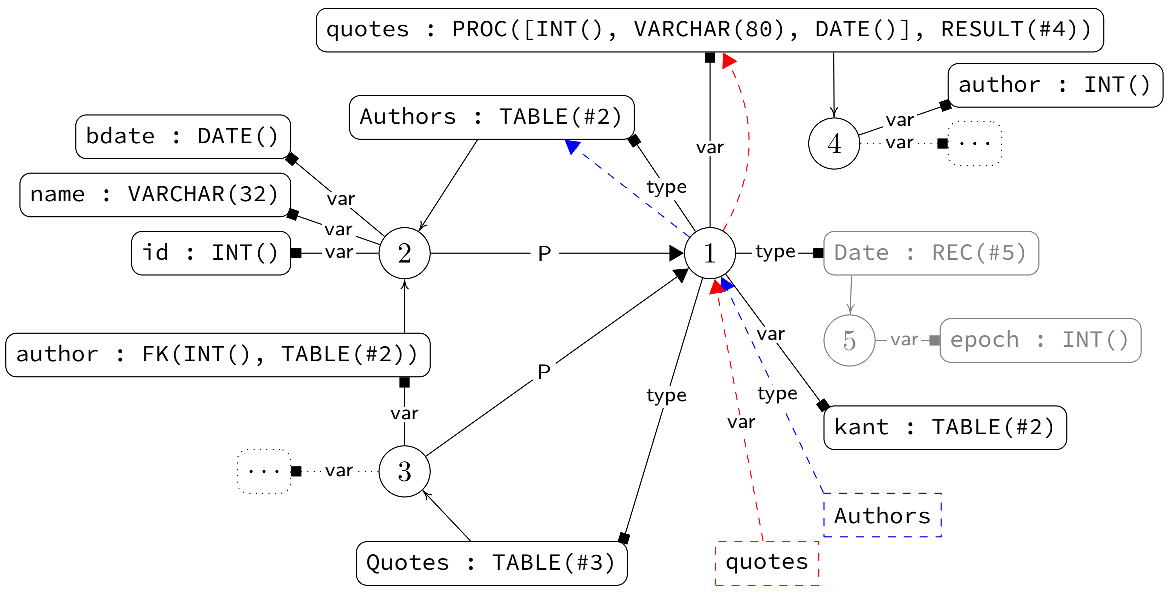


Figure 4.41: Scope Graph of Figure 4.40

Evaluation. In this case study, we found that the Shared Concept Interface (SCI) even scales to languages that do not share a common, or very similar, runtime data model. Moreover, in this case, it is possible to adapt an existing type system to use an interface with only moderate effort. We needed to lift some concepts into the interface, and we needed to define a translation between SQL types and Mod types.

However, there were some features that were not easy to handle neatly. First, the domain mismatch between relational data and the Mod type system gave rise to difficulties and namespacing. In Mod, functions are regular variables with a `FUN` type, while in Mini-SQL procedures are in a separate namespace. In general, there are two possibilities to solve this mismatch. First, the interface can define these concepts in the same namespace (under the same relation). In that case, the language with multiple namespaces must use a customized match predicate in the `filter` clause of its queries, that retains only declarations with an eligible type. On the other hand, the interface can use different namespaces for the respective declarations. The language with only a single namespace then needs to query both relations, and combine the results. In general, we do not recommend this latter approach, because combining results can be tedious, and shadowing rules cannot be implemented using the `min` clause. In this case study, we took the former approach as well. Therefore, columns and stored procedures are both declared using the `var` relation.

Second, different data models usually come with different naming. In the interface, it is needed to choose one particular name, which makes the interface somewhat counter-intuitive to read from the perspective of the language whose natural naming scheme is not chosen.

Third, different data models can come with different naming *conventions*. In this case study, SQL databases usually have a plural name, while record type names in Mod are singular. In the example we choose to have plural names in Mod, because in that context a developer will usually be more aware of the fact that he is integrating with SQL. Another solution to this impedance mismatch is having a language feature in Mod that allows renaming types.

Finally, we learned that creating declarations in the global scope, or scopes that can be reachable from the scope using the interface, need careful consideration. As we show in subsection 6.1.4, the Statix implementation guarantees that it is not possible to accidentally expose definitions of relations that are not defined in the interface. However, sometimes a declaration that can be expressed in terms defined by the interface must still remain hidden. This observation gave rise to the distinction between `TABLE` and `REC` types, which are structurally equivalent, but distinct in visibility.

To conclude this chapter, we have explained how type systems can be specified with cross-language analysis as a primary concern. Furthermore, we have shown how this pattern works in practical examples. However, defining a type system this way is one thing, but actually executing it is another. Therefore we continue with a discussion on the implementation of a runtime that supports executing composed type systems in chapter 5.

Chapter 5

Implementation in Spoofax 3

In this chapter, we describe the implementation of a multi-language runtime that can execute composed type systems, defined according to the design pattern discussed in chapter 4. In order to do that, we first introduce the Statix compiler, solver and runtime in section 5.1. Subsequently, we introduce the Spoofax 3 language engineering framework, in which we integrated our runtime. After discussing these preliminaries, we give a detailed overview of the architecture of the multi-language meta-component. Finally, shortly summarize the key points in the conclusion.

Throughout this chapter, we will use UML class diagrams to display parts of the design of the systems we discuss. Several projects that we discuss in this chapter generate implementations for data classes. In our UML diagrams, we will represent those as if they are regular data classes, using field notation instead of accessor methods. In this way, the distinction between data and operations remains clear. Furthermore, we sometimes shorten class names to keep diagrams concise.

5.1 Background: Architecture of the Statix Implementation

In order to integrate multi-language analysis in Spoofax 3, we need to reuse parts of the existing infrastructure. In general, the Statix infrastructure in Spoofax 2 consists of three components. First, there is a *compiler*, which transforms Statix source files in compiled modules. Secondly, there is a *solver*, implemented in Java, which is called using Stratego primitives. Before calling these primitives, the *Statix runtime* transforms the file AST and the compiled specification into a form that the solver accepts.

As we will discuss in subsection 5.2.4, the Statix compiler is already integrated in the Spoofax 3 compiler, and hence we reuse it. We will not use the Statix runtime, but rather create an adapted reimplementation in Spoofax 3. From this new runtime, we call the existing Statix solver. In order to understand our usage of the existing APIs and the argument that our implementation is correct, we will now discuss the architecture of these three components as they exist in Spoofax 2.

5.1.1 Compiler

The Statix compiler transforms a Statix file into a compiled module. These compiled modules include the rules from their original source file, and the labels and relations used in the module. Moreover, scope extensions (discussed below) are recorded in the module. The compiled modules are stored in the `src-gen/statix` directory of the project.

During compilation, the well-formedness of the specification is checked. The Statix compiler validates whether the constraints are referenced with the right argument counts and types, whether terms are well-formed, and labels and relations are used correctly. Additionally, two more advanced types of analysis are performed: Pattern overlap checking and scope extension permission checking. Moreover, in the generated modules, all names are substituted with fully qualified names. As will turn out in subsection 6.1.4, all of these features are important to consider when validating the correctness of our approach.

Qualified Names. During transformation, all names are qualified explicitly. This allows modules to be merged without name collisions. For declarations of labels and relations, the name of the module in which they are declared is prepended. Similarly, references to labels, relations and constraints are prepended with the name of the module in which the declaration they resolve to was made.

However, the qualification of rules is handled a bit differently. As we demonstrated in subsection 4.1.3, it is possible to define rules for a constraint that is declared in another module. To ensure that those rules are treated together as one constraint, each rule name is qualified with the name of the module in which the predicate it contributes to is declared.

Overlapping Patterns. The Statix solver is implemented using constraint simplification and solving. User-defined constraints, such as `typeOfExp`, are simplified to built-in constraints, such as equality constraints, scope graph declarations and queries.

During constraint solving, the solver can be in a state where multiple rules for a constraint can be applied. For example, the rule that should be chosen to simplify `typeOfExp(s, exp)` depends on the value of `exp`, which might still be unknown. Instead of trying all possible applicable rules (backtracking), the solver will delay the constraint until its arguments are known precisely enough to unambiguously choose a rule that is to be used for simplification.

In order to choose a rule unambiguously, Statix uses a comparison of patterns to define a partial ordering on rules. The term comparison can give four results:

- *Equal* when the two terms match on exactly the same set of terms. For example, `_` and `T` compare equal, and likewise do `MOD(T)` and `MOD(T')` compare equal.
- *Less than* when the left pattern is more specific than the right pattern. That is, when the set of terms that matches on the left pattern is a strict subset of the set of terms that matches on the right pattern. For example, `MOD(T)` is less then `T`.
- *Greater than* when the right term is more specific than the left term.
- *Incomparable* otherwise.

For rules, Statix applies this comparison from left to right on its input patterns. When two patterns are equal or incomparable, comparison proceeds with the next pair of input patterns. Otherwise, the rule with the most specific pattern is preferred over the rule with the more general pattern. For example, in Figure 5.1, the rule `numericType(s, INT())` is preferred over `numericType(s, T)`, because the first arguments compare equal, and for the second pair of arguments, `INT()` is more specific than `T`.

However, incomparable patterns need some more consideration. When two patterns are incomparable, it can not immediately be derived that their input sets are disjoint. When a variable occurs multiple times in a pattern (i.e. the pattern is non-linear), all occurrences

```
rules
  numericType(s, INT()).
  numericType(s, T) :- ....
```

Figure 5.1: Ordered rules

```
rules
  simplify(UNION(T, T)) = simplify(T).
  simplify(UNION(T, INT()) = simplify(T).
```

Figure 5.2: Rules without order

after the first one do not compare as less than a more specific term anymore, because they are restricted by an equality constraint on the first occurrence.

For example, `UNION(T, T)` is not comparable with `UNION(T, INT())`, because, depending on the value of the first `T` of the first pattern, the second `T` might not match `INT()` anymore. Therefore, it can not be derived that `T` is greater than `INT()`. When we consider the sets of terms that match on each of these patterns, we see why this is the case. For example, the pattern `UNION(BOOL(), BOOL())` only matches the first, `UNION(BOOL(), INT())` only matches the second, and `UNION(INT(), INT())` matches both. Hence, neither of these patterns has a strictly smaller input set, and therefore they cannot be compared.

```
rules
  equal(_).
  equal(_) :- false.

  lub(_, NULL()).
  lub(T, T).
```

Figure 5.3: More overlap

However, this poses a problem for the solver. When a constraint is called with inputs that match the patterns of multiple rules, but there is no ordering for these rules, the solver cannot decide which rule it should use. For example, there is no way to choose the correct rule for the constraint `simplify(UNION(INT(), INT())` in the specification of Figure 5.2, because the rules cannot be compared.

In order to avoid this situation, Statix statically forbids patterns where rules have overlapping input patterns, but can not be compared. In addition to the example in Figure 5.2, Figure 5.3 shows some more examples that are rejected. First, the `equal` rules are rejected, because the input patterns of both rules are exactly equal. Second, the `lub` rule is rejected because there are inputs that match only the first rule or only the second rule, and there are inputs that match both rules. Moreover, this example demonstrates that the overlap detection for non-linear patterns extends to cases where the variables occur in different argument positions.

Scope Extensions. In Statix specifications, the solving of typing constraints and name resolution queries can be interleaved. Therefore, Statix needs to resolve queries in scope graphs that are not yet complete. However, it needs to ensure that the results returned by a query cannot be invalidated when the scope graph is expanded. For an in-depth treatment of the way Statix implements sound scheduling of queries we refer to the work of Antwerpen, Poulsen, Rouvoet, and Visser (2018) and Rouvoet, Antwerpen, Poulsen, Krebbers, and Visser (2020). For our discussion it suffices to make two observations.

First, a scope may only be extended with a declaration or a edge to another scope when it is either freshly created in the constraint, or passed to it from a direct argument. It is not allowed to extend scopes that are returned from other predicates, or that are obtained by matching other terms. When a rule violates this constraint, an error is emitted. To implement this analysis, Statix first transitively records for each constraint argument with type `scope` with which labels it might be extended. After that, it checks each constraint call argument with type `scope`. When the called constraint does extend the scope, but the scope is not freshly created by the calling constraint, or passed as a direct argument, it emits an error.

Second, the information collected during the first part of the analysis (with which labels a `scope` argument might be extended) is included in the compiled module. The solver uses this information to schedule queries correctly. In particular, the solver will only return the result of a query for a relation `rel` when all the scope it passes through will not be extended with declarations for that relations, nor with edges to subgraphs that are valid according to the path well-formedness predicate.

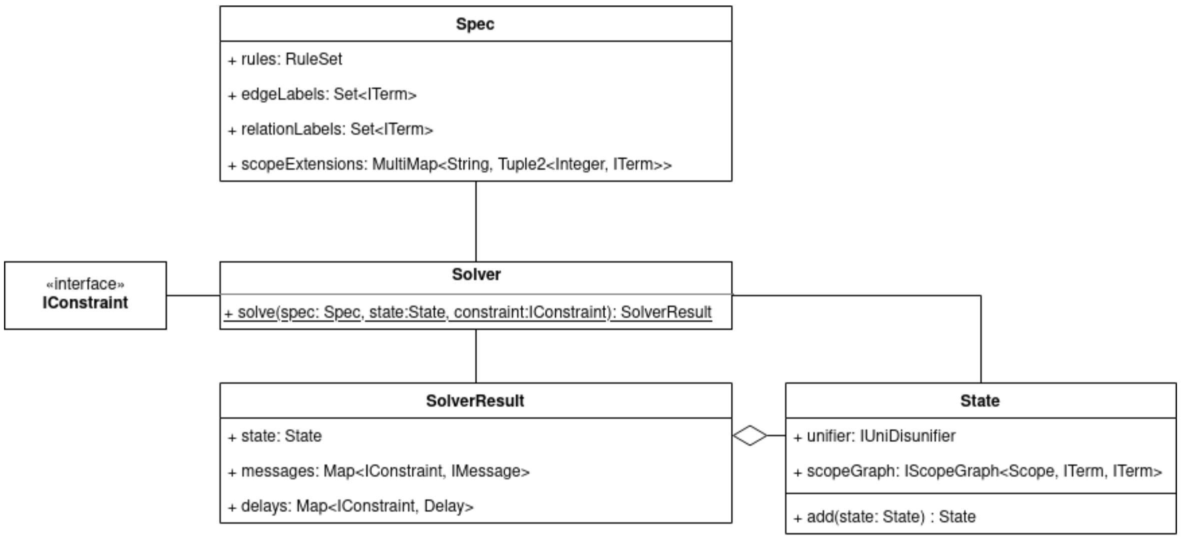

Spec
+ rules: RuleSet
+ edgeLabels: Set<ITerm>
+ relationLabels: Set<ITerm>
+ scopeExtensions: MultiMap<String, Tuple2<Integer, ITerm>>
«interface»
IConstraint
Solver
+ solve(spec: Spec, state:State, constraint:IConstraint): SolverResult
SolverResult
+ state: State
+ messages: Map<IConstraint, IMessage>
+ delays: Map<IConstraint, Delay>
State
+ unifier: IUniDisunifier
+ scopeGraph: IScopeGraph<Scope, ITerm, ITerm>
+ add(state: State) : State

Figure 5.4: Solver API

5.1.2 Solver

Second, Statix provides a solver, which solves a constraint using a specification. A UML diagram of the API is shown in Figure 5.4. The specification is defined in the `Spec` class, which closely resembles the compiled Statix modules. The solver returns a `SolverResult` object, which contains the result state of the solver. This state, defined in the `State` class, contains a scope graph, and a unifier that contains the values of all variables. Such a state can be passed to the solver as an initial state as well. Additionally, the `SolverResult` object contains a set of messages and a set of delayed constraints. Here the `Delay` class contains information about the reason why a constraint could not be solved yet.

5.1.3 Runtime

To integrate this solver in the Spoofax 2 Workbench, a runtime, implemented in Stratego, is provided. This runtime ensures the specification of a language is loaded, that the correct constraints are instantiated and solved. Finally, it creates messages based on the output of the solver.

The current Statix runtime has limited support for incremental analysis with file level granularity (Aerts 2019). This incremental analysis is performed in the following steps:

1. In order to create a global scope, the constraint `new s` is solved.
2. The `projectOk(s)` constraint is solved.
3. For each file, the constraint `fileOk(s, ast)` is solved, and the solver result is cached. In this output, there may be delayed constraints, which depend on information retrieved from the analysis of other files.
4. The states of the solver results of step two and three are combined using the `State.add` method. Using this state, the conjunction of all delayed constraints is solved. This solver run yields the final result. All messages from the solver results of steps two to four are collected. Moreover, the delayed constraints from step four are marked as errors.

When a file is changed, only step three for that file, and step four need to be recomputed. All partial analysis results of the other files can be reused. However, note that the majority of the work generally is performed in step four. Therefore the reuse of results is rather limited.

Finally, The Statix runtime provides an option to register Stratego transformations that are applied to the AST of a file before respectively after the analysis.

5.2 Background: Spoofax 3 Architecture

The Multi-language Statix Runtime is implemented in the Spoofax 3 language workbench (Gabriel Konat 2020). In this section we discuss the architecture of this workbench. To understand the architecture, we first discuss the design goals of Spoofax 3. After that we explain the architecture of PIE, which is used to define incremental pipelines, and Dagger, which is used for dependency injection. Finally, we explain the architecture of Spoofax 3 itself.

5.2.1 Spoofax 3 Design Goals

Spoofax 3 is a reimplementation of the Spoofax 2 Language workbench (Kats and Visser 2010). The main design goals of Spoofax 3 are to be ”modular, flexible, and correctly incremental”.

- Modular: languages should only have a dependency on the meta-components they actually use.
- Flexible: meta-components should depend on the core framework but the core framework should not depend on any meta-component. In this way, meta-components can be adapted or replaced without changing other parts of the system.
- Correctly Incremental: in Spoofax 3, compilation of languages, projects that use a language definition developed using the Spoofax 3 framework, and the framework itself should be compiled incrementally, with the possibility for languages to define their own incremental pipelines.

