Bootstrapping the Statix Meta-Language

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Boris Janssen
born in Tilburg, the Netherlands

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

Bootstrapping the Statix Meta-Language

Author: Boris Janssen
Student id: 4583264
Email: b.f.janssen@student.tudelft.nl

Abstract

The Statix meta-language has been developed in order to simplify the definition of static semantics in programming languages. A high-level static semantics definition of a language in Statix can be used to generate a type-checker, hence abstracting over the shared implementation details. Statix should be able to express the static semantics of itself as well. The process of defining a language using itself is called bootstrapping. In this thesis we discuss the bootstrapping of the Statix meta-language within the Spoofax language workbench. The bootstrapping of a type-system domain specific language hasn't previously been discussed in existing work. It acts as an interesting case study on the use of Statix to define semantics for a constraint-based declarative language as well as the use of Statix throughout a full compiler pipeline. In addition bootstrapping grants Statix a more expressive type system, which enables the future development of the language.

Throughout this thesis we discuss the components of the Statix compiler and explain how these changed as part of the bootstrapping process. The correctness is validated by comparing compiled specifications of the bootstrapped compiler and the existing reference solution for alpha equivalence. We also provide a brief indication of the performance of the bootstrapped Statix compiler.

The bootstrapped compiler is believed to be correct, but does currently not perform to the desired standard. The compiler is successful on smaller language projects, but has a massive growth in compile time when it is used on larger, more complex language projects. This means future work is needed in order for it to be incorporated into the language workbench.

Thesis Committee:

Chair:	Dr. M.T.J. Spaan, Faculty EEMCS, TU Delft
University Supervisor:	Dr. B.P. Ahrens, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
Committee Member:	A.S. Zwaan, Faculty EEMCS, TU Delft

Thesis Advisor: Prof. dr. E. Visser, Faculty EEMCS, TU Delft

Preface

In this preface I would like to take the opportunity to thank those that have helped me finish this thesis and as a result my masters degree. Firstly I would like to thank the late Eelco Visser for helping me get started on this thesis project. His courses throughout my bachelor and masters raised my interest for the field of programming languages and I really appreciated it when he took the time to help me find the right thesis topic after I initially struggled with it. I hope that my thesis can contribute to furthering the work he started. Secondly I would like to thank Aron Zwaan for his guidance throughout the project. I really appreciate his willingness to always answer my questions and all the useful constructive feedback that he has given me. Thirdly I would like to thank Benedikt Ahrens and Matthijs Spaan for their supervision after the untimely passing of Eelco Visser. I truly value the effort they made to help me finish this thesis, even though the topic is not necessarily part of their area of expertise. Finally I want to thank my parents for supporting me throughout my studies and for allowing me to pursue my interests.

Boris Janssen
Delft, the Netherlands
April 12, 2023

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Thesis Outline . . . 2

2 The Statix Meta-language 3
- 2.1 Statix as part of the Spoofax Language Workbench . . . 3
- 2.2 Introduction to Statix . . . 4
- 2.3 Why bootstrap Statix? . . . 8

3 The Statix Compiler 9
- 3.1 Compilation pipeline . . . 9
- 3.2 Static Analysis . . . 10
- 3.3 Normalization . . . 11

4 Bootstrapped Implementation 13
- 4.1 Typing & Name Binding of Statix . . . 13
- 4.2 Duplicate Pattern Checking . . . 17
- 4.3 Scope Graph Extension Analysis . . . 19
- 4.4 Normalization . . . 19

5 Evaluation 21
- 5.1 Correctness . . . 21
- 5.2 Performance . . . 25
- 5.3 Bootstrapping process . . . 28

6 Related work 29
- 6.1 The Statix Meta-language . . . 29
- 6.2 Bootstrapping Meta-languages . . . 30

7 Conclusion 31
- 7.1 Future work . . . 31

Bibliography **33**

List of Figures

2.1 Basic pipeline in Spoofax . . . 3
2.2 Statix signature for plus expressions . . . 4
2.3 Statix specification for plus expressions . . . 4
2.4 Scope graph elements . . . 5
2.5 Statix specification for lambda calculus with plus expressions . . . 6
2.6 Scope graph of $\lambda x.(\lambda y.(x_1 + y_1)x_2)$ . . . 7
2.7 List concatenation of integers and strings . . . 8
2.8 List concatenation using parametric polymorphism . . . 8

3.1 The Statix compiler pipeline . . . 9
3.2 Rule patterns in Statix . . . 10
3.3 Valid and invalid scope extensions . . . 11
3.4 The normalization of a mapping rule . . . 12

4.1 Term Types of Statix . . . 13
4.2 The structure of a scope graph of a Statix specification . . . 14
4.3 Predicate Types of Statix . . . 15
4.4 Label Types of Statix . . . 16
4.5 Statix code relevant to type checking a ListTail term . . . 16
4.6 Fragment of the rules used for comparing rule patterns in Statix . . . 18
4.7 Using the **ref** AST property . . . 19
4.8 Using the **type** AST property . . . 20

5.2 Unit test testing the Statix type system . . . 22
5.3 Scope graph examples of import structures . . . 22
5.4 The equivalence check progress represented by a diagram . . . 23
5.8 Compiler versions compared using Tiger language project . . . 27
5.9 Compiler versions compared using Chicago language project . . . 27

List of Tables

5.1 An overview of the unit tests used to validate type checking 22
5.5 Numbers that reflect the size of several Statix specifications 24
5.6 Results of benchmarking the Statix compiler on the **Tiger** language project 26
5.7 Results of benchmarking the Statix compiler on the **Chicago** language project . . 26

Chapter 1

Introduction

The Statix meta-language (Antwerpen, Poulsen, et al. 2018) is a type-system domain specific language, which has been developed in order to simplify the definition of static semantics in programming languages. Static semantics are the possibly context-sensitive well-formedness conditions of a language that can be checked at compile time. Statix is used to give a high level description of static semantics using a declarative constraint-based approach. It uses scope graphs (Néron et al. 2015) to define name binding patterns. This method is believed to have the potential for standardizing the treatment of name binding in programming languages and their tools (Antwerpen, Poulsen, et al. 2018).

The process of bootstrapping a compiler of a programming language has become a common practice (Konat, Erdweg, and Visser 2016). This process consists of expressing the language that you intend to compile using said language, resulting in a self-compiling compiler. It allows for compiler development to be performed using a higher level language and can additionally be seen as a test of the language being compiled.

Statix is a meta-language which is part of the Spoofax language workbench (Kats and Visser 2010). Language workbenches are compiler compilers that provide a set of high-level meta-languages for defining programming languages and their compilers. The meta-languages each focus on a specific aspect of the language or compiler, such as the syntax and static semantics. This means that a compiler that is expressed using a language workbench consists of definitions in multiple meta-languages.

Since meta-languages are used to only express a specific part of a language compiler, they can't be used to fully bootstrap themselves. Instead, bootstrapping a meta-language is done using the full language workbench, meaning that only the aspect which the meta-language focuses on is expressed using itself. For example a meta-language that is used to describe grammars of programming languages can only be used to describe its own grammar during bootstrapping, while its semantics have to be formulated using other meta-languages. In this thesis we describe how we bootstrapped Statix within the Spoofax language workbench. This consists of expressing the static semantics of Statix using Statix, as well as altering a set of transformations that compile Statix specifications, which are dependent on the semantic analysis result.

There are multiple reasons for bootstrapping Statix. Firstly, bootstrapping Statix allows for further innovation to the language, since its static semantics are defined using a more expressive language. The Statix meta-language can be used to express more complex type systems, such as those that contain parametric polymorphism. This is a feature that Statix itself currently does not support, but it could reduce the size of Statix definitions and makes them easier to express. Secondly, the most recent version of Spoofax, Spoofax 3, does no longer support the meta-language in which the type system of Statix is expressed. This means that in order to bootstrap the entire Spoofax 3 language workbench, Statix type system has to be expressed using Statix. Finally, bootstrapping provides useful insight in the application of

Statix. This is because bootstrapping can be seen as a case study on the use of Statix to define semantics for a constraint-based declarative language as well as the use of Statix throughout a full compiler pipeline.

The main objective of this thesis is bootstrapping the Statix meta-language and validating its correctness. This means that the type checking behavior of Statix after bootstrapping needs to stay consistent and the resulting static semantics definitions should not be altered. Since there are already several language projects that use Statix, this validation process can be done using actual examples.

1.1 Contributions

In this thesis, we claim the following contributions:

- We provide a detailed description of the Statix compiler.
- We discuss our bootstrapped implementation of Statix (Janssen et al. 2023) and explain the motivation behind choices that were made.
- We go through the process of validating the correctness of the bootstrapped implementation.
- We reflect on the feasibility of bootstrapping semantic specification languages.

1.2 Thesis Outline

The remainder of this Thesis is structured as follows. In chapter 2 we give the relevant background on the Statix meta-language. In chapter 3 we give a detailed description of the Statix compiler. The bootstrapped implementation of the Statix compiler is discussed in chapter 4. In chapter 5 the bootstrapped compiler is evaluated. In chapter 6 we look at work related to this thesis. Finally, chapter 7 concludes this thesis.

Chapter 2

The Statix Meta-language

In this chapter we give background information on the Statix meta-language (Antwerpen, Poulsen, et al. 2018). We will explain its role in the Spoofax language workbench. Then we briefly introduce how Statix works, using some examples. Finally we motivate why it is useful to bootstrap Statix.

2.1 Statix as part of the Spoofax Language Workbench

Statix is a meta-language that is part of the Spoofax language designer workbench (Kats and Visser 2010), which is an open-source workbench that language designers can use for designing textual programming languages. Statix can be used to specify the static semantics of a newly designed language. Static semantics are the well-formedness conditions of a language that can be checked at compile time. Two other meta-languages that are part of Spoofax and play an important role in language design are: SDF3 (Souza Amorim and Visser 2020), which is used specify the syntax of a language and Stratego (Visser 2003), which is used to perform transformations on intermediate representations of a language. A basic Spoofax pipeline is shown in Figure 2.1. The following steps are shown:

- In the parse step a grammar specification is used to transform a text file to an abstract syntax tree (AST).
- The static semantics specification is applied to the AST in the analysis step in order to obtain an analysis result.
- The compilation step consists of applying transformation rules to the AST, these transformations can rely on information from the analysis result.