These design principles enable multi-language analysis. It is possible to add a new meta-component that executes multi-language analysis. The static analysis of languages that support multi-language analysis is implemented using that component. When loading the language, the component is integrated in the core framework. In this way, there is no tight coupling between the meta-component and the framework, nor between the meta-component and languages that do not support multi-language analysis.

5.2.2 PIE Build Pipelines

In Spoofax 3, incremental pipelines are implemented using *Pipelines for Interactive Environments* (PIE) (Gabriël Konat, Erdweg, and Visser 2018; Gabriël Konat, Steindorfer, Erdweg, and Visser 2018). PIE is a framework that aims at efficient and precise incrementalization of build scripts. Its efficiency requirement entails that only tasks that are affected by a change are recomputed. Similarly, the precision requirement states that results of an incremental build should be exactly the same as results from a clean build. In this section, we provide a high-level overview of PIE.

Central to the PIE framework is the concept of `taskdefs`. A `taskdef` is similar to a regular function, but it is automatically incrementalized by the PIE runtime. A `taskdef` called with a particular input is referred to as a `task`. A task can call other tasks, `require` resources (files), and `generate` other resources. When a resource is changed, either by an external change or when a task that provides that resource is recomputed, all tasks that required that resource will be recomputed as well.

PIE allows abstracting over a task definition or a task using respectively a `function` or a `supplier`. Functions and suppliers are regular values, and can hence be passed as arguments to other task definitions. This allows loose coupling of task definitions. Moreover, functions can be chained with the `map` method, which makes them useful to transform task inputs and outputs to the correct type.

In PIE, tasks can executed in two ways. When using the top-down approach, the called task calls a particular task definition. If needed, the framework will then compute the result of that task, and the tasks that it requires. On the other hand, when some resources are changed, all tasks that directly or indirectly depend on at least one of these resources are recomputed, and their observers are notified. This feature makes PIE especially suitable for interactive environments, such as an IDE.

In this thesis, we will present PIE pipelines using an enriched dialect of the PIE DSL presented in (Gabriël Konat, Steindorfer, Erdweg, and Visser 2018), although the actual implementation is done in Java. In Figure 5.5, we see an example of such a pipeline that compiles a Java project. At the first two lines, we see two regular functions, declared with the `func` keyword. The `foreign` keyword at the end of the declaration indicates that they are provided by an external dependency. The `javac` function compiles a single Java file to a class, file, and returns the output path. The `jar` function creates a Jar file from a list of class files (indicated by the asterisk). The `compile` task definition wraps the `javac` function. It declares the Java file as input using the `requires` keyword, and similarly indicates the file it produces using the `generates` keyword. These directives ensure that the PIE runtime recomputes the correct tasks when a file changes. Finally, the `build` task definition traverses over all Java source files in the `project` using the `walk` keyword. Each file is incrementally compiled using the `compile` task definition, and all compiled files are bundled in a jar file. Again, the `build` task indicates that it generates the jar file, to ensure correct incrementality.

```
func javac(file: path) -> path = foreign;
func jar(classes: path*) -> path = foreign;

taskdef compile(file: path) -> path {
  requires file;
  val classFile = javac(file);
  generates classFile; classFile
}

taskdef build(project: path) -> () {
  val classes = [compile(file) |
    file <- walk project with extension "java"
  ];
  val jarFile = jar(classes);
  generates jarFile;
}
```

Figure 5.5: Example PIE pipeline for creating a Jar file from a Java project

PIE is used extensively in Spoofax 3. In fact, all objects have to be immutable, and should either be the result of a PIE task, or be created when the framework is initialized. In this way, all mutable state is captured in PIE tasks, which (assuming that the task definitions are implemented correctly) enforces correct incrementalization.

5.2.3 Dagger Dependency Injection

Spoofax 3 uses Dagger[1] for Dependency injection. Unless most other dependency injection frameworks, Dagger is fully static. That means that Dagger creates all its bindings at compile time, using annotation processors.

The main entry point for the use of Dagger is an interface annotated with a `@Component` annotation. A component can be parameterized with modules, which are defined using the `@Module` annotation. Modules define bindings for objects by providing methods that are annotated with a `@Provider` annotation. The Dagger compiler ensures that the parameters of these methods are injected properly. A component can define abstract methods without parameters to define entry points that can be used to access its objects. The Dagger annotation processor will then generate an implementation for these methods that creates these objects

[1]https://dagger.dev/

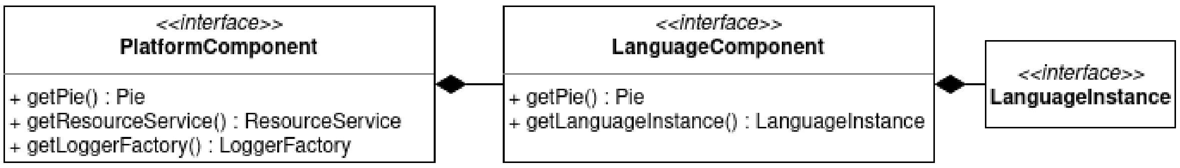


Figure 5.6: Spoofax 3 Core API

```
module multilang

data LanguageInstance = foreign {
  func extensions() -> String*;
  taskdef tokenize(file: path) -> Token*;
  taskdef style(tokens: supplier<Token*>) -> Styling;
  taskdef check(project: path) -> Message*;
}
```

Figure 5.7: Task Definitions in LanguageInstance

using the bindings its modules supply. Furthermore, Dagger uses the annotations provided by the `javax.inject` API to provide qualification and scoping of bindings.

In Dagger, it is possible to define a hierarchy of components. When an object is requested from a particular component, and the modules of the component do not define a binding for it, the request for the object is delegated to the parent components.

5.2.4 Spoofax 3

Central to the architecture of Spoofax 3 is the `LanguageInstance` interface from the *Spoofax Core* project (shown in Figure 5.6). This interface exposes properties and task definitions of a language that enable the Spoofax 3 framework components to work. Some of the interface members are shown in Figure 5.7.

Every *Language Definition Project* should implement this interface, probably using the meta-component libraries Spoofax 3 provides. The instantiation of a `LanguageInstance` subclass is managed by an implementation of the `LanguageComponent` interface. Moreover, Spoofax 3 Core provides a `PlatformComponent` interface, which allows accessing platform services, such as resources and logging.

Furthermore, Spoofax 3 provides libraries that integrate existing meta components, such as the JSGLR parser or the Statix runtime, into the Spoofax 3 framework. These libraries are used by the languages generated by the Spoofax 3 compiler, but do not have a dependency on Spoofax 3 Core.

Moreover, Spoofax 3 provides libraries that help to embed a language in a particular environment, such as Eclipse, IntelliJ or a command-line application. These projects provide a specialized instance of the `PlatformComponent` interface, and classes that integrate the services provided by a language in the environment. These libraries do not depend on any meta-component.

Finally, Spoofax 3 provides a language project compiler. This compiler takes a compiled Spoofax 2 language as input, and transforms it to integrate with the Spoofax 3 framework. In particular, it creates an implementation of the `LanguageInstance` and `LanguageComponent` interfaces. To perform this task, it embeds the compiled artifacts from the Spoofax 2 language in the jar file that is created, and generates classes that load these resources. Based on these resources, it creates PIE task definitions, which are injected to the language instance by the language component. Additionally, it can create a command line interface, an Eclipse plu-

```
module mylang:tasks

// Read file to string
taskdef readFile(file: path) -> String {
  requires file; read file
}

// List source files of language in directory
taskdef sourceFiles(project: path) -> path* {
  requires project; walk project with extensions lang.extensions()
}

taskdef parse(str: supplier<String>) -> (AST, Token*, Message*) {
  jsglr.parse(str.get())
}

taskdef tokenize(file: path) -> Token* {
  val (_, tokens, _) = parse(readFile.supplier(file)); tokens
}

taskdef style(tokens: supplier<Token*>) -> Styling {
  esv.style(tokens.get())
}

taskdef analyze(
  sources: supplier<path*>,
  parser: function<supplier<String>, AST>
) -> Message* {
  val asts = [ast |
    ast = parser.apply(readFile.supplier(file)),
    file <- sources.get()
  ];
  statix.analyzeMulti(asts).messages
}

taskdef check(project: path) -> Message* {
  val parseMessages = [msg | msg <- msgs,
    (_, _, msgs) = parse(readFile.supplier(file)),
    file <- sourceFiles(project)
  ];
  val analysisMessages = analyze(
    sourceFiles.supplier(project),
    parse.function.map((ast, _, _) -> ast)
  );
  parseMessages + analysisMessages
}
```

Figure 5.8: Spoofax 3 Default Pipeline

gin and an IntelliJ plugin, based on the libraries Spoofax provides. Usually, the Spoofax 3 compiler is invoked using a Gradle plugin.

In Figure 5.8, we show the default task definition structure that is generated for language projects with multi-file analysis. Most of the tasks have a straightforward implementation. The `parse` task definition uses the `jsglr` parser to parse its input. In a similar manner, the `style` and `analyze` tasks use an `esv` and `statix` variable to implement their task. In this code, these variable are assumed to be globally available, while in the actual implementation, they are injected into the task definition by the `LanguageComponent`.

The `analyze` task takes two arguments. The `sources` parameter provides the task with all the files that should be analyzed. The `parser` argument is an incremental function that returns an AST, based on a string supplier. For each file, this function is called with a supplier based on the `readFile` task definition. Note how this pattern decouples the `analyze` task from the `parse` task or the file extensions of the language. Finally, the `check` task integrates these tasks. It calls the `parse` and `analyze` tasks with the correct inputs, and aggregates their messages.

5.3 Implementation of Multi-Language Analysis

Having introduced Spoofax 3, we explain how we implemented a runtime that can execute specifications designed according to the pattern introduced in chapter 4. After defining some terminology, we discuss the architecture of the Multi-language meta-component. Finally, we describe how we embedded this component in the Eclipse IDE environment, and how we extended the Spoofax 3 language compiler to generate the boilerplate code that the component needs.

5.3.1 Definition of Terms

First, we define some terminology we use throughout this section:

- Module: a single Statix module. Contrary to the previous chapter, we will mostly refer to compiled modules.
- Partial Specification: an incomplete collection of modules. In general, a module is a partial specification. Furthermore, multiple partial specifications can be merged to form a new partial specification.
- Specification Fragment: a partial specification, containing all the modules of a particular language project. A specification fragment can depend on other specification fragments. When a fragment has dependencies, the fragment itself does not contain the modules of the dependencies. In practice, all modules of an interface constitute a fragment, just as all modules of a particular language.
- Specification: a complete, executable collection of specification fragments. In a specification, all fragment dependencies should be satisfied. That is, when fragment F depends on fragment F', all specifications that include F should by definition include F' as well.
- Composed Specification: a specification that is composed of fragments that were not compiled together.
- Analysis Context: the shared data and configuration of group of languages that are analyzed in conjunction. A context can be referenced by its identifier, or accessed via any language that it includes.

5.3.2 Metadata

In this section, we describe the static metadata that is required for the component to execute properly. All required metadata is collected in the `AnalysisContextService` (shown in Figure 5.9). For each language, its default context identifier is recorded. This context will be used when the project configuration does not override it. Furthermore, some information about a language that is specific for a language that supports multi-language analysis is maintained in `LanguageMetadata` objects. This information includes:

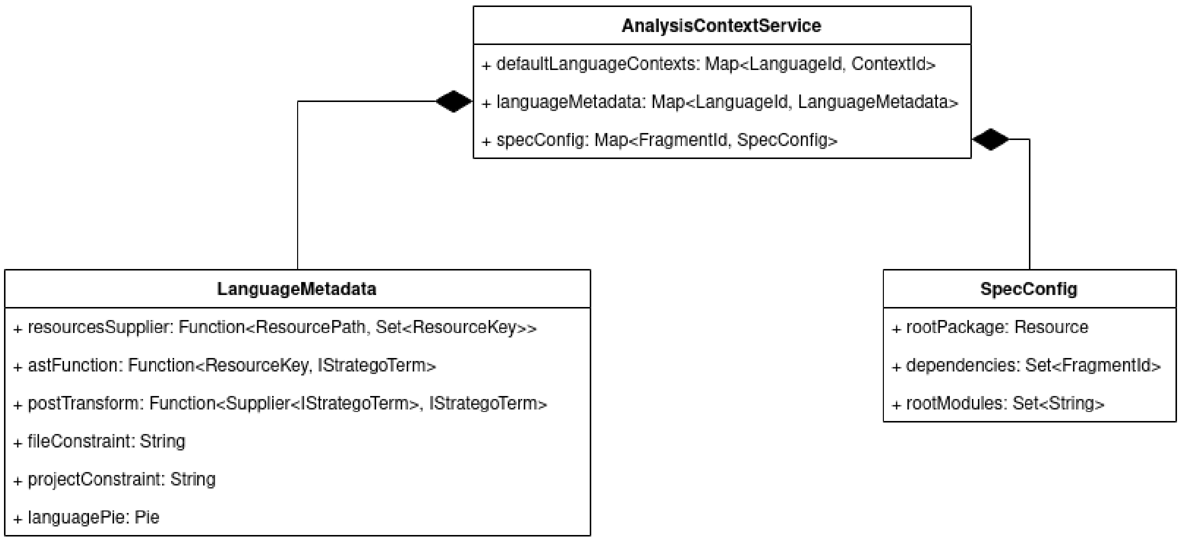

AnalysisContextService
+ defaultLanguageContexts: Map<LanguageId, ContextId>
+ languageMetadata: Map<LanguageId, LanguageMetadata>
+ specConfig: Map<FragmentId, SpecConfig>
LanguageMetadata
+ resourcesSupplier: Function<ResourcePath, Set<ResourceKey>>
+ astFunction: Function<ResourceKey, IStrategoTerm>
+ postTransform: Function<Supplier<IStrategoTerm>, IStrategoTerm>
+ fileConstraint: String
+ projectConstraint: String
+ languagePie: Pie
SpecConfig
+ rootPackage: Resource
+ dependencies: Set<FragmentId>
+ rootModules: Set<String>

Figure 5.9: Analysis Metadata

- A PIE function that returns the paths of all files of a language in a particular directory.
- A PIE function that returns the AST of a particular file.
- A PIE function that executes a transformation after applied to the analyzed AST.
- The names of the root constraints, as introduced in subsection 4.1.4.

The role and the definition of these functions will be discussed in subsection 5.3.4.

Furthermore, information that is required to load a specification fragment is collected in `SpecConfig` objects. These objects contain the following information:

- A resource in which the modules of this fragment are located.
- The fragment identifiers of the fragments this fragment depends on.
- A set of root modules that serve as entry points for the specification loading algorithm.

In this configuration, a distinction is made between languages and specifications. This distinction is made to reflect the fact that interfaces are not complete languages, but rather specification fragments. Hence a `LanguageMetadata` object can not be provided for interfaces, and neither is it needed.

When initializing an `AnalysisContextService`, we validate that all languages have an accompanying specification fragment. Moreover, we validate that all specification fragment dependencies can be satisfied.

Every platform has exactly one `AnalysisContextService` instance. This instance is injected into all language-parametric task definitions that use either a `LanguageMetadata` or a `SpecConfig` instance. Therefore, in the code snippets in the remainder of this section, we assume that an `analysisContextService` variable is implicitly available. Similarly, the language-specific task definitions and `data` types have an implicit `language` variable, which denotes the language instance, and an implicit `languageId` variable, that denotes the `LanguageId` of the language.

5.3.3 Compiling Multi-Language Specifications

Besides the language-specific metadata, we need to obtain the specification that we want to execute. In this section, we explain how we compile Statix specifications, and what their runtime representation looks like. How the compiled specifications are loaded into the runtime is explained in subsection 5.3.4.

At the time this framework was implemented, it was required to have a Spoofax 2 language project to create a Spoofax 3 language. The Spoofax 3 compiler did not compile sources of meta-languages itself, but rather generates boilerplate code that wraps the artifacts of a Spoofax 2 language. Hence, we must define Spoofax 2 language projects for our languages first.

In general, there are two types of projects that we need to consider. First, an interface project contains only the Statix sources that define the interface. Those can be compiled using the standard Statix compiler, and packaged in a Spoofax language artifact.

Second, there are concrete languages, which have specifications for other aspects, such as a syntax definition or editor service specifications, as well. The Statix compiler does not support separate compilation, but requires the sources of all modules that are referenced to be present. Hence, when compiling the Statix specification of a concrete language, we need access to the interface definition. Unfortunately, it is not possible to reference Statix modules included in (source) dependencies. Therefore we copy these into the language project using gradle tasks. Thus, the interface is re-analyzed and recompiled for each language that implements it. Using the exports option from the `metaborg.yaml` file, we ensure that a language artifact does not include the compiled modules from the interface. In this way, we ensure that all modules are included in a language exactly once.

Figure 5.10 shows how the specification is defined in the runtime. The `Spec` class is provided by the Solver API, as shown in Figure 5.4. The `Module` class wraps a `Spec` instance for a single module. A `SpecFragment` wraps all modules from a single language project, and explicitly records which dependencies are not yet resolved. Such delayed dependencies occur when a module from a concrete language imports a module from an interface, which are included in the fragment for the interface, and not in the fragment from the language.

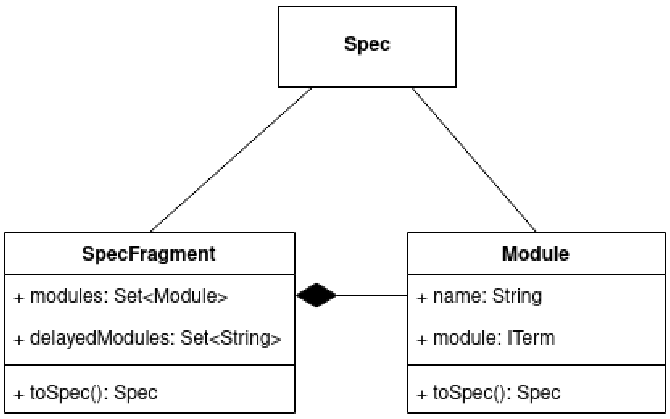


Figure 5.10: Specification definition

In Figure 5.11, we summarize the specification management graphically. On the left of the picture, there are three language projects. The first project is an interface, and the other two are projects for a concrete language. The statix modules of the interface are copied to the languages, but their compilation artifacts (the `.spec` files) are not included in the language artifacts of the concrete languages, but only in the archive for the interface language.