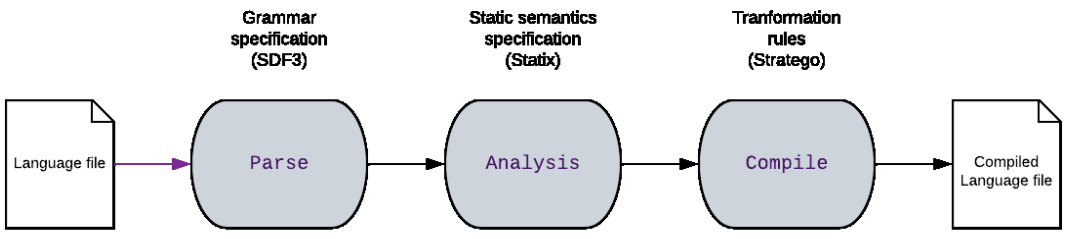

Figure 2.1: Basic pipeline in Spoofax

The role of Statix in Spoofax is to specify a set of constraints on an abstract syntax tree (AST) of a language that represent the static semantics of the designed language. These constraints are context sensitive, unlike parsing constraints. The Statix specification gets normalized (see chapter 3) to an intermediate representation, which can together with the object language AST be passed to the Statix solver to be solved. The Statix solver returns an analysis result that contains type information, such as the types of terms, and possibly error messages when the constraints cannot be fully solved.

2.2 Introduction to Statix

In this section we explain how Statix is used to specify the static semantics of an object language and illustrate this using an example. First we will explain that typing rules in Statix closely resemble formal inference rules. Then we will introduce scope graphs (Néron et al. 2015), which is a concept that allows for the encoding of name binding patterns in a wide range of programming languages.

2.2.1 Typing rules

Typing rules in Statix are expressed using a construction that is similar to formal inference rules. This means that every rule has a conclusion that holds when all the premises that belong to it hold as well. This often means that a Statix rule will have a conclusion that matches a node of an AST and this conclusion holds if the premises that apply to its subtrees also hold. We will illustrate this with the following example of a Statix definition for plus expressions, for which the signature is given in Figure 2.2.

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

  sorts Type constructors
    INT     : Type
```

Figure 2.2: Statix signature for plus expressions

```
rules
  typeOfExp : Exp -> Type

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

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

Figure 2.3: Statix specification for plus expressions

In Figure 2.3 the Statix rules that apply to plus expressions are shown together with its signature. The rule for integers doesn't have any premises, while the rule for a plus expression is a conclusion with two premises after the ":-" which are linked by the conjunction

operator ",". Using this example we can see that if we have to substitute the plus expression 2 + 3, which has the textual AST: `Plus(Int(2), Int(3))`, in the bottom rule that the two premises match the top rule, meaning the premises and consequently the conclusion on the plus expression holds.

2.2.2 Scope Graphs

The typing rules in the simple example above don't need any context with regards to the AST node for them to hold, but this isn't true for all AST nodes in less basic expressions. For example a variable reference node without context, won't allow for a conclusion on the type of the variable. This is why traditionally an environment is passed down to every premise, such that a rule about a reference can have access to its declaration. However Statix doesn't use an environment for storing its declarations but it uses a scope graph (Antwerpen, Poulsen, et al. 2018; Antwerpen, Néron, et al. 2016; Néron et al. 2015) instead. We will now explain the elements of a scope graph, which are shown visually in Figure 2.4:

- Scopes represent a region of a program where name resolution behavior is uniform. Scopes in a scope graph are represented by nodes.
- Labeled, directed edges between scope nodes model the visibility between scope nodes. This allows for the ability to for example differentiate between the visibility within a programming module and its imported modules.
- Declarations in scope graphs are represented by a node containing the information of the declaration, such as variable name and type, and a edge connecting to a scope node labeled by a relation name. The relation represents what subset of declarations (namespace) the declaration belongs to. Examples of frequently used relations in scope graphs are variables and functions.
- References in scope graphs are known as queries. The query specifies which relation the reference belongs to and what the rules are regarding its visibility. The Statix solver will be able to infer which declaration(s) belong to a query.

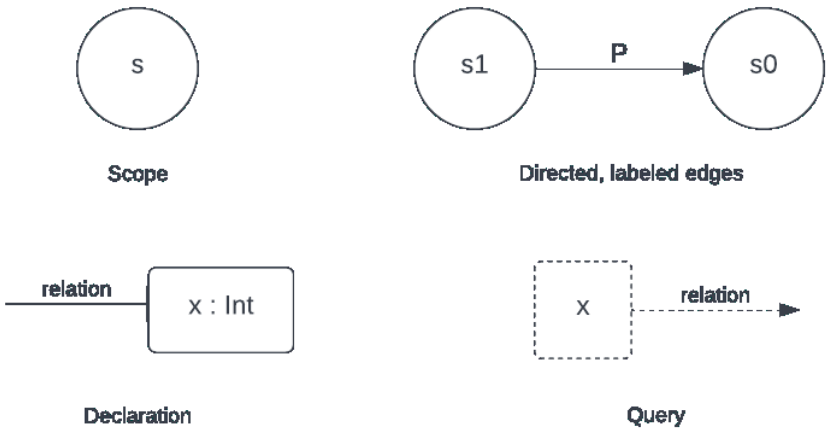

Figure 2.4: Scope graph elements

Rather than passing down an environment to premises, in Statix a scope will be passed down. This scope can then be used to add new declarations to it, extend the scope with another scope using a labeled edge or use it as the entry point for query to resolve a reference.

We will now demonstrate how scope graphs in Statix are used using an example of a Statix specification for a basic lambda calculus with plus expressions which is given in Figure 2.5. We will go through this specification using the scope graph that successfully gets created from the following lambda calculus example (including reference indexes): $\lambda x.(\lambda y.(x_1 + y_1)x_2)$, which has textual AST: `Fun("x", App(Fun("y", Plus(Var("x"), Var("y"))) Var("x")))`. The scope graph is shown in Figure 2.6.

```
signature
  sorts Exp constructors
    Int     : int -> Exp
    Plus    : Exp * Exp -> Exp
    Fun     : string * Exp -> Exp
    Var     : string -> Exp
    App     : Exp * Exp -> Exp

  sorts Type constructors
    INT   : Type
    FUN   : Type * Type -> Type

  relations
    var : string -> Type

  name-resolution
    labels P

rules
  typeOfExp : scope * Exp -> Type

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

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

  typeOfExp(s, Fun(x, e)) = FUN(S, T) :- {s_fun}
    new s_fun,
    s_fun -P-> s,
    !var[x, S] in s_fun,
    typeOfExp(s_fun, e) == T.

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

  typeOfExp(s, App(e1, e2)) = T :- {S}
    typeOfExp(s, e1) == FUN(S, T),
    typeOfExp(s, e2) == S.
```

Figure 2.5: Statix specification for lambda calculus with plus expressions

The first thing we consider is that part of the signature are the declarations of the relation and edge label, which will be used in scope graph creation. The rule signature of `typeOfExp` is different than the one in Figure 2.3, because now a scope is added such that this can be passed down to sub-expressions, which can be seen in the plus expression rule (Line 24-26).

Scope creation in this example happens in the rule for functions (Line 28-32). In the first two premises of this rule a new scope is introduced and then the current scope is extended using a `P` labeled edge, the result of this can be seen twice in Figure 2.6. The following premise contains the declaration of the variable introduced by the lambda in the newly created scope using the appropriate relation. Finally the type of sub-expression is checked using the new scope.

In the rule for variable references (Line 34-38) scope querying occurs. A scope graph query consists of multiple parts, that each have an effect of what the result of the query will be. Firstly it is defined under which relation the query is performed. Secondly a regular expression describes which edges can be traversed, in our example in Figure 2.5 this can be any number of `P` edges. After this a constraint can be defined on the resulting declarations that filters them, in our example the declaration string has to be equal to the string of the variable reference. Then an order can be defined on which paths in a scope graph are preferred, the query in Figure 2.5 prefers declarations in the current scope (`$`) over those reached via one or more `P` edges. Finally the scope in which the query starts is given and the term that will match with the result of the query. The result term is a list where every element is a tuple of a path term, representing the path taken and a tuple term containing the retrieved declaration.

In the scope graph in Figure 2.6 it can be seen that both variables are declared with the integer type, but if you look at the `Fun` rule in Figure 2.5 you can see that the declaration occurs with a variable `S` instead of the `Int()` type constructor. The declaration in the scope graph will be of integer type however, since Statix uses type inference and the variable `S` will be solved to an integer type, because the variable references are part of plus expressions, which require its sub-expression to be of integer type.