At the right hand side of the picture, the runtime representation of the specification is shown. Each source module corresponds to a `Module` instance, and similarly does each `SpecFragment` correspond to a language project. Moreover, the `delayedModules` field maintains the modules that must be included from another fragment. These delayed modules correspond with `import` statements in the original Statix source files. Since module `d` only imports module `b` from the interface, the `delayedModules` set in the fragment of language B contains only `b`. Finally, we see how fragments are combined into full specifications on the right. The `langA Spec` is the specification that is used to solve the partial constraint of lan-

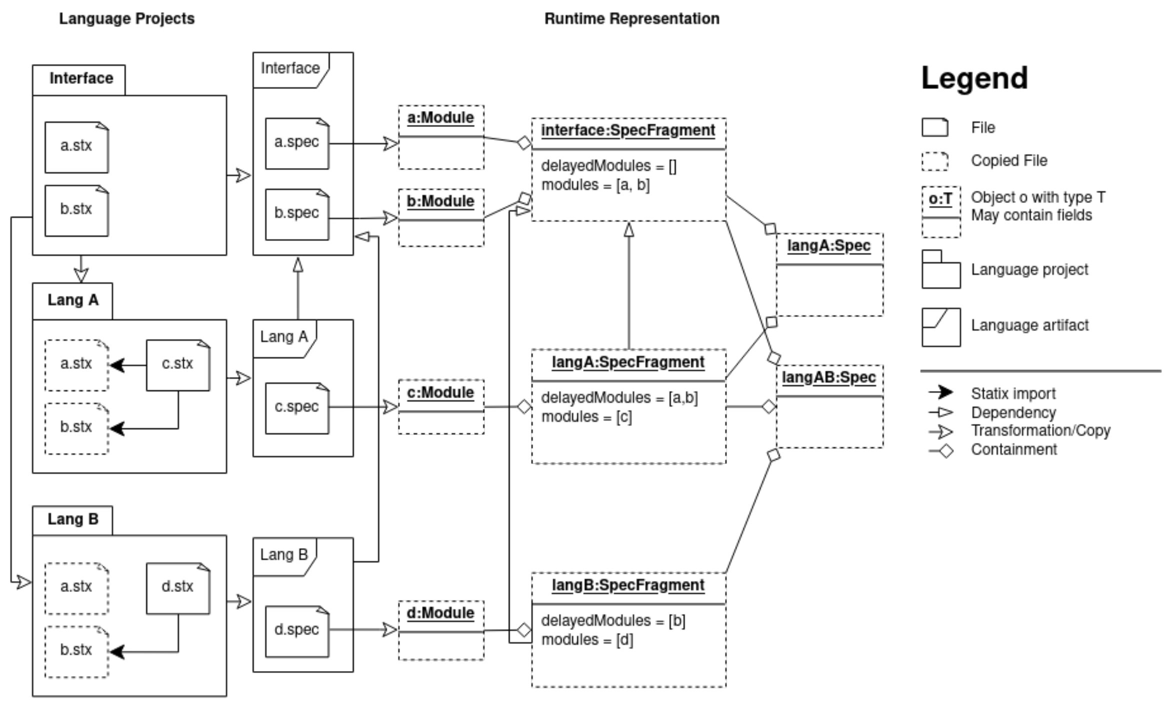


Figure 5.11: Specification Management

guage A. Furthermore, the `langAB Spec` is the specification that is used to perform the final phase of the analysis in a context where language A and B analyzed together.

5.3.4 Pipeline

Next, we discuss the PIE pipeline that we implemented to be able to execute analysis incrementally. We first describe how the ASTs are transformed, how the correct specification for each task is loaded, and how project-specific configuration is loaded. After that, we explain how the analysis is executed, and how the results are processed. Finally, we discuss some implementation details on how we handle exceptions, PIE usage, solver logging and shared term factories.

In this section, we use the PIE DSL to represent the various task definitions that constitute the system. Some of these task definitions are part of the framework, while others are provided by the language implementation. In the code snippets, this is indicated with a top-level `module` statement. All task definitions in the `multilang` module are provided by the framework, and all task definitions in the `mylang` module are language-specific.

AST transformations. First, we discuss the part of the pipeline that transforms resources. The pipeline is shown in Figure 5.12. First, there are two library functions that add a resource attachment and a Term index annotation to the AST. These attachments and annotations are used by the runtime to determine the correct position of messages. Moreover `applyStrategy` function applies a strategy to an AST.

The first two functions are used by the `index` task definition to add the attachments and annotations to the AST that provided by the `parse` task output provided `resource` parameter.

After indexing the AST, the `preTransform` task definition applies the pre-analysis transformation strategy to the annotated AST. We separated this task from the `index` task definition because the abstract task definition for transformations requires a `supplier<AST>`, while the `index` task must have access to the `path` of the resource it is indexing. Furthermore, this al-

```
module mylang:multilang

// Library function adding ResourceKey attachment to AST
func setResource(ast: AST, resource: path) -> AST = foreign;
// Library function adding term indices
func addIndices(ast: AST) -> AST = foreign;
// Library function applying stratego transformation on an ast
func applyStrategy(strategy: String, ast: AST) -> AST = foreign;

taskdef parse(str: supplier<String>) -> (AST, Token*, Message*) { ... }

taskdef index(resource: path) -> AST {
  val (ast, _, _) = parse(readFile.supplier(resource));
  val ast' = setResource(ast, resource);
  addIndices(ast')
}

taskdef preTransform(ast: supplier<AST>) -> AST {
  applyStrategy("pre-transform", ast.get())
}

taskdef postTransform(result: supplier<(AST, SolverResult)>) -> AST {
  val (ast, _) = result.get();
  applyStrategy("post-transform", ast)
}
```

Figure 5.12: Transformation Task definitions

```
module mylang:multilang

data MylangMetadata : LanguageMetadata {
  resourcesSupplier() -> function<path, path*> {
    project -> walk project with extensions language.extensions()
  },
  astFunction() -> function<path, AST> {
    resource -> preTransform(index.supplier(resource))
  },
  postTransform() -> function<supplier<(AST, SolverResult)>, AST> {
    postTransform.function
  }
}
```

Figure 5.13: Transformation Integration in LanguageMetadata

lows a language designer to provide a custom implementation for the pre-transformation task definition without touching the required `index` task definition.

Finally, the `postTransform` task definition applies the post-analysis transformation strategy to the AST returned by the `preTransform` and the final analysis result. In its default implementation, it applies the post-analysis transformation strategy without using the solver result. Together, the `preTransform` and `postTransform` strategies provide the same features as the transformation hooks in the Spoofax 2 Statix runtime.

It must be noticed that all these task definitions are language-specific. The task definition bodies presented in Figure 5.12 are the default implementations as created by the compiler. The Runtime therefore cannot call them directly, but rather uses the `LanguageMetadata` instance for a language to retrieve PIE functions wrapping these task definitions. Figure 5.13

```
module multilang:spec

func loadModule(mod: path) -> Module = foreign;
func imports(mod: Module) -> String* = foreign;
func combine(fragments: SpecFragment*) -> Spec? = foreign;

taskdef loadFragment(id: FragmentId) -> SpecFragment {
  val config = analysisContextService.getSpecConfig(id);
  var loaded = []; var delayed = [];
  var worklist = [mod | mod <- config.rootModules];
  while(modName <- worklist) {
    val modPath = config.rootPackage.append(modName);
    if (!exists modPath) {
      // Module in another fragment (i.e. interface)
      delayed += modName; continue;
    }
    requires modPath;
    val mod = loadModule(modPath); loaded += mod;
    worklist += [import | import <- imports(mod)
      if (!import in loaded && !import in delayed && !import in worklist)
    ];
  };
  SpecFragment {
    modules = loaded,
    delayedModules = delayed
  }
}

func dependencies(id: FragmentId) -> Set<FragmentId> {
  val directDeps = analysisContextService.getSpecConfig(id).dependencies;
  val transDeps = [dep |
    dep <- dependencies(id),
    id <- directDeps
  ];
  set(transDeps + id)
}

taskdef buildSpec(ids: LanguageId*) -> Spec? {
  val fragments = [loadFragment(fragment) |
    fragment <- dependencies(lang), lang <- ids
  ];
  combine(fragments)
}
```

Figure 5.14: Transformation Task definitions

illustrates how these task definitions are wrapped into the metadata instance of a particular language. When we discuss the part of the pipeline that actually executes the analysis, we will demonstrate how this interface is used.

Spec loading. Secondly, the runtime will load the specification that it requires to do the analysis. In subsection 5.3.3, we have shown how the specification is defined in the language artifacts and the runtime. In this section, we discuss the task definitions that load the specification. The implementation of these task definition is shown in Figure 5.14.

The `loadFragment` task definition loads all modules of a single specification fragment. First it retrieves the config of this specification using the analysis context service. The loading algorithm maintains three lists. First, the `loaded` variable contains all loaded modules. The `delayed` variable holds the names of all modules that are imported by a module in `loaded`, but that could not be found in the fragment root package. These modules are for now assumed to be in a fragment dependency. Finally, the `worklist` variable contains all module names that the still need to be loaded. When a module in the worklist cannot be resolved, it is added to the `delayed` list. Otherwise, it is loaded and added to the `loaded` list. After that, all imports of the module are added to the worklist if they have not yet been encountered.

Preferably, we would have recursively walked the package in which the specification is located. However, listing package contents is not supported inside a jar file, breaking the currently default way of distribution.

The `buildSpec` task definition takes a set of language identifiers, and loads the combined specification for these languages. It first computes the identifiers of all fragments that must be included by transitively including the dependencies of the supplied languages as recorded in their `SpecConfig`. Then it loads all individual specification fragments using the `loadFragment` task. Finally, the `Spec` instances of all fragments are combined into a single specification. During combination, declarations of labels, relations and constraints are qualified with the identifier of their fragment. Likewise, the references are qualified with the fragment to which they originally resolved as well. This yields an alpha-equivalent specification, in which no naming collisions between declarations in sibling fragments can occur. Moreover, the `combine` method validates that this set of fragments is a valid specification by checking that:

- For each fragment, all modules that could not be resolved in the fragment itself are supplied by another fragment.
- Rules for a constraint are contained in one fragment. That is, it can not occur that a fragment contains a rule for a constraint declared in another fragment.

For an example on how Specification fragments relate to language projects and complete specifications, we refer to Figure 5.11.

Comparing this procedure to the Statix runtime of Spoofax 2, we see that the concept of specification fragments is new. We introduced it to ensure we can properly validate the consistency of *composed specifications*, and to ensure that we do not include interface multiple times in a composed specification. The procedure of merging modules is similar to the implementation in Spoofax 2[2].

Configuration loading. Multi-language analysis takes place in a particular context. This context has the following properties:

- The languages included in it.
- The log level passed to the solver debug context.

```yaml
languageContexts:
  'mb.minisdf': "ctx"
  'mb.ministr': "ctx"
contextConfigs:
  ctx:
    logging: "debug"
```

Figure 5.16: Configuration

In subsection 5.3.2, we explained that each language provides a default context identifier. Furthermore, the default log level is set to `warn`. Therefore, having a project-specific configuration file is entirely optional.

However, it is possible to override these settings using a `multilang.yaml` file in the root of the project. Using this file the default values for the settings named in the above paragraph can be overridden. An example of such a configuration file is shown in Figure 5.16. In this file,

[2]https://github.com/metaborg/nabl/blob/master/statix.runtime/trans/statix/runtime/analysis.str

```
module multilang:config

data Config = foreign {
  contexts: Map<LanguageId, ContextId>,
  settings: Map<ContextId, Settings>
};
data Settings = foreign { ... };
data ContextConfig = foreign {
  languages: LanguageId*,
  settings: Settings
};

func defaultSettings() -> Settings = foreign;
func deserializeConfig(str: String) -> Config = foreign;

taskdef readConfig(project: path) -> Config {
  val configFile = project.append("multilang.yaml");
  if (!exists configFile) {
    Config { }
  } else {
    requires configFile;
    deserializeConfig(read configFile)
  }
}

taskdef buildConfig(langId: LanguageId, project: path) -> ContextConfig {
  val config = readConfig(project);
  val defaultContextId = analysisContextService.defaultContext(langId)
  val contextId = config.contexts.getOrDefault(langId, defaultContextId);
  var languages = [lang | lang <- analysisContextService.languagesForContext(contextId)];
  languages -= config.contexts.keys;
  languages += [lang |
    (lang, context) <- config.contexts if context == contextId
  ];
  ContextConfig {
    languages = languages,
    settings = config.settings.getOrDefault(contextId, defaultSettings())
  }
}
```

Figure 5.15: Configuration Task definitions

Mini-SDF and Mini-STR are configured to run in the `ctx` context. This context is configured with log level `debug`.

This configuration is read and parsed by the `readConfig` task. When there is no configuration file, an empty `Config` instance is returned. The `buildConfig` task specializes the configuration to a particular language. When an option is not customized, a default value from the `AnalysisContextService` is substituted. Moreover, the set of languages that are included in the context is made explicit. From the default set of languages, all languages that override their default are removed, and all languages that explicitly declare to be in the same context of the `language` parameter are included again.

Executing Analysis. Now we have all our inputs in place, we can explain how we implemented the actual analysis. The implementation we provide closely resembles the current runtime, as discussed in subsection 5.1.3. A representation of the implementation in the PIE

DSL is shown in Figure 5.17.

As the first step of the pipeline, the `globalScope` task creates a global scope by solving the `new s` constraint (embedded using the `$stx` syntax). The global scope variabe and the initial state are passed to all the other tasks.

Second, the `solveProject` task definition ensures that the project constraint for a particular language, of which the identifier is passed as an input, is solved. The output of the task is the `SolverResult` that is returned by the solver. Note how the `analysisContextService` is again used to find language-specific information in a language-parametric task.

Third, the `solveFile` task definition implements the third step of the analysis. This task definition takes a language identifier and a resource key as input. Then it requires the AST for the resource using the `preTransform` function of the language that was passed as input. After retrieving the proper AST, the task creates the specification of the language using the `buildSpec` task. Subsequently, the Statix solver is called with this specification to solve the language's `fileOk` constraint for the supplied AST. The returned `SolverResult`, together with the AST passed to it, is returned as output.

The fourth step of the pipeline is implemented in the `analyzeProject` task definition. This project takes a project path and a set of language identifiers as input. For all languages included in its context, it computes the solver results of its project constraints, and the file results of all sources of the languages. The source files for a language are resolved using the `resourcesSupplier` method retrieved using the appropriate language metadata. The intermediate results of these partial analyses are merged, and the conjunction of all residual constraints are solved. The final solver result, as well as the intermediate results of all file and project constraints, are provided as output.

Finally, the `check` task definition, shown in the Figure 5.18, integrates this pipeline into the Spoofax 3 framework. It takes a project path as input, and provides a list of messages. This signature ensures that the `check` method of a `LanguageInstance` implementation (as introduced in Figure 5.7) of a language that supports multi-language analysis can use this task definition. After extracting the parse messages for all files, the task computes the configuration for the context, based on the project and the language is (which is globally available). Then, it executes the `analyzeProject` task for the languages in the context. Based on the analysis result, messages are created. In order to prevent the duplication of messages, only messages for the language for which this task is defined are returned.

We split the `analyzeProject` and `check` task definitions to enable other tasks to use the analysis results. In this way languages can define tasks that perform other processing of the analysis results. Moreover, the `check` task specialized the output to a particular language, while the `analyzeProject` task definition is language-parametric.

In Figure 5.19, we summarize all task definitions we discussed in this section. Columns two to four list the responsibility that each task has. The last column indicates whether a task is language-specific or generic. Generic tasks are implemented fully in the runtime library. The language-specific tasks are generated per language, based on an abstract class in the runtime library.

Exception Handling. In this analysis pipeline, we need to handle exceptions appropriately. When an exception occurs during analysis, we do not want our environment to crash, nor bother users with pop-ups. On the other hand, we do not want exceptions to be ignored silently. In order to make it easier to implement this behavior properly, Spoofax 3 has a `Result<V, E>` type, which is either a value with type `V`, or an exception with type `E`. All task definitions that can fail actually have such a result type as output. To keep the figures concise, we have shown only `V` in this section.