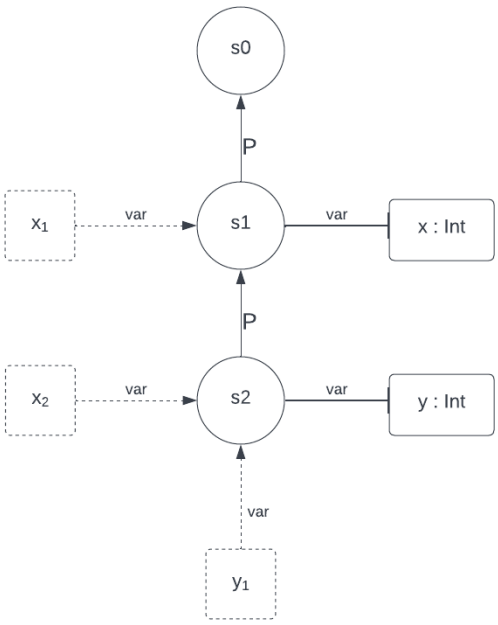

Figure 2.6: Scope graph of $\lambda x.(\lambda y.(x_1 + y_1)x_2)$

2.3 Why bootstrap Statix?

In this section we will explain the motivation behind bootstrapping Statix. Firstly the type system of Statix is currently expressed using NaBL2 (Antwerpen, Néron, et al. 2016), which is meta-language used to express static semantics that is more restrictive than Statix. This restrictiveness means that certain improvements that could be made to Statix are currently not possible, due to the limitations of the type system in NaBL2. An example of such an improvement is the addition of parametric polymorphism in constraint type signatures. Currently Statix requires every user defined constraint to have an explicit type signature, which means that constraints that operate on lists require the exact type signature of the elements of the list. This results in the user having to define a different constraint for every type of list it wants to use the constraint on. In Figure 2.7 you can see that currently you would require two different constraints for concatenating lists of different types even though the rule structure is equal, so ideally you would use parametric polymorphism as shown in 2.8 such that you would need only one constraint definition.

```
rules
  concatIntList: list(int) * list(int) -> list(int)

  concatIntList([], ys) = ys.
  concatIntList([x|xs], ys) = [x|concatIntList(xs, ys)].

  concatStringList: list(string) * list(string) -> list(string)

  concatStringList([], ys) = ys.
  concatStringList([x|xs], ys) = [x|concatStringList(xs, ys)].
```

Figure 2.7: List concatenation of integers and strings

```
rules
  concatList: A => list(A) * list(A) -> list(A)

  concatList([], ys) = ys.
  concatList([x|xs], ys) = [x|concatList(xs, ys)].
```

Figure 2.8: List concatenation using parametric polymorphism

The second reason for bootstrapping Statix is that in the newest version of the Spoofax language workbench, Spoofax 3, the NaBL2 language is no longer supported, which means that in order for the language workbench to be bootstrapped the static semantics definition of Statix can no longer be in NaBL2, but has to use Statix instead.

Finally bootstrapping Statix can give use useful insight on the application of Statix. For example it allows for further studies on the performance of the Statix solver, since currently existing Statix projects now use the Statix solver when they get built instead of the NaBL2 solver. This means these projects can now be used to benchmark the Statix solver, while in the past you would need to use a project written in the language for which the Statix project has defined static semantics.

Chapter 3

The Statix Compiler

In this chapter we discuss the Statix compiler. We mention the different steps of the compilation pipeline and we go into depth about the analysis and normalization steps.

3.1 Compilation pipeline

In this section we will describe the steps which transform a textual Statix specification file to a file which is compatible with the Statix solver. The Statix pipeline can be seen in Figure 3.1 along with the pipeline of an object language that the Statix specification file belongs to, such that it is clear how the two are related. In the bootstrapping scenario the object language is Statix as well.

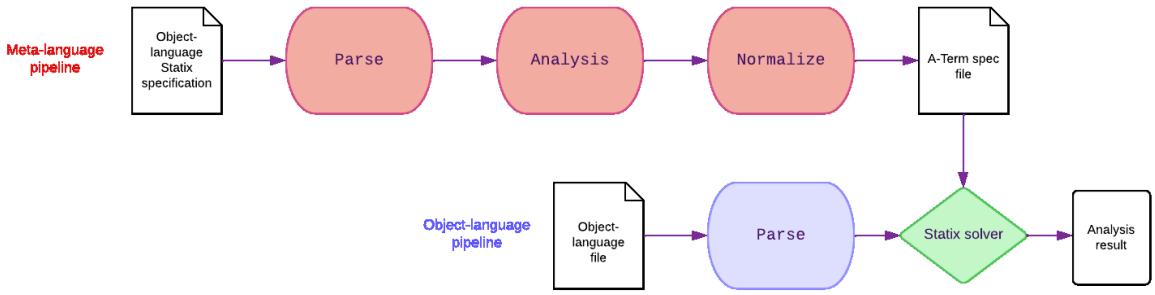

Figure 3.1: The Statix compiler pipeline

These are the steps taken in the Statix compiler pipeline:

- The first step is parsing the text file to an abstract syntax tree (AST) according to its grammar specification in SDF3 (`parse : textfile → AST`).
- After this step static analysis is performed on the AST by combining it with the static semantics specification and solving the constraints. The results of the solver are stored in an analysis object (`analysis : AST → analysis result`).
- The type information stored in the analysis object can then be used to transform the AST to the spec A-Term file supported by the Statix solver in the normalization step (`normalize : AST, analysis result → spec`).

3.2 Static Analysis

The static analysis mostly gets done by solving the constraints defined in the specification file. In the current compiler this file is written in the deprecated NaBL2 meta-language (Antwerpen, Néron, et al. 2016), while in the bootstrapped version of the compiler the constraints will be specified in Statix itself. A large part of the constraint specification concerns the typing of Statix and how names in Statix are resolved. An in depth explanation of how this has been implemented using Statix is given in section 4.1.

3.2.1 Custom Analysis

The majority of the static semantics of Statix can be expressed using the aforementioned meta-languages, but there are some constraints that the meta-languages cannot support. An example is checking whether the name of a Statix module corresponds to its file-path, since both NaBL2 and Statix don't have access to this information. To make sure that these kind of static semantics can also be checked, the runtime of the meta-languages allows the designers to write a custom analysis section in the Stratego meta-language, which is a less restricted language used for the manipulation of ASTs. Checking for duplicate rule patterns and correct scope graph extension behavior are two more complex examples of this custom analysis in the Statix compiler. We will explain these analysis examples in the following subsections.

3.2.2 Duplicate rule patterns

As part of the solving process the Statix solver needs to perform pattern matching on the rules that are part of a specification. To make sure that the solver will always match on the same rule when presented with the same argument terms, Statix has a rule specificity order, where the most specific rule will be selected during pattern matching. To make sure that it will never be unclear which pattern is the most specific, this is statically checked.

```
rules
  rule: int * int

  rule(8, 8).
  rule(x, x).
  rule(x, y).
  rule(a, b) :- true.
  rule(_, _) :- false.
```

Figure 3.2: Rule patterns in Statix

In Figure 3.2 a set of rule patterns is given for a rule that takes two integers. The top three rules are decreasingly specific and are allowed to exist together. The bottom three rules all have the same specificity, namely they accept any pattern of two integers, so they shouldn't be allowed to exist together.

In the Statix compiler where NaBL2 is used to express the static semantics, the check for invalid overlapping rule patterns is part of the custom analysis, but in the bootstrapped Statix compiler this analysis is expressed in Statix and is described in section 4.2.

3.2.3 Scope graph extensions

To ensure that the Statix solver can correctly solve all its given constraints using a constructed scope graph, the scope graph construction has to adhere to some rules. In Figure 3.3 some

examples of valid and invalid scope graph extensions are given.

```
1  rules
2    extend: scope
3
4    extend(s1) :- {s2} new s2, s2 -P-> s1.            //valid
5    extend(s1) :- {s2} s2 -P-> s1.                    //invalid
6
7    rule: string
8
9    rule("correct") :- {s} new s, extend(s).          //valid
10   rule("incorrect") :- {s} extend(s).               //invalid
11
12   extend(s) :- !var["x"] in s.                      //valid
13   extend(s) :- query var                            //invalid
14                   filter P* and {x :- !var[x] in s}
15                   in s |-> _.
```

Figure 3.3: Valid and invalid scope extensions

The first two rules in lines 4 and 5 show that in order for a scope to be extended it has to be owned first, for a more detailed explanation on what this entails see (Antwerpen, Poulsen, et al. 2018). A newly introduced variable is not an owned scope until it is referenced in a `new` constraint. A scope passed down to a rule head is assumed to be owned, but if one of the rules belonging to a certain constraint extends the scope that is passed down, all calls to that constraint in other rules will require the passed down scope to be owned, which you can see in lines 9 and 10.

The rules in lines 12-15 of Figure 3.3 show in which context a scope can and cannot be extended. In rule 4 it is attempted to extend the scope graph within the context of a scope graph query, if this was valid it would mean that within the query the scope would have an extra declaration, but outside of it that would not be the case and this would violate the fact that a scope graph is consistent and the outcome of its queries are always the same regardless of where the query is placed within the specification.

The analysis that checks whether a Statix specification adheres to rules set on scope graph extensions is implemented in Stratego using the analysis result of the static analysis in both versions of the compiler. A more in depth explanation of this analysis and how it need to be adjusted during bootstrapping is given in section 4.3.

3.3 Normalization

After the analysis step has completed the resulting type information can be used to transform the abstract syntax tree to a file format that is supported by the Statix solver. The first part of this transformation step consists of converting the syntactic sugar used in Statix to the core constructs that are supported by the Statix solver.

An example of such a normalization is given in Figure 3.4. In this example you can see how the `maps` construct gets normalized to a signature and two separate rules. The `maps` construct is used as a shorthand for a constraint that applies another constraint to every element of a list or lists. In the example in Figure 3.4 the `checkExpressions` constraint will go through a list of `Exp` using the `checkExpression` constraint. The `Exp` parameter of the `checkExpression` constraint is lifted to a list parameter(`list(*)`), while the scope parameter is not (`*`), this can also be seen in the normalized signature. This transformation is dependent on the analysis

result, because it needs to infer what the type of the * is in the mapping rule. The two rules that are constructed as part of the normalization are a base case rule for an empty list and a recursive rule, in which the mapped constraint gets applied to the head element of a list and a recursive step is applied to the tail of the list.

```
rules
    checkStatements: scope * Exp

    checkStatements maps checkStatement(*, list(*))

    =======>

    checkStatements: scope * list(Exp)

    checkStatements(_, [ ]) :-
      true.

    checkStatements(x_1, [x_2|xs_2]) :-
      checkStatement(x_1, x_2),
      checkStatements(x_1, xs_2).
```

Figure 3.4: The normalization of a mapping rule

Once all the sugared constructs have been transformed an A-Term is generated. An A-Term is tuple that gets stored to a file and contains all the information of the Statix specification that the Statix solver needs to solve the constraints of a specification applied to an AST of an object language. The A-Term tuple is made up of the following five lists:

- A list of imported modules
- A list of edge labels introduced in the module
- A list of relations introduced in the module
- A list of all rules specified in the module. These rules consist of a rule pattern and their premises, the rule signatures are not needed.
- A list of scope graph extensions that occur in the rules used by the module. An extension is made up of a rule name, a number indicating which parameter of that rule is a scope that get extended and the name of the label with which the scope get extended, either a relation or an edge label.