Furthermore, when some tasks require multiple other tasks, it can occur that multiple exceptions are returned. To handle these properly, we introduced a custom `MultilangException`

```
module multilang:analysis

typealias FileResult = (path, LanguageId, SolverResult, AST);
typealias ProjectResult = (LanguageId, SolverResult);
typealias FinalResult = (FileResult*, ProjectResult*, SolverResult);

func makeConstraint(name: String, params: Term*) -> Constraint = foreign;
func solve(spec: Spec, constraint: Constraint, state: State) -> SolverResult = foreign;
func merge(states: State*) -> State = foreign;
func conjoin(constraints: Constraint*) -> Constraint = foreign;

taskdef globalScope() -> (Var, SolverResult) {
  val res = solve(emptySpec(), $stx({s} new s), emptyState());
  (extractVar(res, "s"), res)
}

taskdef solveProject(lang: LanguageId) -> SolverResult {
  val (s, state) = globalScope();
  val cname = analysisContextService.getMetadata(lang).projectConstraint;
  solve(buildSpec([lang]), makeConstraint(cname, [s]), state)
}

taskdef solveFile(lang: LanguageId, res: path) -> (AST, SolverResult) {
  val metadata = analysisContextService.getMetadata(lang)
  val ast = metadata.preTransform().apply(res);
  val (s, state) = globalScope();
  val cname = metadata.fileConstraint;
  val res = solve(buildSpec([lang]), makeConstraint(cname, [s, ast]), state);
  (ast, res)
}

taskdef analyzeProject(context: LanguageId*, project: path) -> FinalResult {
  // Collect partial results
  val projectResults = [solveProject(lang) | lang <- context];
  val fileResults = [ (file, lang, res, ast) |
    (res, ast) = solveFile(lang, file),
    file <- metadata.resourcesSupplier().apply(project),
    metadata = analysisContextService.getMetadata(lang)
    lang <- context
  ];
  val results = projectResults + [res | (_, _, res, _) <- fileResults];
  // Solve residual constraints
  val constraint = conjoin([res.delay | res <- results]);
  val state = merge([res.state | res <- results]);
  val finalResult = solve(buildSpec(context), constraint, state);
  // Apply post-transformations
  val fileResults' = [(f, l, res, ast') |
    ast' = metadata.postTransform().apply(() -> (ast, finalResult)),
    metadata = analysisContextService.getMetadata(l),
    (f, l, res, ast) <- fileResults
  ];
  (fileResults, projectResults, finalResult)
}
```

Figure 5.17: Analysis Execution Task definitions

```
taskdef check(project: path) -> Message* {
  val parseMsgs = /* omitted */;
  // Execute analysis
  val config = buildConfig(languageId, project);
  val (fileRes, projectRes, finalRes) = analyzeProject(config.languages, project);
  // Extract messages
  val projectMsgs = [msg | msg <- res.messages,
    (l, res) <- projectRes if l == languageId
  ];
  val fileMsgs = [msg | msg <- res.messages,
    (_, l, res, _) <- fileRes if l == languageId
  ];
  val finalMsgs = [msg | msg <- finalRes.messages if msg.origin.languageId == languageId];
  parseMsgs + projectMsgs + fileMsgs + finalMsgs
}
```

Figure 5.18: Check Task definition

Name	Role	Input	Output	Generic
`index`	Add metadata annotations to AST	`path`	`AST`	No
`preTransform`	Apply transformation on AST before analysis	`AST`	`AST`	No
`postTransform`	Apply transformation on AST after analysis	`(AST, SolverResult)`	`AST`	No
`loadFragment`	Load specification fragment	`FragmentId`	`SpecFragment`	Yes
`buildSpec`	Create specification for languages	`LanguageId*`	`Spec`	Yes
`readConfig`	Load project configuration	`path`	`Config`	Yes
`buildConfig`	Create configuration for language in project	`(LanguageId, path)`	`ContextConfig`	Yes
`globalScope`	Create global scope	`()`	`SolverResult`	Yes
`solveProject`	Solve project constraint	`LanguageId`	`SolverResult`	Yes
`solveFile`	Solve file constraint	`(LanguageId, path)`	`FileResult`	Yes
`analyzeProject`	Finish constraint solving	`(LanguageId*, path)`	`FinalResult`	Yes
`check`	Create messages for project	`path`	`Message*`	No

Figure 5.19: Summary of Task Definitions

type. When multiple exceptions are reported, a new exception is constructed which takes all the original exceptions as parameter. In this way, all exceptions are threaded up the task hierarchy.

When the task executing results in an exception, the `check` task definition transforms the exception into messages, and displays them at the proper location in the editor. In this way, exceptions are neither silently ignored nor reported too obtrusively.

PIE usage. When using multi-language analysis, a single session can execute tasks and access resources from different languages. However, a `Pie` instance needs to have all task definitions registered explicitly, before a task definition can be called. Therefore, we have two `Pie` instances for a language. The *prototype* `Pie` contains only the task definitions for the languages. This instance is provided by the `LanguageMetadata.languagePie()` method. The `AnalysisContextService` then creates a `Pie` instance with the prototype `Pie` instances of all languages as ancestors. This instance is injected into the `LanguageComponent` of each language. In this way, the `Pie` instance that is used for multi-language analysis can always access all task definitions it uses.

Log Level Input. Furthermore, the framework allows to configure the log level of the Statix solver. This log level is passed as an input to all applicable tasks. However, the log level should not influence the result of the analysis. Hence, we do not want to re-analyze a project when the log level is changed. Therefore, we did not include the log level argument in the identity of an input. In this way, tasks will not be re-instantiated when the log level is changed. After cleaning the project, the analysis can be rerun with a different log level.

Term Factory. When loading Statix specifications and transforming ASTs, we need to supply an `ITermFactory` instance to properly build terms. Especially, to ensure Statix terms can be build correctly, and error messages can be located properly, usage of a specific implementation, the `ImploderOriginTermFactory`, is required. However, we do not want the Multi-language component to tightly integrate with this particular implementation, because we want to support (future) languages that use a custom term factory implementation as well. Therefore, we allow languages to provide their own term factory using the `termFactory` method from the `LanguageMetadata` interface.

5.3.5 Injection

Having discussed the Multilang Component in isolation, we explain how we integrated this component into the Spoofax 3 framework. First, we introduced the `MultiLangComponent` class, which provides an `AnalysisContext` instance, and instances for all task definitions. This component depends on the `PlatformComponent` and a `MultiLangModule`. The constructor of this module takes an `AnalysisContext` supplier as argument, which is injected into all applicable task definitions. Language components can declare this component as parent component.

However, this approach initially introduced a cyclic dependency. In order to initialize the `AnalysisContextService`, we need to aggregate information about particular languages, retrieved from the components of these languages. However, to initialize these components, a `MultiLangComponent` instance must be supplied, which requires an `AnalysisContextService`. To break this cycle, we made the initialization of the `AnalysisContextService` lazy, and call the supplier only when the first instance is requested. Because the methods of this class are only called by the task definitions introduced above, this approach is safe.

5.3.6 Integration in Eclipse

To embed this framework in Eclipse, we introduced a new Eclipse plugin. This plugin provides a singleton `AnalysisContextService`. Eclipse plugins for particular languages can retrieve this component and initialize their own components with it.

However, individual languages must supply their `LanguageMetadata` and `SpecConfig` instances to the Multi-language plugin as well. Because all classes in Spoofax 3 should be immutable, we cannot just add them to the `AnalysisContextService` we retrieved from the component. Instead, we introduced an *extension point*. Plugins for concrete languages can use this extension point to register implementations of the `LanguageMetadataProvider` interface. When the Multi-language plugin initializes the `MultiLangModule`, it provides a supplier that uses the extension point to resolve all language metadata providers, and creates an analysis context service based on the aggregated information.

Moreover, we needed to ensure that eclipse handles concurrency correctly. Until now, builds of individual languages did not interfere, and hence they could be executed in parallel. For builds with multi-language analysis this is not the case. Therefore, the plugin provides a scheduling rule that ensures only one multi-language build is running at the same time. This rule is injected into the eclipse plugins for a language.

5.3.7 Generating Boilerplate

As follows from the previous sections, there is a fair amount of boilerplate that should be supplied by a language. Since we do not want language developers to write that, we extended the Spoofax 3 compiler to generate it. This compiler takes the following input parameters:

- *Root Modules*: non-empty list of modules that the `loadFragment` task uses as initial items in the worklist.
- *Language identifier*: string used to identify the language. Default value is the base package of the language.
- *Default context*: the identifier of the default context that this language uses. Its default value is the language identifier, meaning that static analysis is done in isolation. Alternatively, the identifier of the interface could be provided, meaning that multi-language analysis is performed by default. The context identifier can be customized per project using project specific configuration, as we explained above.
- *Dependencies*: references to Spoofax 3 projects that supply a specification component that this language depends on. By default a language has no dependencies.
- *File constraint*: name of the file constraint, introduced in subsection 4.1.4. Its default value is `statics!fileOk`.
- *Project constraint*: name of the project constraint. Its default value is `statics!projectOk`.
- *Pre-analysis strategy name*: name of the stratego strategy that is to be executed before analysis. Its default value is `pre-analyze`.
- *Post-analysis strategy name*: name of the stratego strategy that is to be executed after analysis. Its default value is `post-analyze`.

First, the compiler generates a `SpecConfigFactory`. This class provides the identifier for this specification fragment, and a `Map<FragmentId, SpecConfig>` that contains the specification configurations for this fragment, and all its dependencies. The specification configurations are retrieved by calls to their configuration factories. We include them into this configuration to ensure that we do not need to create an eclipse plugin for each interface fragment. The specification is included in the `src-gen/statix` folder inside the java package of the language.

Second, when the project implements a full language (not an interface), the compiler creates implementations for the language specific tasks. In the transformation tasks, the correct strategy names are inserted. In the `index` task definition, a parse function is injected. Additionally, in the `check` task, the language identifier is substituted. Finally, an `cmdAnalyzeProject` task definition is generated. This task definition converts the output of the `check` task to the `CommandFeedback` type, to ensure multi-language analysis can be used from the command line.

Third, the compiler adjusts some other classes to fully integrate multi-language analysis in the framework. First, adds a dependency on the multi-language component on the generated language component. Furthermore it adds a method that provides the metadata of the language. Third, the `check` method of the generated language instance class is adjusted to supply a task based on the `check` task generated by the multi-language analyzer compiler, instead of the default one. In this way, the pipeline can be embedded in the different platforms. Last, the language module is adapted to provide the specification configuration retrieved from the specification factory, and to create a language metadata instance that aggregates and exposes all information from the settings and generated sources.

Last, the compiler adjusts the generated Eclipse plugin. The component supplied by the Multi-language plugin is added to the component supplied by the language plugin. Second, an extension is added to the extension point that the Multi-language plugin provided. This extension supplies the specification configuration and language metadata to the analysis context service.

5.4 Conclusion

In this chapter, we introduced Spoofax 3 and the Statix Compiler and runtime. In this environment, we have implemented a meta-component that allows to execute composed type systems that are implemented according to the Shared Concept Interface design pattern introduced in chapter 4. This meta-component is integrated in the Eclipse IDE, and the language-specific boilerplate code can be generated by the language compiler.

Having presented a design pattern and accompanying runtime, we will evaluate our approach in the next chapter. After that, we compare our approach with the other approaches we found in the literature, and conclude this thesis.

Chapter 6

Evaluation

In this chapter, we evaluate our approach with respect to the criteria we posed in section 2.4, and the guarantees about composition it provides. In this evaluation, we used the criteria by posed Leduc, Degueule, Wyk, and Combemale (2020). After that, we recall the criteria for the Multi-Language Programming Environment, and discuss our compliance with them.

After discussing the functional properties of the system, we assess how resilient our approach is with respect to changes in its dependencies (the Statix Solver and PIE). After that, we analyze how potential users would experience the system by assessing the amount of work that needs to implement our framework for a concrete language project. Moreover, we compare the performance of our runtime with the reference implementation in Spoofax 2. Based on the results of these evaluations, we propose several directions of improvement for our framework as well as for the Statix solver and PIE. Finally, we summarize our findings and conclude the chapter.

6.1 Correctness of the SCI Design Pattern

In this section, we will argue that the SCI pattern adheres to the criteria specified in section 2.4. While the first three criteria are guaranteed by the design of the Statix language and the SCI design pattern, we investigate several threats that could invalidate the *Correctness* criterion, and explain why the restriction on specifications we imposed prevents these from happening.

6.1.1 Derivability

The first criterion we posed was in section 2.4 that cross-language type-checking should be integrated in the type system of the language. By using an interface module to implement the type system, it is immediately and unambiguously derivable which definitions to expose to and potentially reference from other languages.

6.1.2 Editor/Build Support

The second criterion states that the specification of type systems should be usable for both compilation environment and editor services. This is guaranteed by the fact that Statix is designed to yield executable specifications, which can be embedded in any type of environment. Hence, type systems specified using this pattern can naturally be used for both compilation and editor analysis.

6.1.3 Loose Coupling

Third, we required that there may be no dependency between the type systems of two concrete languages. This is implemented by lifting all shared concepts of a type system into an interface. In this way, no direct dependency between type systems is required.

The main motivation for this criterion was guaranteeing $O(n)$ development effort for cross-language type systems. While actually validating this requirement would require more experimental analysis, it is reasonable to assume that the additional effort that must be invested in designing and implementing a good interface, discounted for the advantage of reusing the interface, is proportional to the development effort of the type system of a language this uses this pattern.

6.1.4 No Incorrect Results

Finally, the last proposed criterion was correctness. Meeting this requirement is not trivial. In this section, we discuss some restrictions we impose to ensure correctness.

The first usage scenario in which the design pattern and runtime should provide correct results is when a language is used in isolation. Correctness in this situation is guaranteed by the fact that a specification that uses an interface module is not different from a regular Statix specification, and hence is executed in isolation in a similar fashion.

Second, when multiple languages are used in a project, but no joined analysis is desired, accidental interaction must not occur. This criterion is met by the fact that the framework makes it explicitly configurable which languages should be analyzed together. When languages are not analyzed together, a different analysis context is used. Therefore, their intermediate results are not merged, and hence their scope graphs remain disjoint. Hence, no accidental interference in rule simplification or query resolution can occur.

Third, we need to ensure that no accidental interference can occur when languages are in the same context. Accidental interference can occur when a reference to a predicate, relation or label from a particular language resolves to a declaration from the specification of another language. In principle, this could occur when two languages declare constraints, labels or rules with the same name in a module with the same name. To prevent this, we qualify all declarations and references for labels, relations and constraints with the identifier of the fragment in which they are declared.

Additionally, when a language contributes rules to a constraint that is declared in an interface, some well-formedness conditions with respect to scope extensions, overlap detection and rule ordering may be violated. For that reason, we close constraint definitions in a fragment by rejecting specifications that add rules to a constraint from a different fragment. In the remainder of this section, we explain the well-formedness violations that this behavior prevents in more detail.

Scope Extensions. Regarding the scope extension permissions, consider the example in Figure 6.1. In this example, the `interface` module represents an interface fragment, and the other modules represent fragments of a language specification. The `make` predicate returns a fresh scope. The `ext` predicate is not implemented in the interface, but language `B` contains a rule that adds a declaration to its parameter. Language `A` contains a rule `foo` that combines the constraints of the interface, by calling `ext` with the result of `make`.

In subsection 5.1.1, we mentioned that a scope may only be extended with a declaration or a edge to another scope when it is either freshly created in the constraints, or passed to it from a direct argument. In the example in Figure 6.1, this restriction holds for the combination of the `interface` and `langA` modules. The scope in `foo` can not be extended because it is returned from another predicate, but neither are extensions found for the argument of `ext`. Hence this reference is considered fine. The same holds for the combination of the `interface` and `langB` modules. An extension for `ext` is recorded, but there is no invalid call to `ext`. Hence this pair of modules can be compiled together as well. However, when we would try to compile all three modules together, we get an error at the call to `ext` from `foo`, because it is detected that `ext` might extend its parameter, but `s` has no permission to be extended.

```
module interface

rules
  // Creates a new scope
  make: -> scope
  // Adds declaration to scope
  ext: scope

  make() = s :-
    new s.
```

```
module langA
imports interface

rules


  foo:

  foo() :- {s}
    s == make(),
    ext(s).
```

```
module langB
imports interface

signature
  relations
    rel: string

rules
  ext(s) :-
    !rel["x"] in s.
```

Figure 6.1: Extension Permission not composable

```
module interface

rules
  // "Extension point"
  typeEq: TYPE * TYPE
```

```
module langA
imports interface

rules
  typeEq(T, T) :- primitive(T).
```

```
module langB
imports interface

rules
  typeEq(T, T).
```

Figure 6.2: Overlap Detection not composable

This example shows that separately compiled rules cannot be composed trivially. The analysis result of a particular module can influence the result of another module, without having a direct dependency. However, the restriction that a rule cannot extend a constraint from another specification fragment prevents this situation, because the rule for `ext` in module `langB` extends the declaration from `interface`, which is in another fragment. Hence the specification of language B is rejected when it is loaded.

Overlapping Patterns. The example from the previous section only involves predicates with scope arguments. Hence it could be solved by only forbidding extension of rules with scope arguments. However, besides the fact that the type of an argument can not be retrieved at specification loading time, a similar situation regarding the overlapping pattern detection can occur for any type of argument. For instance, consider the example in Figure 6.2. Here, the interface defines a predicate `typeEq` without rules. Both languages define a rule for this predicate that indicates that similar terms denote equal types. However, language `A` limits this to primitive types only.

Again, the combination of the interface with either language `A` or `B` compiles fine, because there are no overlapping patterns. However, the combination of all three modules does have such a pattern. When the Solver run from the `analyzeProject` task would need to simplify `typeEq(INT(), INT())`, it would get stuck, because it is not possible to choose one of both rules.

With the restriction that rules cannot extend a constraint from another specification fragment, both `langA` and `langB` are rejected, because they define a rule that extends the `typeEq` predicate.

Interference by Precedence. Yet, the `RuleSet.getAllEquivalentRules` method, which is part of the Statix solver API, can be used to detect these situations easily. We could use this method to reject combinations of fragments that would result in rules with overlapping patterns. However, there is still another more intricate form of interference, that can not be detected statically nor at specification loading time in a trivial way. Consider the example in Figure 6.3. In this example, there is again a predicate `typeEq` that is extended by both languages. In language `A`, equal terms denote equal types, but language `B` does explicitly prohibit coercion for values with `INT()` type.

```
module interface

rules
  // "Extension point"
  typeEq: TYPE * TYPE
```

```
module langA
imports interface

rules
  typeEq(T, T).
```

```
module langB
imports interface

rules
  typeEq(INT(), _) :- false.
```

Figure 6.3: Precedence Interference

In this example, even when compiling all three modules together, there are no overlapping patterns. However, suppose that in a context with these languages the constraint `typeEq(INT(), INT())` must be simplified. Now, the rule that is chosen to simplify this constraint depends on the phase the analysis is in. When the constraint is simplified by the `solveFile` task for a file of language `A`, only the `typeEq(T, T)` rule, supplied by the specification of language `A`, is available in the specification, and is hence chosen. In that case, solving the constraint succeeds. However, when it is simplified in the final solver phase, executed by the `analyzeProject` task, the rule defined in the fragment of language `B` will be chosen, because it has higher priority.

This inconsistent behavior is rather unpredictable, and hence confusing and undesired. However, this type of interference can not be detected statically. Therefore we decided to over-approximate it by preventing fragments to define rules for constraints declared in another fragment. In this way, we guarantee that in any phase of the analysis pipeline, for any constraint reference, all rules are available.

No missed Results. This explanation shows that a query can not return results it was not intended to return. However, the converse must hold as well. A query should return all declarations that are reachable, even when their declaring constraints were defined in specifications of other languages. We argue the existing scheduling algorithm scales directly to multi-language analysis. This algorithm works as follows: all queries that cross file boundaries (and possibly language boundaries) do this either directly or indirectly through a declaration in the global scope. When solving file and project constraints partially (phase two and three), the solver is aware that other solvers may contribute declarations to the global scope as well, and hence it will never return results from queries that pass through the global scope. Instead, these constraints are delayed to phase four. As a result of this, all queries that start in scopes retrieved from queries that pass the global scope (e.g. with projection) are delayed as well, because its start scope can not be ground. When these queries are revisited in phase four, the solver knows all constraints that still require solving, and hence can schedule the queries correctly.

Scope extension analysis, the overlapping patterns check and the pattern precedence rules are the only aspects of the Statix well-formedness criteria that do whole-program analysis. In this section we explained the restrictions we imposed to ensure the results of all these analyses remain valid for composed specifications. Therefore, we are confident there are no other factors that could make composed specifications behave differently than specifications that were compiled together.

6.2 Composability

In this section, we assess the functional properties of our system. We will evaluate the capabilities of our system regarding the composition of type systems. After that, we reflect on the goal of the Multi-Language Programming environment we set out in section 2.2.

Recently, Leduc, Degueule, Wyk, and Combemale (2020) presented an analogy between the Expression Problem and the *Language Extension Problem* (LEP). They define this problem using five criteria:

> *Extensibility in both dimensions*: It should be possible to extend the syntax and adapt existing semantics accordingly. Furthermore, it should be possible to introduce new semantics on top of the existing syntax.
> *Strong static-type [sic] safety*: All semantics should be defined for all syntax.
> *No modification or duplication*: Existing language specifications and implementations should neither be modified nor duplicated.
> *Separate compilation*: Compiling a new language (e.g., syntactic extension or new semantics) should not encompass recompiling the original syntax or semantics.
> *Independent extensibility*: It should be possible to combine and used [sic] jointly language extensions (syntax or semantics) independently developed.

Although there is a slight domain mismatch, we evaluate our solution using the last four of these criteria. We do not evaluate the *Extensibility in both dimensions* criterion, because we did not include syntactic extension in this thesis[1].

6.2.1 Type Safety

When *Strong Static Type Safety* is defined using the regular notions of progress and preservation (Wright and Felleisen 1994), this criterion holds for regular Statix specifications. The type system of Statix guarantees that variables are always unified with a term of the type that was inferred. Moreover, Statix has well-defined semantics for constraint simplification, even when there is no matching rule. In these cases the constraint will fail, which, from the Statix viewpoint, is just a part of the solver result.

However, the fact that Statix has a fall-back mechanism that ensures specification execution can not get stuck does not give much guarantees for the behavior of the combined specifications. Ultimately, users of multi-language analysis do not want to have errors due to incomplete combined specifications. Hence, we think this criterion is partially satisfied in our implementation.

First, because the using SCI design pattern does not involve extending syntax, behavior in this respect does not change. That is, a constraint on an AST node for which no inference rule is defined will fail regardless whether multi-language analysis is used or not.

Additionally, we can consider extending this criterion to operations on semantic terms, such as types. In our framework it is allowed to declare new constructors for sorts declared in another fragment. Hence, an operation (constraint) that was initially defined on all terms of a sort has now become partial. When such a term is exposed using a relation defined in an interface, it is even possible that queries from other languages return a term build using that constructor, and apply operations on it. These operations can fail, even when the operation was total with respect to constructors that were defined when the language was developed.

For example, recall the case study with Mod and Mini-SQL. The specification fragment for Mod extends the `TYPE` sort, declared in the interface, with a `REC : scope -> TYPE` constructor. When a record type is declared, queries from Mini-SQL could potentially resolve to that declaration. However, it cannot do anything useful with it, because the `REC` constructor was not visible when Mini-SQL was compiled.

However, this leaves a slight hole in strong type safety guarantees regular Statix specifications have. When the Mini-SQL specification contains a constructor declaration with

[1]In general, we think Spoofax matches this criterion quite well. SDF3 has features to extend existing syntax, and in Statix rules for a constraint can be added in separate modules as well. However, it is not clear how to introduce new semantics to existing syntax.

the same name, operations defined for that declaration could accidentally be applied on the remote term as well. When the constructor in question has the same arity, but different argument sorts (e.g. `REC : string -> TYPE`), it can occur that variables are unified with terms *from another type*, hence breaking the guarantees the type system provides in regular specifications. Because Statix has no guarantees about solving order, these values could propagate to other places in the specification, yielding undefined results.

In general we think the impact of this downside is rather small, because of the large number of coincident features that are required to actually trigger the behavior. Ideally, we would like to prevent this in the same way we prevented declaring rules for constraints from other fragments. However, the signatures of terms are not included in the compiled Statix specifications, and hence we can not perform this check when loading specifications. In section 6.7, we propose a strategy to solve these problems together.

6.2.2 Compilation

The No Duplication/Modification criterion states that it should not be necessary to copy or modify sources to implement composition. This criterion is currently not satisfied, because interface sources need to be copied to the projects of languages that implement these interfaces. Moreover, the copied modules are then reanalyzed and recompiled for each language implementing the interface as. This is necessary to validate the well-formedness of the specification. That means that Separate Compilation is not fully implemented. On the other hand, it is not required to fully recompile combinations of language specifications.

6.2.3 Independent Extensibility

Fifth, the Independent Extensibility criterion states that it must be possible to use separately defined extensions (which in our case are language fragments) together. We support it almost completely, because we can combine arbitrary type systems by merging their specifications. One might think that sharing an interface is a prerequisite for joined analysis, but we want to point out that it is only a requirement for *meaningful interaction*. Under the same consistency preconditions as for regular multi-language analysis, it is possible to analyze sources of languages in the same context, even when the type system specifications do not share an interface. This feature is useful in usage scenarios where a language uses multiple interfaces in its type system. In that case, other languages that do not interact directly can still be included in the same analysis context. For example, consider an extensible variant of Java that integrates with a query language and a template language. To leverage all possibilities for joined analysis, Java, the query language and the template language should be analyzed together. Our system supports this usage pattern, and the correctness guarantees discussed in section 6.1 ensure that no undesired interaction between the latter two will take place.

6.2.4 Glue Code

In addition to these five criteria inspired by the Expression Problem, the paper of Leduc, Degueule, Wyk, and Combemale (2020) raises the concern of *Automatic Composability*. That is, no 'glue code' should be required to compose extensions. Because composition is not performed by combining scope graphs, instead of passing data, this criterion is fully satisfied in our implementation as well.

6.3 Revisiting The Multi-Language Programming Environment

With this assesment of correctness and composability in place, we assess whether our runtime can be used to realize the Multi-Language Programming Environment we sketched in chapter 2. In section 6.1, we argued that our approach largely satisfies the criteria we posed in section 2.4, although we add some well-formedness restrictions to the type system specifications.

However, a significant restriction to fully realizing this goal is the fact that this framework is confined to languages for which the type system can be expressed in Statix. Currently, it is not clear how to define Hindley-Milner style polymorphic type inference, substructural type systems or dependent types in Statix. This restriction precludes large groups of languages and hence limits the generality of our solution.

Secondly, we do not think that the additional restrictions we impose to guarantee correctness of runtime composition precludes any languages a priori. The restriction that constraints can not be extended over fragment boundaries does not limit the support for individual languages, but only restrict the ways interfaces can model the interactions between languages. Whether this restriction makes particular interaction patterns impossible is not yet clear.

Finally, we do not yet support analysis over project boundaries. The reason for that is that Spoofax 3 does not yet have a notion of a project using a Spoofax 3 languages. Hence, we cannot retrieve dependencies, nor adjust our analysis accordingly. Nonetheless we think cross-project analysis should be possible by constructing edges from the global scope of the depending project to the global scope of the dependent project, but that is future work.

6.4 Durability

Our runtime uses Statix in a way that is on the verge of what it was intended to support. When Statix develops further, additional well-formedness criteria might be imposed. These criteria may either require adaption of our implementation to maintain the correctness guarantees, possibly imposing more restrictions on the design pattern, or in the worst case invalidate the approach completely. Due to this relation, it is difficult to estimate the required future maintenance effort and assess the durability of the implementation.

Lastly, this meta-component adds analysis infrastructure next to the existing runtime (which is integrated in Spoofax 3 as well). This imposes an additional maintenance burden, and might increase build times as well. In the future, we must evaluate whether the new runtime supports all use cases that the old runtime supports as well, and consider replacing it entirely.

6.5 Usability

From a language designer perspective, the framework itself requires minimal effort to use. When a language specification is compiled with multi-language analysis enabled, all relevant code and configuration is generated, and the new runtime is used in the Eclipse IDE. No further actions are required to use the framework. Presumably, the most effort when using multi-language analysis goes to defining a good interface, and distributing it properly.

On the other hand, Spoofax 3 does not yet support dynamic loading of languages. Therefore, testing multi-language interactions requires restarting the IDE with updated plugins after every edit, which is rather time-consuming.

Furthermore, until now, we have tested the framework only on small toy languages. To assess the usability of our framework better, a more extensive study on real-world examples is required.

6.6 Performance

We wanted to gain an impression on the performance of our runtime. We did not have the goal to be significantly faster than the Spoofax 2 runtime, but neither do we want that user experience regarding responsibility drops significantly when switching to our runtime. In order to validate this, we implemented some benchmarks.

6.6.1 Benchmark setup

As a language, we choose the Java implementation in Spoofax, because that is the only language that has a specification of reasonable size, and for which large enough code bases exist. The performance measurements are implemented using the Java Microbenchmark Harness (JMH)[2]. The benchmarks are executed on a Ubuntu 20.04.1 LTS PC with a Intel Core i7-4710MQ processor with four 2.5 GHz cores and eight threads, and 12 GB RAM available. For each run, a maximum of 10 GB memory was allocated. Per benchmark, we performed five warm-up iterations, and 20 measurement iterations.

	Files	LOC
Source set 1	1	2
Source set 2	26	1524
Source set 3	1	1424

Figure 6.4: Size of Benchmark Source sets

In the implementation of the runtimes, there are two important differences.

- The Multi-language Runtime is implemented in Java, while the Spoofax 2 runtime is implemented in Stratego.
- The Multi-language Runtime loads specifications using PIE tasks. Hence loaded specifications will be cached by the runtime. In constrast, the Spoofax 2 runtime loads specifications for every analysis.

For each of these differences, we want to measure the impact on the running times of full as well as incremental analyses, if appropriate. Ideally, we would like to measure the effect of the fact that Spoofax 2 executes phases two and three using a parallel stream, while Spoofax 3 is strictly sequential. However, there is no Statix specification available which does considerable local analysis and for which a codebase of considerable size is available.

We used two source sets for our benchmarks. The first project contains only an empty class. Using a small source set ensures that the solver times are negligible, and hence the runtime overhead can be measured better. The second source set consists of a set of MiniJava examples [3]. Since this source set contains rather many files, compared to the actual lines of code, it is a good project to assess the impact of having multiple files. To make that comparison more precise, we concatenated all files from source set two in source set three. The project sizes are summarized in Figure 6.4. The values in the lines of code (LOC) column are measured with tokei 12.04[4], where columns and blank lines are ignored.

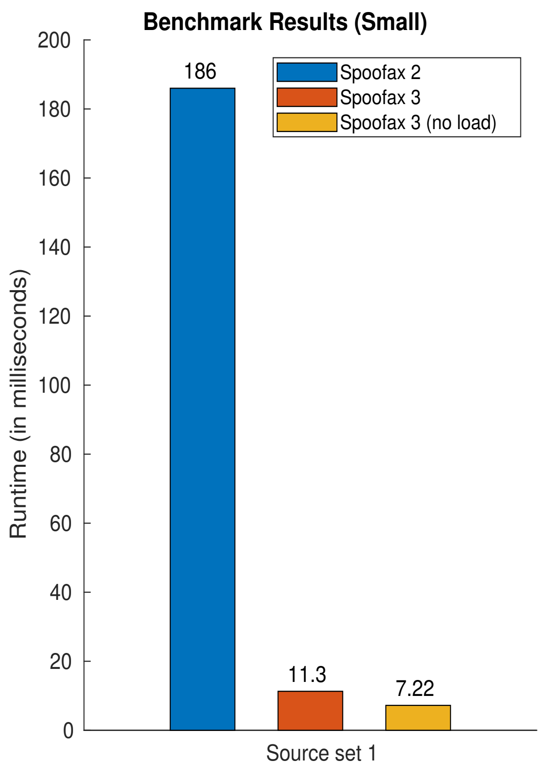
In order to benchmark the incremental analysis, we use five different change sets:

- A change set in which no file is marked as changed.
- A change set in which all files are marked as changed, but their contents remain identical (which could happen e.g. when a new branch is checked out).
- A change set in which only layout is changed.
- A change set in which a constant is changed in one file.
- A change set in which a constant is changed in all files.

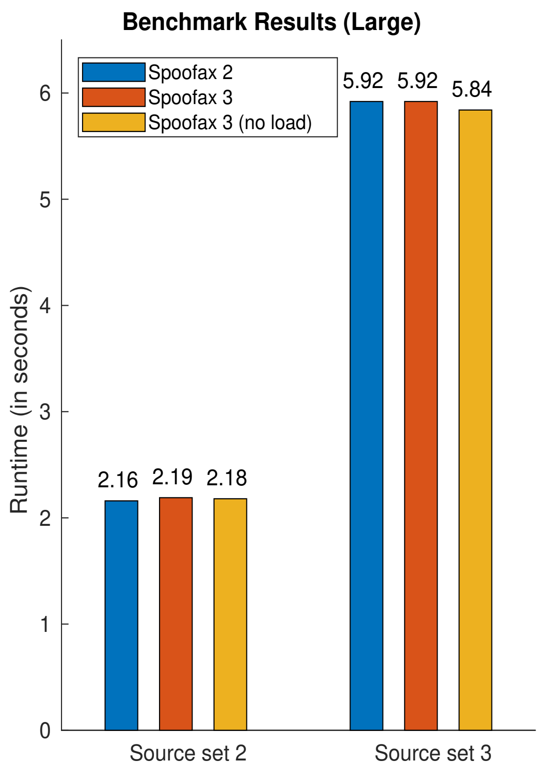
[2] `https://github.com/openjdk/jmh`

[3] `https://github.com/chrismwendt/MiniJava/tree/ff2c614d4dc9660ce054ad9f5100957eebe40ae9/examples`

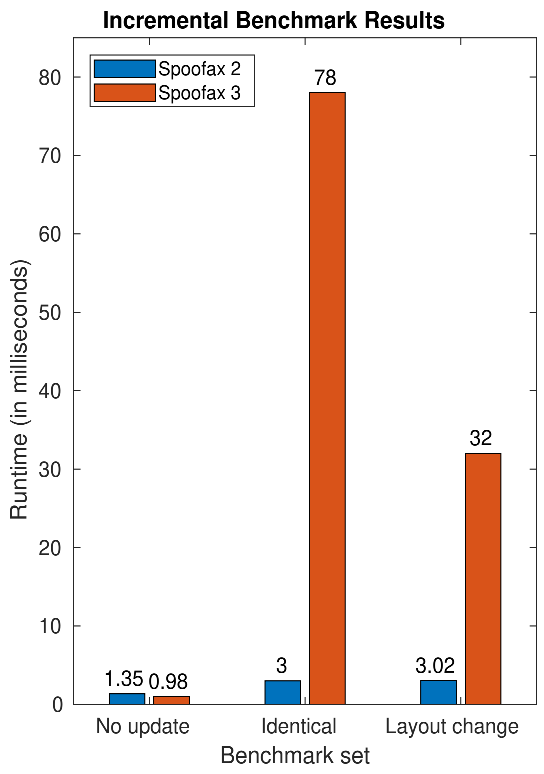
[4] `https://github.com/XAMPPRocky/tokei`



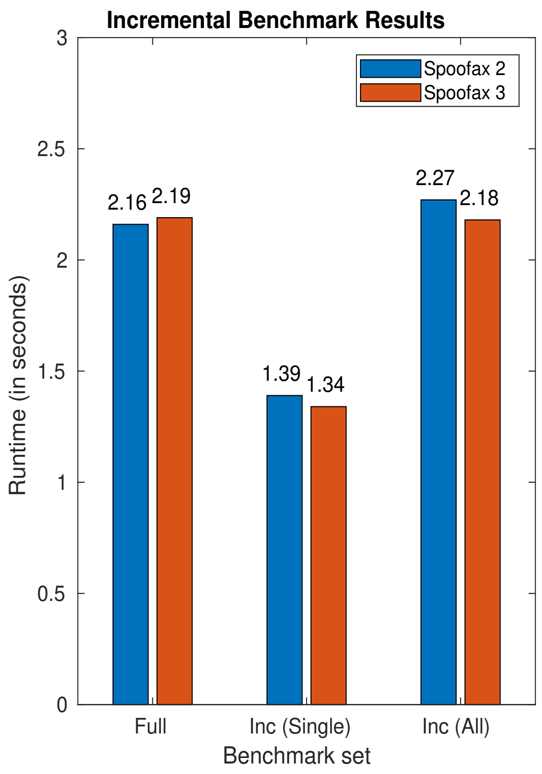
(a) Benchmark Results Source set 1



(b) Benchmark Results Source set 2 and 3



(c) Benchmark Results Incremental Analysis on Source set 2 (1)



(d) Benchmark Results Incremental Analysis on Source set 2 (2)

Figure 6.5: Benchmark Results

6.6.2 Results

The results of this benchmark are shown in Figure 6.5. Based on these results, we can draw the following conclusions.

In general, the overhead of the runtimes is insignificant for source sets of reasonable size. In Figure 6.5a, we see that the Spoofax 2 analysis takes roughly 15 times as long as the Spoofax 3 analysis. Analyzing the flame charts generated by our benchmark shows that this is mainly due to the overhead of the Stratego interpreter.

In Figure 6.5b, it is interesting to observe that the Spoofax 2 running times for single-file source sets are slightly larger than those of our runtime, while for the multi-file source set, this relation is reversed. Although this could be caused by Spoofax 2 solving the file constraints in parallel, we cannot rule out that this mismatch is just caused by measurement error.