The first four lists can easily be extracted from the normalized AST, the list of scope graph extensions is a result of the custom analysis mentioned in section 3.2.3.

Chapter 4

Bootstrapped Implementation

In this chapter we discuss the important elements of the bootstrapped implementation (Janssen et al. 2023) as well as the relevant design choices that were made. This chapter could help someone better understand the code that accompanies this thesis.

4.1 Typing & Name Binding of Statix

In this section we will describe the type system of Statix (Antwerpen, Poulsen, et al. 2018) and how this type system is used with regard to name binding and name resolution in Statix. We will cover the names that can be defined in Statix and explain the motivation behind their name binding policy in the bootstrapped version of Statix. We we will then show how types in Statix are used to type check Statix using an example.

4.1.1 Term Types

Constraints in Statix are largely made up of terms. These can be simple terms like e.g. integers and strings or they can be composite terms like e.g. lists and tuples. In order to be able to define typing rules for constraints the term types seen in Figure 4.1 are defined. A majority of these types do not require further explanation since they state what they represent. The `Sort` type contains the string that refers to the sort that the type is associated with. The List type contains the type of all elements of the list, since all elements of a list in Statix should have the same type. Finally the Tuple type consists of a list of types that the tuple is composed of.

```
sorts
  TType

constructors
  INT : TType
  STRING : TType
  PATH : TType
  LABEL : TType
  AST_ID : TType
  SCOPE : TType
  LIST : TType -> TType
  TUPLE : list(TType) -> TType
  SORT : string -> TType
```

Figure 4.1: Term Types of Statix

4.1.2 Modules

The modules of Statix are defined in the scope-graph using a relation that pairs the module identifier with a scope (`module : string * scope`). The scope is the module scope in which all constraints, sorts, constructors, relations and labels are declared. This scope can be retrieved from the scope graph when a module is referenced in an import statement and then can be used to add an import edge to the scope graph. The relation is not a functional relation from identifier to scope, because the relation is queried using a scope to find the corresponding identifier as well, such that it can be used for error messages or qualifying the name of a predicate or label reference. In the next section we will briefly describe the structure of a scope graph created from a Statix specification.

4.1.3 Scope graph structure

There is one global scope, which contains the declarations of all Statix modules, this scope gets extended by all individual modules. The module scopes contain the declarations of user defined constraints, sorts, constructors, relations and scope graph labels. When a module imports another module there is a an import edge in the scope graph between the two module scopes.

Module scopes get extended with any number of rule scopes representing every rule inside of the module. The rule scope contains the declarations of the variables that are part of its rule pattern. A rule scope can get extended one or more times by a scope of an existential constraint or a scope of a higher order constraint belonging to a scope graph query, which will contain the declarations of variables introduced by that constraint.

A schematic overview of a scope graph of a Statix specification is shown in Figure 4.2.

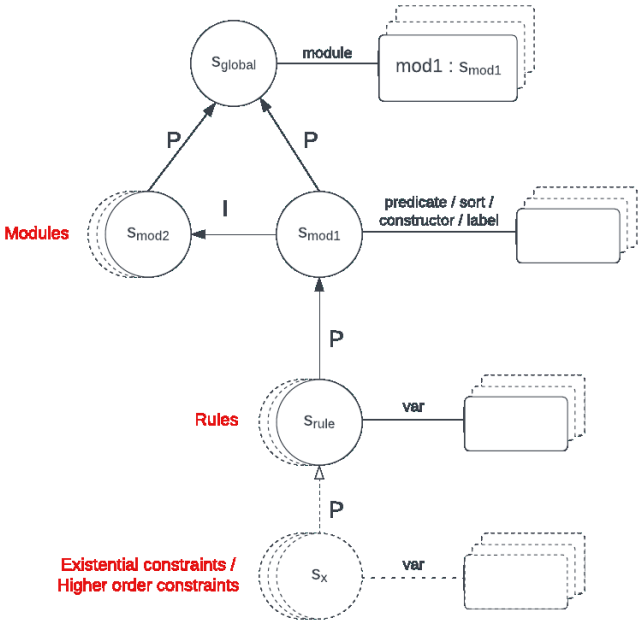

Figure 4.2: The structure of a scope graph of a Statix specification

4.1.4 User Defined Constraints

User Defined Constraints can't be expressed using a term type, but they do have a type signature that consists of term types. This is why a sort for constraint types is defined to represent

```
sorts
  IType

constructors
  PRED : list(TType) -> IType
  FUN : list(TType) * TType -> IType
```

Figure 4.3: Predicate Types of Statix

constraints, which can be seen in Figure 4.3. A basic constraint or predicate type is composed of a list of the term types of its parameters, while a functional constraint type also contains a separate term type to represent the type of the term it returns.