In Figure 6.5c, we see that the Spoofax 2 runtime can detect that ASTs for changed files remain the same much faster than our runtime can. The reason for this behavior is that Spoofax 2 compares precomputed AST hashcodes, while the early cut-off algorithm of PIE does a structural comparison or the parser output, which contains a list of tokens and a set of parse messages as well as the AST.

Another remarkable observation is the big difference in performance between the 'identical' and the 'layout' change set, executed by the Spoofax 3 runtime. Moreover, the raw benchmark data shows that *no* PIE tasks are executed for the 'identical' change set, and 27 tasks for the 'layout' change set. The reason for this behavior as follows. When files are identical, the parser results will be identical as well. Again, the early cut-off feature of PIE requires comparison of these outputs, which takes considerable amount of time. In contrast to that, it is quite soon detected that parse outputs are different when the layout is changed. PIE will then continue executing the `index` tasks, and compare the returned AST with the previous results. These are equal again, and hence no more tasks are executed. Apparently, executing the AST indexing task and comparing its result is approximately twice as fast as comparing the complete parser output. Because the Spoofax 2 runtime does not compare complete parse outputs, but only precomputed AST hashcodes, it does not suffer from this problem.

Finally, in Figure 6.5d, we see that there is no significant difference in performance when a real change is made. When only one file in the source set is changed, we see that approximately 35% of the work could be reused. However, again no significant benefit of parallelizing file constraint solving can be observed. Therefore we believe that the execution time we save is mainly due to the fact we do not have to transform the ASTs of cached files. Finally, the incremental analysis poses a small overhead when all the files in a source set are changed.

Summarizing, we see that for tiny source sets, the execution times of the runtimes can differ significantly. However, when projects get larger, the solver execution time becomes the dominant factor. Therefore we expect neither decreased nor increased user experience when transitioning to our runtime.

6.7 Possible Improvements

Having established this quality assessment, we discuss two adaptations to the current Statix compiler that improve the current solution. After that, we propose adaptations to the Statix Solver and PIE that would improve the quality of our solution as well.

6.7.1 Improving Compilation

In this section, we discuss two major improvements to the current compilation procedure. First, Separate Compilation should remove the need to copy interface sources. When Separate Compilation is implemented, we can move the name qualification that is currently done at specification loading time to the compiler. Ultimately, that would allow relaxation of the restrictions on rule extension.

Separate Compilation. First, the current compilation procedure, as presented in subsection 5.3.3, should be improved. Currently, the sources of interface modules must be duplicated in each language project. Although it can be automated, it imposes an additional maintenance burden. In order to prevent source copying, the Statix specification format should be enriched to enable Separate Compilation. Furthermore, when the Statix compiler in Spoofax 3 matures, we should design and implement a way to distribute compiled specifications in such a way that they can be used as a source dependency in other languages. Related to this, we would like to distribute only the signatures of interfaces as compile-time dependency. In this way, we can enforce strict adherence to the conventions with respect to relation usage of an interface that are now implicit (as explained in section 4.2.3) by hiding the relations in the implementation fragment.

Language Identifier in Qualified Names. Secondly, while loading a specification, we prefix labels names, relations names and constraint names with the fragment identifier in which they are included. This prevents naming collisions between fragments that were not compiled together. However, generating fully qualified names should be done at compilation time, to prevent duplicate work at specification loading time.

However, hen we would implement that right now, our current compilation procedure would be invalidated, because the declarations in copied signature modules will be qualified differently. For example, consider an interface that defines a label `interface/base!P` in module `base`. When this module is copied to a language A, it will be analyzed again. In that analysis, `langA` will be prepended instead of `interface`, because the compiler is not aware of the fact these modules are actually copied. This mismatch will propagate to all references to `P` in the specification of language A. At runtime, the interface modules are loaded from the original interface language, and not from language A. This will result in a naming mismatch, causing queries to return incorrect results. Therefore, this change can only be performed after the separate compilation with enriched specifications is implemented, because that invalidates the need to copy sources, and hence avoids the incorrect qualification.

Additionally, when the compiler knows whether a constructor is declared in the current fragment or in a dependency, we can slightly relax the restriction that a fragment can not contribute rules to a constraint declared in another fragment. In particular, we can allow rules that:

- Have at least one constructor that is declared in the current fragment in their input pattern.
- Are linear.
- Have no overlap with the rules from the dependency.

These conditions guarantee that no runtime accidental overlap between sibling specification fragments can occur.

6.7.2 Improving Dependencies

Our Statix Runtime mainly relies on two components: The Statix Solver and PIE. Hence the runtime can piggy-back on improvements on these components as well. We envision three major approaches that could improve performance of analysis executed using our runtime.

Incremental Solver. When the Statix solver would get better support for concurrent and/or incremental analysis, we can increase the performance for our runtime as well. However, when the existing solver implementation would be replaced with a new solver implementation (as recent work on a concurrent solver is doing), we need to re-implement the part of the pipeline that executes the actual analysis. Furthermore, a different solver architecture may change the requirements and functional guarantees with regard to specification as well.

Parallel Task Execution. Moreover, when PIE would support parallel execution of tasks, the partial constraints could be solved in parallel, which can decrease the time required to transform the ASTs and execute phases two and three of an analysis. When this could be abstracted in the PIE Runtime, no adaption of the Multi-language Statix Runtime is required.

Improved Early Cut-off. Finally, in the current implementation, the post-transformation tasks for each file are executed after each time the analysis is executed. We implemented it this way to enable language implementers to define custom transformations that use the analysis result. To keep the input size small, the post-transformation task takes *a supplier* of the analyzed AST and analysis result as input. Hence the task is executed when the AST or the analysis result changes. Ideally, we would want an option to have only the latter, but in PIE, it is not possible to do early cut-off when a task does not use a part of its input. As a matter of fact, this behavior caused the strange benchmarking result with the 'identical' change set as well. We think this could be solved when PIE would support requiring a task with a custom equality comparing function.

6.8 Conclusion

In conclusion, we have seen that our framework has quite good properties with respect to composition. Most of the limitations are due to lacking features in the infrastructure, rather than algorithmic limitations. With some restrictions in place, we explained that our implementation adheres to the criteria we proposed in section 2.4, and should therefore be capable to solve the problems we encountered during the analysis of alternative approaches in section 2.3. The most important remaining work is ensuring strong static type safety again and improving the compilation infrastructure. Moreover we could research possibilities to relax the newly introduced restrictions.

We are not the first to try to combine type systems of multiple languages. Therefore, before we conclude this thesis, we discuss related work on Type System Composition in the next chapter.

Chapter 7

Related Work

In this chapter, we discuss other work related to Type System Composition. Generally, there are numerous concrete implementations of static analysis tools/frameworks that in someway or another derive information from sources in different languages. Yet, all of these are either restricted to a particular language family or a particular domain (e.g. security or standards compliance). In contrast, we explicitly aimed for generality in this thesis.

As far as we are aware, there are no approaches to compose type system specifications for constraint-based type checkers yet. Neither are we aware of approaches that pursue language-parametric composition by cross-language referencing. However, for other approaches to type system composition are taken.

First, we discuss a classification of composition, proposed by Erdweg, Giarrusso, and Rendel (2012). After that, we discuss related work on Attribute Grammars. Finally, we compare our work with some more pragmatic approaches found in the literature.

7.1 Taxonomy

Erdweg, Giarrusso, and Rendel (2012) propose a taxonomy of language composition patterns. This taxonomy helps us to relate our work to the work of others. The types of composition they distinguish are:

- Language Extension (LE): adding a new syntactic construct, with accompanying semantics to a language.
- Language Restriction (LR): removing a construct from a language. This category can be seen as a subpattern of Language Extension.
- Language Unification (LU): integrating two equivalent language definitions together, possibly by using some additional code (glue code). An example of this pattern is the integration of JavaScript and HTML.
- Self-Extension (SE): defining an extension for a host language in the same host language.
- Extension Composition (EC): using several language extensions together simultaneously.

In this taxonomy, our work is in the Language Unification category. In particular, we compose *validation procedures*.

7.2 Attribute Grammars

Attribute Grammars (Knuth 1968) is a formalism in which context-free grammars are extended with attributes that can be defined on productions. In its initial conception, contained two types of attributes. Synthesized attributes calculate an attribute value based on the attributes of its subtrees, while inherited attributes are calculated using attributes of the parent and sibling nodes.

Before comparing the approaches, it must be observed that Attribute Grammars serve a different composition style than the framework presented in this thesis. While we combine analyses results of several files in different languages, Attribute Grammars provide means to transport information within an AST. Therefore, they are more suitable to implement composition (and extension) by syntax embedding than by cross-language referencing. In particular, extensions of the original concept have been explored for Language Extension and Extension Composition (Ekman and Hedin 2007a; Kaminski and Wyk 2012; Kaminski, Kramer, Carlson, and Wyk 2017; Mernik 2013). Assuming that composition of languages is roughly the same as composing extensions of an empty language, attribute grammars are promising for language composition as well.

In general, attributes are defined using small functions. Hence, type system implementations such for languages usually do not resemble the formal specification as precise as Statix does. Hence our approach stays closer to the goal of closing the gap between specification and implementation than attribute grammar based approaches do. On the other hand, attribute grammar systems can be used to create type systems for a broader range of languages. Especially, a subset of Haskell, including polymorphic type inference is implemented using Attribute Grammars Dijkstra and Swierstra (2004).

First, JastAdd (Ekman and Hedin 2007b) is a system that extends the attribute grammar formalism with several other types of attributes:

- Attributes referencing other AST nodes, which make the AST more like a graph (Hedin 2000).
- Decoupling parent nodes from child nodes and vice versa by broadcasting attributes and parameterized attributes.
- Automatic rewriting interleaved with reading attributes.
- Adding generated nodes (non-terminal nodes) to an AST.
- Circular attributes implemented using fixpoint calculation.

By using these extensions to the attribute grammar formalism, JastAdd aims to have the AST as the only data structure in the compiler pipeline. This system has successfully been used to implement Java 1.5 using extensions on a Java 1.4 definition (Ekman and Hedin 2007a).

Due to these additional complications, well-formedness for JastAdd specifications can not be guaranteed statically anymore. Hence JastAdd has runtime consistency checks that validate some consistency requirements, but leaves others unchecked. Examples of such unchecked properties are confluence and termination, both of which Statix guarantees.

Secondly, Silver (Wyk, Bodin, Gao, and Krishnan 2010) uses an Attribute Grammar implementation extended with forwarding (Wyk, Moor, Backhouse, and Kwiatkowski 2002), higher-order attributes (Vogt, Swierstra, and Kuiper 1989) and reference attributes (Hedin 2000). Forwarding entails that a new production can specify a term it 'forwards to'. When an attribute on that new production must be calculated, but no definition is provided, it is calculated using the definition provided by term it forwards to. Additionally, unless specified otherwise, inherited attributes are by default copied to all child nodes. Unlike JastAdd, language specifications in Silver are still strongly typed. Additionally, Silver is one of the few attribute grammar systems with separate compilation. Silver has been used to implement AbleC (Kaminski, Kramer, Carlson, and Wyk 2017) and AbleJ (Wyk, Krishnan, Bodin, and Schwerdfeger 2007), which are implementations of respectively the C11 standard and Java 1.4. For both languages, a considerable amount of reliably composable extensions have been developed.

Third, LISA (Mernik 2013) is a system that supports inheriting multiple attribute grammars. In this way, all composition patterns from Erdweg et al. are supported. Unlike our approach, LISA requires some glue code to compose languages. Therefore, in contrast to our system, it can not be used to compose mutually agnostic type systems automatically.

In the context of Forwarding Attribute Grammars (e.g. Silver), Kaminski and Wyk (2017) investigate the problem of *Interference*. Interference occurs when two extensions work correctly when used separately, but cause invalid behavior when used together. By adding some restrictions to the definition of a language extension, they manage to prove that composition of such extensions maintains the correctness guarantees each extension has. This work actually inspired us to formulate our correctness condition in section 2.4, and to thoroughly assess the risk of interference in our solution. Although they operate in a different context, both of our approaches need to impose some restrictions to the original system to ensure correct behavior in any composition settings. However, we had to impose some restrictions for operational reasons as well rather than just declarative ones. On the other hand, we actually enforce the imposed restrictions at runtime, while Kaminski and Van Wyk provide a randomized testing approach to validate whether the restrictions they impose hold. We both claim that the imposed restrictions still allow a useful class of languages or language extensions to be composed.

7.3 Other Approaches

Other approaches to Multi-language analysis have been pursued as well. For example, Strein, Kratz, and Löwe (2006) define a *common model*, which is defined as "an integrative representation of a whole mixed-language program including cross-language relations". This model is populated by frontends for particular languages, and in return provides high-level analysis and some refactoring options.

Comparing to our approach, it seems that this common meta-model fulfills a role similar to our interfaces. They write: "[The common meta-model includes only] those language concepts, that are relevant to higher level analyses or to other languages." Like our approach, this framework allows cross-file analysis as well. Moreover, their approach seems rather demand-driven, whereas we achieved a much stronger foundation in type theory because we use specifications written in Statix. For that reason, we can provide well articulated assessments of composability and correctness. On the other hand, their system has stronger experimental validation. It has been used to integrate several web languages (HTML, XML and JavaScript) and server-side languages (JSP, Java, ASP, C#, Visual Basic and J#) in a single model.

Finally, Pfeiffer and Wasowski (2012b) present TexMo, which is a Development Environment especially designed for Multi-Language applications. TexMo provides a universal language representation, which can be used to represent and manipulate sources of 75 different GPLs and DSLs. The static analysis of TexMo is concentrated on heuristic-based reference resolution rather than type-checking. Therefore our system provides much stronger guarantees regarding completeness and correctness. Furthermore, it is not clear what range of language classes TexMo supports.

A case study using TexMo (Pfeiffer and Wasowski 2012a) shows that using *Cross-Language Support* mechanisms, including static analysis, navigation and refactoring, significantly aid developers maintaining large Multi-language code-bases. This result is a strong motivation for all work on Multi-language Analysis.

Chapter 8

Conclusion

A Multi-Language Programming Environment is an environment that helps a developer to understand and manipulate a codebase that contains sources in multiple programming languages. To provide these services, the information that static analysis of these sources provides must be processed in an integrated fashion. The integration of the information must be consistent with the formal semantics of the languages, available for editors as well as build tools, not require tight coupling nor glue code, and deliver results that are consistent with the actual interaction pattern.

Type Systems that comply with all these criteria can be specified in Statix using the Shared Concept Interface (SCI) design pattern. In this pattern, the semantic concepts that should be shared across languages are encoded in a separate interface language. Concrete languages depend on these interfaces to specify their own type systems. In this way, a language designer can regulate the information that is exposed and retrieved precisely.

We provide a runtime that can load and execute composed specifications, and integrated it in Spoofax 3. This runtime is implemented using a pipeline of PIE tasks, which ensures it has the same level of incrementality as the reference implementation in Spoofax 2. The Spoofax 3 compiler is adapted to generate all boilerplate code for languages that support multi-language analysis, which makes the framework easy to use. The framework is platform-agnostic, and an embedding in the Eclipse IDE is provided. Finally, the project-specific configuration options give the user fine-grained control regarding the sources that should be analyzed together.

In order to maintain correctness guarantees for combined specifications, we imposed the restriction that a predicate definition, including its rules, must be contained in a single language project.

The proposed design pattern and the corresponding implementation are validated with two case studies. First, we integrated a subset of SDF3 and a subset of Stratego. This integration allows to check whether terms in Stratego rules were well-formed with respect to a signature derived from an SDF3 syntax specification. Second, we integrated a subset of SQL with a toy language that features expressions, records, modules and functions. In this integration, table definitions can be interpreted as record types, and stored procedures as functions. Both case studies were implemented successfully, demonstrating that the approach is actually working.

8.1 Future Work

Multi-Language Type System Composition is a broad and lively field of research. Hence we hope this work will inspire a lot of other research in this area. Especially, we see several opportunities to use heterogeneous scope graphs for other, currently unsupported, composition patterns. Besides that, there are several research directions and technical improvements that could improve our current solution.

In particular, there are forms of composition that the system this thesis presents does not yet cover. First, we believe heterogeneous scope graphs can be used to type-check programs that use syntactic embedding as well. In that case, the scope graph of the object program will be a subgraph of the enclosing scope in the host language. Similarly, references from anti-quotations can be resolved through the scope graph of the object program.

To further leverage the flexibility of heterogeneous scope graphs, we should consider adding support for extension composition, similar to the pattern many Attribute Grammar systems support. When we can relax the restrictions on rule composition as described in section 6.7 and introduce 'extension fragments' that provide a specification fragment that extends the base language, we could implement meaningful language extensions.

Third, we do not yet support type-checking across project boundaries. Again, we think this could be implemented by using heterogeneous scope graphs. In this case, an edge from the global scope of the dependent project to the global scope of the dependency should be added.

Throughout chapter 5 and chapter 6, we encountered several properties of the Statix compiler that prevented correct composition in general. This problem is caused by the fact that Statix has several whole-program analyses. To remediate these problems, we should adapt the compiler to comply with the *Separate Compilation* criterion of Leduc, Degueule, Wyk, and Combemale (2020).

Fifth, we should investigate if we can use the findings on the composability of Statix to create distributable language component libraries, in the style that Butting, Reikermannobert, Hölldobler, Jansen, Rumpe, and Wortmann (2020) recently presented. This would prevent repeated reimplementation of the basic aspects of a type system, making language engineering more efficient.

Finally, we have only verified our solution on small toy languages. To get a better evaluation of the possibilities and limitations of our approach, case studies on real-world languages should be performed.

Bibliography

Aerts, Taico (2019). “Incrementalizing Statix: A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs”. MA thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:3e0ea516-3058-4b8c-bfb6-5e846c4bd982 (visited on 11/12/2020).

Antwerpen, Hendrik van, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2016). “A constraint language for static semantic analysis based on scope graphs”. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. ISBN: 978-1-4503-4097-7. DOI: 10.1145/2847538.2847543. URL: http://doi.acm.org/10.1145/2847538.2847543.

Antwerpen, Hendrik van, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). “Scopes as types”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

Anwer, S., A. Adbellatif, M. Alshayeb, and M. S. Anjum (Mar. 2017). “Effect of coupling on software faults: An empirical study”. In: *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, pp. 211–215. DOI: 10.1109/C-CODE.2017.7918930.

Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser (2008). “Stratego/XT 0.17. A language and toolset for program transformation”. In: *Science of Computer Programming* 72.1-2, pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: http://dx.doi.org/10.1016/j.scico.2007.11.003.

Butting, Arvid, Robert Reikermannobert, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann (2020). “A Library of Literals, Expressions, Types, and Statements for Compositional Language Design”. In: *Journal of Object Technology* 19.3. DOI: 10.5381/jot.2020.19.3.a4. URL: https://doi.org/10.5381/jot.2020.19.3.a4.

Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler (2008). “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, pp. 209–224. ISBN: 978-1-931971-65-2. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

Cartwright, Robert and Mike Fagan (1991). “Soft Typing”. In: *Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. Ed. by David S. Wise. ACM, pp. 278–292. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113469.

Chang, Stephen, Alex Knauth, and Ben Greenman (2017). “Type systems as macros”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D.

Gordon. ACM, pp. 694–705. ISBN: 978-1-4503-4660-3. URL: http://dl.acm.org/citation.cfm?id=3009886.

Codd, E. F. (1970). “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6, pp. 377–387.

Cousot, Patrick and Radhia Cousot (1977). “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*, pp. 238–252.

Dijkstra, Atze and S. Doaitse Swierstra (2004). “Typing Haskell with an Attribute Grammar”. In: *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*. Ed. by Varmo Vene and Tarmo Uustalu. Vol. 3622. Lecture Notes in Computer Science. Springer, pp. 1–72. ISBN: 3-540-28540-7. DOI: 10.1007/11546382_1. URL: http://dx.doi.org/10.1007/11546382_1.

Ekman, Torbjörn and Görel Hedin (2007a). “The JastAdd extensible Java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297029. URL: http://doi.acm.org/10.1145/1297027.1297029.

— (2007b). “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3, pp. 14–26. DOI: 10.1016/j.scico.2007.02.003. URL: http://dx.doi.org/10.1016/j.scico.2007.02.003.

Erdweg, Sebastian, Paolo G. Giarrusso, and Tillmann Rendel (2012). “Language composition untangled”. In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, p. 7. ISBN: 978-1-4503-1536-4. DOI: 10.1145/2427048.2427055. URL: http://doi.acm.org/10.1145/2427048.2427055.

Fenton, Norman and James Bieman (2014). *Software metrics: a rigorous and practical approach.* 3rd ed. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. CRC Press.

García-Munoz, Javier, Marisol García-Valls, and Julio Escribano-Barreno (2016). “Improved Metrics Handling in SonarQube for Software Quality Monitoring”. In: *Distributed Computing and Artificial Intelligence, 13th International Conference, DCAI 2016, Sevilla, Spain, 1-3 June, 2016*. Ed. by Sigeru Omatu et al. Vol. 474. Advances in Intelligent Systems and Computing. Springer, pp. 463–470. ISBN: 978-3-319-40161-4. DOI: 10.1007/978-3-319-40162-1_50. URL: http://dx.doi.org/10.1007/978-3-319-40162-1_50.

Groenewegen, Danny M., Zef Hemel, Lennart C. L. Kats, and Eelco Visser (2008). “WebDSL: a domain-specific language for dynamic web applications”. In: *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*. Ed. by Gail E. Harris. ACM, pp. 779–780. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449858. URL: http://doi.acm.org/10.1145/1449814.1449858.

Hedin, Görel (2000). “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3, pp. 301–317.

Hemel, Zef, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser (2011). “Static consistency checking of web applications with WebDSL”. In: *Journal of Symbolic Computation* 46.2, pp. 150–182. DOI: 10.1016/j.jsc.2010.08.006. URL: https://doi.org/10.1016/j.jsc.2010.08.006.

Kaminski, Ted, Lucas Kramer, Travis Carlson, and Eric Van Wyk (2017). “Reliable and automatic composition of language extensions to C: the ableC extensible language framework”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA. DOI: 10.1145/3138224. URL: http://doi.acm.org/10.1145/3138224.

Kaminski, Ted and Eric Van Wyk (2012). “Modular Well-Definedness Analysis for Attribute Grammars”. In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, pp. 352–371. ISBN: 978-3-642-36089-3. DOI: 10.1007/978-3-642-36089-3_20. URL: http://dx.doi.org/10.1007/978-3-642-36089-3_20.

— (2017). “Ensuring non-interference of composable language extensions”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, pp. 163–174. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136023. URL: http://doi.acm.org/10.1145/3136014.3136023.

Karakoidas, Vassilios, Dimitris Mitropoulos, Panagiotis Louridas, and Diomidis Spinellis (2015). “A type-safe embedding of SQL into Java using the extensible compiler framework J%”. In: *Computer Languages, Systems & Structures* 41, pp. 1–20. DOI: 10.1016/j.cl.2015.01.001. URL: http://dx.doi.org/10.1016/j.cl.2015.01.001.

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

Khedker, Uday P., Amitabha Sanyal, and Bageshri Sathe (2009). *Data Flow Analysis - Theory and Practice*. CRC Press. ISBN: 978-0-8493-2880-0. URL: http://www.crcpress.com/product/isbn/9780849328800.

Knuth, Donald E. (1968). “Semantics of Context-Free Languages”. In: *Theory Comput. Syst.* 2.2, pp. 127–145. URL: http://www.springerlink.com/content/m2501m07m4666813/.

Konat, Gabriel (2020). *Spoofax 3*. URL: https://metaborg.github.io/spoofax-pie/ (visited on 11/06/2020).

Konat, Gabriël, Sebastian Erdweg, and Eelco Visser (2018). “Scalable incremental building with dynamic task dependencies”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, pp. 76–86. DOI: 10.1145/3238147.3238196. URL: https://doi.org/10.1145/3238147.3238196.

Konat, Gabriël, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser (2018). “PIE: A Domain-Specific Language for Interactive Software Development Pipelines”. In: *Programming Journal* 2.3, p. 9. DOI: 10.22152/programming-journal. URL: https://doi.org/10.22152/programming-journal.org/2018/2/9.

Kullbach, Bernt, Andreas Winter, Peter Dahm, and Jürgen Ebert (Oct. 1998). “Program Comprehension in Multi-Language Systems”. In: *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*. Honolulu, HI, USA, pp. 135–143. DOI: 10.1109/WCRE.1998.723183.

Leduc, Manuel, Thomas Degueule, Eric Van Wyk, and Benoît Combemale (2020). “The Software Language Extension Problem”. In: *Software and Systems Modeling* 19.2, pp. 263–267. DOI: 10.1007/s10270-019-00772-7. URL: https://doi.org/10.1007/s10270-019-00772-7.

Long, Fan and Martin Rinard (2015). “Staged program repair with condition synthesis”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, pp. 166–178. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786811. URL: http://doi.acm.org/10.1145/2786805.2786811.

Mayer, Philip, Michael Kirsch, and Minh Anh Le (2017). “On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers". In: *J. Software Eng. R&D* 5, p. 1. doi: 10.1186/s40411-017-0035-z. url: https://doi.org/10.1186/s40411-017-0035-z.

Meijer, Erik and Peter Drayton (Jan. 2004). "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages". In: url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.394.3818&rep=rep1&type=pdf (visited on 11/05/2020).

Mernik, Marjan (2013). "An object-oriented approach to language compositions for software language engineering". In: *Journal of Systems and Software* 86.9, pp. 2451–2464. doi: 10.1016/j.jss.2013.04.087. url: http://dx.doi.org/10.1016/j.jss.2013.04.087.

Nagy, Csaba and Anthony Cleve (2018). "SQLInspect: a static analyzer to inspect database usage in Java applications". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman. ACM, pp. 93–96. doi: 10.1145/3183440.3183496. url: http://doi.acm.org/10.1145/3183440.3183496.

Néron, Pierre, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2015). "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 205–231. isbn: 978-3-662-46668-1. doi: 10.1007/978-3-662-46669-8_9. url: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

Pfeiffer, Rolf-Helge and Andrzej Wasowski (2011). "Taming the Confusion of Languages". In: *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*. Ed. by Robert B. France, Jochen Malte Kuester, Behzad Bordbar, and Richard F. Paige. Vol. 6698. Lecture Notes in Computer Science. Springer, pp. 312–328. isbn: 978-3-642-21469-1. doi: 10.1007/978-3-642-21470-7_22. url: http://dx.doi.org/10.1007/978-3-642-21470-7_22.

— (2012a). "Cross-Language Support Mechanisms Significantly Aid Software Development". In: *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Vol. 7590. Lecture Notes in Computer Science. Springer, pp. 168–184. isbn: 978-3-642-33665-2. doi: 10.1007/978-3-642-33666-9_12. url: http://dx.doi.org/10.1007/978-3-642-33666-9_12.

— (2012b). "TexMo: A Multi-language Development Environment". In: *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*. Ed. by Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos. Vol. 7349. Lecture Notes in Computer Science. Springer, pp. 178–193. isbn: 978-3-642-31490-2. doi: 10.1007/978-3-642-31491-9_15. url: http://dx.doi.org/10.1007/978-3-642-31491-9_15.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. Cambridge, Massachusetts: MIT Press.

Rouvoet, Arjen, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020). "Knowing When to Ask. Sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA. doi: 10.1145/3428248. url: https://doi.org/10.1145/3428248.

Siek, Jeremy G. and Walid Taha (2007). "Gradual Typing for Objects". In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, pp. 2–27. isbn: 978-3-540-73588-5. doi: 10.1007/978-3-540-73589-2_2. url: http://dx.doi.org/10.1007/978-3-540-73589-2_2.

Souza Amorim, Luís Eduardo de (2019). "Declarative Syntax Definition for Modern Language Workbenches". base-search.net (fttudelft:oai:tudelft.nl:uuid:43d7992a-7077-47bab38f-113f5011d07f). PhD thesis. Delft University of Technology, Netherlands. URL: https://www.base-search.net/Record/261b6c9463c1d4fe309e3c6104cd4d80fbc9d3cc8fbc66006f34130f481b506f.

Souza Amorim, Luís Eduardo de and Eelco Visser (2020). "Multi-purpose Syntax Definition with SDF3". In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Ed. by Frank S. de Boer and Antonio Cerone. Vol. 12310. Lecture Notes in Computer Science. Springer, pp. 1–23. ISBN: 978-3-030-58768-0. DOI: 10.1007/978-3-030-58768-0_1. URL: https://doi.org/10.1007/978-3-030-58768-0_1.

Strein, Dennis, Hans Kratz, and Welf Löwe (2006). "Cross-Language Program Analysis and Refactoring". In: *Source Code Analysis and Manipulation, IEEE International Workshop on* 0. DOI: 10.1109/SCAM.2006.10. URL: http://doi.ieeecomputersociety.org/10.1109/SCAM.2006.10.

Tanenbaum, Andrew S. (1978). "A Comparison of PASCAL and ALGOL 68". In: *Computer Journal* 21.4, pp. 316–323.

Vogt, Harald, S. Doaitse Swierstra, and Matthijs F. Kuiper (1989). "Higher-Order Attribute Grammars". In: *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. Ed. by Richard L. Wexelblat. ACM, pp. 131–145. ISBN: 0-89791-306-X.

Vollebregt, Tobi, Lennart C. L. Kats, and Eelco Visser (2012). "Declarative specification of template-based textual editors". In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, pp. 1–7. ISBN: 978-1-4503-1536-4. DOI: 10.1145/2427048.2427056. URL: http://doi.acm.org/10.1145/2427048.2427056.

Wright, Andrew K. and Matthias Felleisen (Nov. 1994). "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1, pp. 38–94. DOI: 10.1006/inco.1994.1093.

Wyk, Eric Van, Derek Bodin, Jimin Gao, and Lijesh Krishnan (2010). "Silver: An extensible attribute grammar system". In: *Science of Computer Programming* 75.1-2, pp. 39–54. DOI: 10.1016/j.scico.2009.07.004. URL: http://dx.doi.org/10.1016/j.scico.2009.07.004.

Wyk, Eric Van, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger (2007). "Attribute Grammar-Based Language Extensions for Java". In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, pp. 575–599. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2_27. URL: http://dx.doi.org/10.1007/978-3-540-73589-2_27.

Wyk, Eric Van, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski (2002). "Forwarding in Attribute Grammars for Modular Language Design". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, pp. 128–142. ISBN: 3-540-43369-4. URL: http://link.springer.de/link/service/series/0558/bibs/2304/23040128.htm.

Yourdon, Edward and Larry L. Constantine (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. 1st. USA: Prentice-Hall, Inc. ISBN: 0138544719.

Appendix A

Statix Specification of Mini–SDF and Mini–STR Case Study

In this appendix, we include the Statix specification of the Mini–SDF and Mini–STR case study.

A.1 Interface

```
module abstract-sig/types

signature

  sorts TAG = scope

  sorts SORT constructors
    SORT                   : TAG * string -> SORT

  sorts TYPE constructors
    SINGLE                 : SORT -> TYPE
    OPT                    : SORT -> TYPE
    ITER                   : SORT -> TYPE
    STAR                   : SORT -> TYPE

  sorts CONS constructors
    CONS          : SORT * list(TYPE) * TAG -> CONS

  name-resolution
    labels
      P // lexical parent
      I // Module import
```

```
module abstract-sig/sorts

imports
  abstract-sig/types

signature

  relations
    sort: string -> SORT

rules

  declareSort: scope * string

  declareSort(s, n) :- {stag}
    new stag,
    !sort[n, SORT(stag, n)] in s,
    // Check for sorts with same name in same scope
    query sort
      filter e and { n' :- n' == n }
      in s |-> [_] | error $[Duplicate declaration of sort [n]],
    // Check for sorts with same name in imported modules
    query sort
      filter P* I* & ~e and { n' :- n' == n }
      in s |-> [] | error $[Shadowing imported sort [n]].

rules

  sortOfSort: scope * string -> SORT
  sortOfSort(s, n) = T :-
    query sort
      filter P* I* and { n' :- n' == n }
         min $ < P, $ < I, P < I
          in s |-> [(_, (_, T)) | _]
    | error $[Sort [n] not declared].
```

```
module abstract-sig/constructors

imports
  abstract-sig/types

signature

  relations
    // Store arity in relation arguments
    cons: string * int -> CONS

rules

  declareCons : scope * SORT * string * list(TYPE)

  declareCons(s, T, n, S) :- {a ctag}
    new ctag,
    a == arityOfSig(S),
    !cons[n, a, CONS(T, S, ctag)] in s,
    // Check for constructors with same name in same scope
    query cons
      // In this data well-formedness predicate, both the name and arity should match
      filter e and { t :- t == (n, a) }
      in s |-> [_]
    | error $[Duplicate declaration of constructor [n]/[a].],
    // Check for constructors with same name in imported modules
    query cons
      filter P* I* & ~e and { t :- t == (n, a) }
         min $ < P, $ < I, P < I
          in s |-> []
    | error $[Shadowing imported constructor [n]/[a].].

rules

  resolveCons : scope * int * string -> CONS

  resolveCons(s, a, n) = C :-
    query cons
      filter P* I* and { t :- t == (n, a) }
      in s |-> [(_, (_, _, C))| _]
    | error $[Constructor [n]/[a] not declared].

rules

  arityOfSig : list(TYPE) -> int

  arityOfSig([]) = 0.
  arityOfSig([_]) = 1.
  arityOfSig([h|t]) = res :- {ts}
    ts == arityOfSig(t),
    res #= ts + 1.
```

```
module abstract-sig/conflicts/sorts

imports
  abstract-sig/types
  abstract-sig/sorts

signature

  sorts STI = (string * TAG) // Sort type info

rules

  sortsUnique: scope * scope

  sortsUnique(s_imp, s_mod) :- {SM SI}
    sortsInScope(s_mod) == SM, // Sorts in imported module scope
    importedSorts(s_imp) == SI, // Sorts imported in importing module scope
    sortsDisjoint(SM, SI).

rules

  sortsInScope : scope -> list(STI)

  sortsInScope(s) = typeInfosOfSorts(S) :-
    query sort
      filter P* I*
      in s |-> S.

rules

  importedSorts : scope -> list(STI)

  importedSorts(s) = typeInfosOfSorts(S) :-
    query sort
      filter P* I* & ~e
      in s |-> S.

rules

  typeInfosOfSorts maps typeInfoOfSort(list(*)) = list(*)
  typeInfoOfSort: (path * (string * SORT)) -> STI

  typeInfoOfSort((_, (n, SORT(id, _)))) = (n, id).


rules
  // Double maps to validate each pair
  sortsDisjoint maps sortDisjoint(list(*), *)
  sortDisjoint maps sortPairDisjoint(*, list(*))

  sortPairDisjoint: STI * STI

  sortPairDisjoint((n, id1), (n, id2)) :-
    id1 == id2 | error $[Duplicate import of sort [n]].
  sortPairDisjoint((n1, _), (n2, _)).
```

```
module abstract-sig/conflicts/constructors

imports
  abstract-sig/types
  abstract-sig/constructors

signature

  sorts CTI = (string * int * TAG) // Constructor type info

rules

  conssUnique: scope * scope

  conssUnique(s_imp, s_mod) :- {CM CI}
    conssInScope(s_mod) == CM, // Constructors in imported module scope
    importedConss(s_imp) == CI, // Constructors imported in importing module scope
    consDisjoint(CM, CI).

rules

  conssInScope : scope -> list(CTI)

  conssInScope(s) = typeInfosOfConss(C) :-
    query cons
      filter P* I*
      in s |-> C.

rules

  importedConss : scope -> list(CTI)

  importedConss(s) = typeInfosOfConss(C) :-
    query cons
      filter P* I* & ~e
      in s |-> C.

rules

  typeInfosOfConss maps typeInfoOfCons(list(*)) = list(*)
  typeInfoOfCons: (path * (string * int * CONS)) -> CTI

  typeInfoOfCons((_, (n, a, CONS(_, _, id)))) = (n, a, id).

rules
  // Double maps to validate each pair
  consDisjoint maps consDisjoint(list(*), *)
  consDisjoint maps consPairDisjoint(*, list(*))

  consPairDisjoint: CTI * CTI

  consPairDisjoint((n, a, id1), (n, a, id2)) :-
    id1 == id2 | error $[Duplicate import of cons [n]/[a]].
  consPairDisjoint((n1, _, _), (n2, _, _)).
```

```
module modules/modules

imports
  abstract-sig/types

  abstract-sig/conflicts/sorts
  abstract-sig/conflicts/constructors

signature

  sorts MODULE constructors
    MODULE : scope * ModuleType -> MODULE

  sorts ModuleType constructors
    SUPPLY : ModuleType
    CONSUME : ModuleType

  relations
    mod : string -> MODULE

rules

  // Rule used for validating unique import of sorts/conss/rules
  itemsOk: string * scope * scope

  itemsOk(_, s_imp, s_mod) :-
    sortsUnique(s_imp, s_mod),
    conssUnique(s_imp, s_mod).

rules

  declareMod: scope * string * scope * ModuleType

  declareMod(s, n, s_mod, T) :-
    !mod[n, MODULE(s_mod, T)] in s,
    // Validate module name unique
    query mod
       filter P* and { n' :- n' == n }
       in s |-> [_]
    | error $[Module [n] declared multiple times].

rules

  import: scope * string * ModuleType -> scope

  import(s_imp, n, T) = s_mod :- {T1}
    s_imp -I-> s_mod,
    query mod
       filter P* and {n' :- n' == n }
       in s_imp |-> [(_, (_, MODULE(s_mod, T1))) | _]
    | error $[Module [n] could not be found],
    // Validate module type combination is correct.
    T1 == T | error $[Module of type [T] cannot import module of type [T1]],
    // Validate no conflicts in imported constructs
    itemsOk(n, s_imp, s_mod).
```

A.2 Mini–SDF

```
module mini-sdf/sorts

imports
  signatures/minisdf-sig
  abstract-sig/types
  abstract-sig/sorts

rules

  sortsOk maps sortOk(*, list(*))

  sortOk: scope * ID
  sortOk(s, n) :-
    declareSort(s, n).
```

```
module mini-sdf/productions

imports
  abstract-sig/types
  abstract-sig/constructors
  abstract-sig/sorts

  signatures/minisdf-sig
  mini-sdf/sorts

rules

  prodsOk maps prodOk(*, list(*))

  prodOk: scope * Production
  prodOk(s, Production(sn, cn, t)) :- {T T1}
    sortOfSort(s, sn) == T,
    typeOfSymbols(s, prodTerms(t)) == T1,
    declareCons(s, T, cn, T1).

rules

  // Filter away Terminals, since they do not have a sort
  // (and hence don't need type-checking, or a position in the constructor signature)
  prodTerms: list(Symbol) -> list(Symbol)
  prodTerms([]) = [].
  prodTerms([Terminal(_) | tl]) = prodTerms(tl).
  prodTerms([s@Term(_) | tl]) = [s | prodTerms(tl)].

rules

  typeOfSymbols maps typeOfSymbol(*, list(*)) = list(*)

  typeOfSymbol: scope * Symbol -> TYPE
  typeOfSymbol(s, Term(t)) = typeOfTerm(s, t).

rules

  typeOfTerm : scope * Term -> TYPE

  typeOfTerm(s, Plus(t))       = ITER(sortOfSort(s, t)).
  typeOfTerm(s, Option(t))     = OPT(sortOfSort(s, t)).
  typeOfTerm(s, IterStar(t))   = STAR(sortOfSort(s, t)).
  typeOfTerm(s, Ref(t))        = SINGLE(sortOfSort(s, t)).
```

```
module mini-sdf/imports

imports
  modules/modules
  abstract-sig/sorts
  signatures/minisdf-sig

  abstract-sig/conflicts/sorts
  abstract-sig/conflicts/constructors

rules

  importsOk maps importOk(*, list(*))

  importOk : scope * MOD

  importOk(s, n) :-
    import(s, n, SUPPLY()) == _.
```

```
module mini-sdf

imports
  signatures/minisdf-sig

  abstract-sig/types
  modules/modules

  mini-sdf/sorts
  mini-sdf/productions
  mini-sdf/imports

rules

  projectOk : scope
  projectOk(_).

rules

  fileOk : scope * Start
  fileOk(s, Module(n, sec)) :- {s_mod}
    new s_mod,
    s_mod -P-> s,
    declareMod(s, n, s_mod, SUPPLY()),
    sectionsOK(s_mod, sec).

rules

  sectionsOK maps sectionOk(*, list(*))

  sectionOk : scope * Section

  sectionOk(s, SortsDecl(str)) :-
    sortsOk(s, str).
  sectionOk(s, ContextFreeSyntax(prd)) :-
    prodsOk(s, prd).
  sectionOk(s, ImportSection(i)) :-
    importsOk(s, i).
```

A.3 Mini–STR

```
module mini-str/signatures/sorts

imports
  abstract-sig/sorts
  abstract-sig/types

  signatures/ministr-sig

rules

  sortsOk maps sortOk(*, list(*))

  sortOk: scope * SMBL

  sortOk(s, n) :-
    // Declare sort in module root scope
    declareSort(s, n).
```

```
module mini-str/signatures/constructors

imports
  abstract-sig/types
  abstract-sig/sorts
  abstract-sig/constructors

  signatures/ministr-sig
  mini-str/signatures/sorts

rules

  conssOk maps consOk(*, list(*))

  consOk: scope * ConstructorDef

  consOk(s, NoArgs(cn, sn)) :- {T}
    sortOfSort(s, sn) == T,
    declareCons(s, T, cn, []).

  consOk(s, WithArgs(cn, p, sn)) :- {T T1}
    sortOfSort(s, sn) == T,
    sortsOfParams(s, p) == T1,
    declareCons(s, T, cn, T1).

rules

  sortsOfParams maps sortOfParam(*, list(*)) = list(*)

  sortOfParam: scope * ArgSort -> TYPE

  sortOfParam(s, Sort(n)) = SINGLE(sortOfSort(s, n)).
  sortOfParam(s, SOpt(n))  = OPT(sortOfSort(s, n)).
  sortOfParam(s, SIter(n)) = ITER(sortOfSort(s, n)).
  sortOfParam(s, SStar(n)) = STAR(sortOfSort(s, n)).
```

```
module mini-str/signatures/signatures

imports
  signatures/ministr-sig
  mini-str/signatures/sorts
  mini-str/signatures/constructors

rules

  signaturesOk maps signatureOk(*, list(*))

  signatureOk: scope * SignatureSection

  signatureOk(s_mod, Sorts(s)) :-
    sortsOk(s_mod, s).

  signatureOk(s_mod, Constructors(c)) :-
    conssOk(s_mod, c).
```

```
module mini-str/rules/resolution

imports
  abstract-sig/types
  abstract-sig/sorts
  mini-str/rules/list-sorts
  mini-str/rules/sorts-names

signature

  sorts RTDECL = (path * (string * RULE))
  sorts RULE constructors
    RULE : TYPE * TYPE -> RULE // Rule type: input sort => output sort

  name-resolution
    labels S

  relations
    ruleInst: string -> RULE // Declaration of single rewrite rule
    rule: string -> RULE     // Module-unique declaration of rule

rules
  // Declares a single rule instance in a scope
  declareRule : scope * scope * string * TYPE * TYPE
  // Constraint declaring rule which respects all earlier declared rules in its typing
  declRule: scope * scope * list(RTDECL) * list(RTDECL) * string * RULE
  equitype : RULE * RULE

  declareRule(s, s_seq, n, Tin, Tout) :- {rls rts rits}
    query rule
      filter P* I* & ~e and { n' :- n' == n }
      in s |-> rts,
    query ruleInst
      filter S+ and { n' :- n' == n }
      in s_seq |-> rits,
    declRule(s, s_seq, rts, rits, n, RULE(Tin, Tout)).

  // Rule definition is valid when no parent rule specified
  declRule(s_mod, s_seq, [], [], n, T) :-
    !rule[n, T] in s_mod,
    !ruleInst[n, T] in s_seq.

  // Rule definition is valid when it complies with parent
  declRule(s_mod, s_seq, [(_, (_, T_decl)) | _], _, n, T) :- equitype(T_decl, T).
  declRule(s_mod, s_seq, [], [(_, (_, T_decl)) | _], n, T) :- equitype(T_decl, T).

  equitype(RULE(T1, T2), RULE(T3, T4)) :-
    T1 == T3 | error $[Input type [T3] does not match with specified type [T1]],
    T2 == T4 | error $[Output type [T4] does not match with specified type [T2]].

rules

  resolveRule : scope * string -> RULE

  resolveRule(s, n) = R :-
    query rule
        filter P* I* and { n' :- n' == n }
           min $ < P, $ < I, P < I
            in s |-> [(_, (_, R)) | _].
```

```
module mini-str/rules/list-sorts

imports
  abstract-sig/sorts
  signatures/ministr-sig
  mini-str/rules/resolution

signature

  constructors // Create exact types for list build patterns
    EMPTY      : TYPE
    SINGLETON  : SORT -> TYPE
    MULTI      : SORT -> TYPE

rules

  // Rule validating if types match
  // signature: expected type at position * actual type at position
  typeEq: TYPE * TYPE

  // Equal types match
  typeEq(T, T).

  typeEq(EMPTY(), STAR(_)).
  typeEq(EMPTY(), OPT(_)).

  typeEq(SINGLETON(T), OPT(T)).
  typeEq(SINGLETON(T), STAR(T)).
  typeEq(SINGLETON(T), ITER(T)).

  typeEq(MULTI(T), STAR(T)).
  typeEq(MULTI(T), ITER(T)).

rules

  typeEq(OPT(T), STAR(T)).
  typeEq(ITER(T), STAR(T)).

rules

  typeOfList : Pattern * SORT -> TYPE

  typeOfList(List([]), _) = EMPTY().
  typeOfList(List([_]), T) = SINGLETON(T).
  typeOfList(List([_ | _]), T) = MULTI(T).

rules

  typeOfContent: TYPE -> SORT

  typeOfContent(SINGLE(T))    = T.
  typeOfContent(STAR(T))      = T.
  typeOfContent(ITER(T))      = T.
  typeOfContent(OPT(T))       = T.
  typeOfContent(SINGLETON(T)) = T.
  typeOfContent(MULTI(T))     = T.
  typeOfContent(EMPTY())      = _.
```

```
module mini-str/rules/variables

imports
  abstract-sig/types
  signatures/ministr-sig

signature

  relations
    var : string -> TYPE

rules

  declareVar : scope * string * TYPE

  declareVar(s, n, T) :-
    !var[n, T] in s,
    query var
      filter P* and { n' :- n' == n }
      in s |-> [_]
    | error $[Variable [n] declared multiple times].

rules

  resolveVar : scope * string -> TYPE

  resolveVar(s, n) = T :-
    query var
      filter P* and { n' :- n' == n }
      in s |-> [(_, (_, T)) | _]
    | error $[Variabe [n] not declared] @n.
```

```
module mini-str/rules/rules

imports
  abstract-sig/types
  abstract-sig/sorts

  signatures/ministr-sig
  mini-str/rules/variables
  mini-str/rules/resolution
  mini-str/rules/patterns

rules

  rulesOk: scope * scope * scope * list(RuleDef)
  rulesOk(_, s_out, s_out, []).
  rulesOk(s_root, s_seq, s_out, [h | t]) :- {s}
    new s, s -S-> s_seq,
    ruleOk(s_root, s_seq, h),
    rulesOk(s_root, s, s_out, t).

  ruleOk: scope * scope * RuleDef
  ruleOk(s_root, s_seq, RewriteRuleDef(n, m, b, w)) :-
    {s_match s_build s_with T1 T2}
      new s_match s_build s_with,
      s_match -P-> s_root,
      s_build -P-> s_with,
      declareRule(s_root, s_seq, n, T1, T2),
      typeOfBuild(s_build, b) == T2,
      withOk(s_match, s_with, w),
      typeOfMatch(s_match, m) == T1.

rules

  withOk: scope * scope * list(With)

  withOk(s_match, s_build, []) :-
    s_build -P-> s_match.

  withOk(s_match, s_build, [With(str)]) :-
    strategyOk(s_match, s_build, str).

rules

  strategyOk : scope * scope * Strategy

  strategyOk (s_match, s_decl, Assign(n, p)) :- {T}
    s_decl -P-> s_match,
    typeOfBuild(s_match, p) == T,
    declareVar(s_decl, n, T).

  strategyOk(s_match, s_build, Seq(str1, str2)) :- {s_int}
    new s_int,
    strategyOk(s_match, s_int, str1),
    strategyOk(s_int, s_build, str2).
```

```
module mini-str/rules/patterns

imports abstract-sig/types
imports abstract-sig/sorts
imports abstract-sig/constructors

imports signatures/ministr-sig
imports mini-str/rules/resolution
imports mini-str/rules/list-sorts
imports mini-str/rules/variables

signature
  sorts PATTERN constructors
    MATCH       : PATTERN
    BUILD       : PATTERN

rules // Generic pattern rules
  ptrnsOk maps ptrnOk(*, *, list(*), list(*))
  ptrnOk : scope * PATTERN * Pattern * TYPE
  lptrnsOk maps lptrnOk(*, *, list(*), *)
  lptrnOk : scope * PATTERN * Pattern * SORT
  typeOfRules : scope * list(string) -> RULE
  typeOfBuild : scope * Pattern -> TYPE
  typeOfMatch : scope * Pattern -> TYPE
  arityOfCons : list(Pattern) -> int

  ptrnOk(s, pt, Constr(n, p), SINGLE(T)) :- {T'}
    resolveCons(s, arityOfCons(p), n) == CONS(T, T', _)
    | error $[Expected constructor of sort [T]] @n,
    ptrnsOk(s, pt, p, T').

  ptrnOk(s, pt, l@List(i), T) :- {Tc}
    typeOfContent(T) == Tc,
    typeEq(typeOfList(l, Tc), T) | error $[Expected list of type [T]],
    lptrnsOk(s, pt, i, Tc).

  ptrnOk(s, pt, c@RuleCall(n, op), T) :- {T1 T2}
    typeOfRules(s, n) == RULE(T1, T2),
    ptrnOk(s, pt, op, T1),
    typeEq(T, T2).

  ptrnOk(s, MATCH(), Var(n), T) :- declareVar(s, n, T).

  ptrnOk(s, BUILD(), Var(n), T) :-
    typeEq(resolveVar(s, n), T) | error $[Expected variable of sort [T]] @n.

  lptrnOk(s, P, p, S) :- ptrnOk(s, P, p, SINGLE(S)).

  typeOfRules(s, [n]) = resolveRule(s, n).
  typeOfRules(s, [n | t]) = RULE(Tin, Tout) :- {T}
    resolveRule(s, n) == RULE(Tin, T),
    typeOfRules(s, t) == RULE(T, Tout).

  typeOfBuild(s, P) = T :- ptrnOk(s, BUILD(), P, T).
  typeOfMatch(s, P) = T :- ptrnOk(s, MATCH(), P, T).

  arityOfCons([]) = 0.
  arityOfCons([_ | t]) = res :-
    res #= arityOfCons(t) + 1.
```

```
module mini-str/imports

imports
  modules/modules
  signatures/ministr-sig

  abstract-sig/types
  abstract-sig/conflicts/sorts
  abstract-sig/conflicts/constructors

  mini-str/rules/resolution

rules // Import section

  importsOk maps importOk(*, list(*))
  importOk : scope * MOD

  importOk(s, n) :- {s_mod}
    import(s, n, _) == s_mod,
    rulesImpOk(n, s, s_mod).

rules // Check overlapping imports of rules

  rulesImpOk: string * scope * scope

  rulesImpOk(n, s_imp, s_mod) :- {rds}
    query rule
       filter P* I*
          min $ < P, $ < I, P < I
           in s_mod |-> rds,
    ruleMergesOk(n, s_imp, namesOfRules(rds)).

rules

  ruleMergesOk maps ruleMergeOk(*, *, list(*))
  ruleMergeOk: string * scope * string
  ruleTypesOk : string * string * (list(TYPE) * list(TYPE))

  ruleMergeOk(m, s, rn) :- {rls T1 T2}
    query rule
       filter P* I*
          min $ < P, $ < I, P < I
           in s |-> rls,
    ruleTypesOk(m, rn, typesOfRules(rls)).

  ruleTypesOk(_, _, ([_], [_])).
  ruleTypesOk(m, r, (_, _)) :- false
  | error $[Importing conflicting definitions for rule [r]]@m.

rules

  namesOfRules maps nameOfRule(list(*)) = list(*)
  typesOfRules maps typeOfRule(list(*)) = (list(*), list(*))

  nameOfRule : (path * (string * RULE)) -> string
  nameOfRule((_, (n, _))) = n.

  typeOfRule: (path * (string * RULE)) -> (TYPE * TYPE)
  typeOfRule((_, (_, RULE(T1, T2)))) = (T1, T2).
```

```
module mini-str

imports
  signatures/ministr-sig

  modules/modules
  abstract-sig/types

  mini-str/rules/rules
  mini-str/rules/resolution
  mini-str/rules/list-sorts
  mini-str/signatures/signatures
  mini-str/imports

rules

  projectOk: scope
  projectOk(_).

rules

  fileOk : scope * Start
  fileOk(s, Module(n, sec)) :- {s_mod s_seq}
    new s_mod s_seq,
    s_mod -P-> s,
    declareMod(s, n, s_mod, CONSUME()),
    sectionsOk(s_mod, s_seq, _, sec).

rules

  sectionsOk: scope * scope * scope * list(Section)
  sectionOk: scope * scope * scope * Section

  sectionsOk(_, s_out, s_out, []).

  sectionsOk(s_root, s_seq, s_out, [h | t]) :- {s_int1 s_int2}
    sectionOk(s_root, s_seq, s_int1, h),
    new s_int2, s_int2 -S-> s_int1,
    sectionsOk(s_root, s_int2, s_out, t).

  sectionOk(s, s_in, s_out, Rules(rls)) :-
    rulesOk(s, s_in, s_out, rls).

  sectionOk(s, s_seq, s_seq, Signatures(sigs)) :-
    signaturesOk(s, sigs).

  sectionOk(s, s_seq, s_seq, Imports(i)) :-
    importsOk(s, i).
```