Constraints are declared in the scope graph using a relation that maps their identifier to their constraint type (`predicate` : `string` → `IType`). Constraints in Statix are declared with their type signature, which makes their name binding straightforward. A constraint type is resolved when either a rule for it is defined or it is used as an inline constraint or term.

4.1.5 Variables

Variables that are used in rules are defined by a relation that maps the variable identifier to a term type (`variable` : `string` → `TType`). Variables can be declared either by being part of a rule head or they can be introduced in an existential constraint. Both ways of declaration presented challenges for declaring the variables in a scope-graph in Statix, which we will now explain.

A variable can occur multiple times in a rule pattern, but should be declared only once. This means that when you encounter a variable in a rule pattern as part of an AST while type checking, you can't just declare it immediately since that might lead to duplicate declarations of the same variable. Instead all variables occurrences are collected when type checking a rule pattern, duplicates get filtered out and then the variables that remain get declared.

In an exists constraint new variables get introduced, but these variables have no type associated with them. They can get bound to any type, but the type has to be consistent throughout its use. This means that when the variable has to be declared the type is unknown, but it does have to be declared in the scope-graph somehow, since there should be a way to resolve it when the variable gets referenced. To get around this problem we make a clever usage of the way Statix constraints are solved. We declare variables with a free unification variable instead of an already derived type, but because Statix constraints are solved using type inference, the type will be correctly inferred from its references.

4.1.6 Sorts & Constructors

Sorts can be defined by just their user defined name or they can alias another type. In the scope-graph sorts are declared using a relation that maps their identifier to a term type (`sort` : `string` → `TType`), for a non-aliased sort this term type will be of type sort with the newly declared identifier as its parameter, while for alias declarations the type will be the aliased type.

The relation that defines constructors in the scope-graph maps a combination of the constructor's identifier and an integer to both a list of term types and a singular term type (`constructor` : `string` * `int` → (`list`(`TType`) * `TType`)). The reason that the integer is also needed to define a particular constructor is because it represents the number of arguments a constructor has and it is allowed to use the same identifier for multiple constructors as long as

```
sorts
  LType

constructors
  EDGE        : LType
  RELATION    : IType -> LType
```

Figure 4.4: Label Types of Statix

they have different arities. The right-hand side of the relation represents the types of the arguments of the constructor and the type of the sort the constructor belongs to.

4.1.7 Relations & Scope Graph Edge Labels

Relations and scope-graph edge labels are defined by a single relation. We chose one relation, because relations and scope-graph labels belong to the same name-space and the Statix solver models relations as edge labels (Rouvoet et al. 2020). We could have chosen to use two separate relations, but that would mean that checking whether you are using a duplicate name requires two scope-graph queries instead of one, which is less efficient. To differentiate between relations and scope-graph edge labels we have defined a label type that is shown in Figure 4.4, and the relation maps identifiers to these label types (`label : string` $\rightarrow$ `LType`). A label type can either be of edge type or of relation type, where a relation type contains an instance of the same internal type we use to define constraints. This type can be used, since relations are also made up out of a list of term types and can possibly be functional.

4.1.8 Type Checking Example: ListTail Term

```
signature
  constructors
    ListTail: list(Term) * Term -> Term

  constraints
    termOk: scope * Term -> TType
    listTermOk: scope * list(Term) -> TType

rules
  termOk(s, ListTail(hs, tail)) = LIST(T) :-
    T == listTermOk(s, hs),
    LIST(T) == termOk(s, tail).
```

Figure 4.5: Statix code relevant to type checking a ListTail term

The types and scope-graph relations described in the previous subsections are used to typecheck Statix programs. During this process conditions are checked like the types of a user defined constraints' parameters match the types of the arguments of some reference of that particular constraint. Another example is that the types of a ListTail term (Figure 4.5) are correct, we will use this as an example of what type checking Statix in Statix looks like.

A ListTail term in Statix is like a cons term, but instead of a singular head you can have any arbitrary number of head elements. This means that the ListTail term is made up out of

a list of terms of the same type T and a tail term that needs to be of list type of type T. For example $[1, 2, 3|[4]]$ is a correct ListTail term while $[1, ''2'', 3|[4]]$ and $[1, 2, 3|4]$ are terms that will not type check. The statix code for this type checking behaviour can be seen in Figure 4.5.

4.2 Duplicate Pattern Checking

In this section we explain how we have defined the constraints that check for duplicate rule patterns in Statix using Statix. The necessity of such an analysis is described in section 3.2.2. In short, this analysis is required to ensure that the Statix solver always has a clear answer, when it needs to determine what constraint it should match on.

A rule pattern in Statix is also called the rule head and is made up of a constraint name, which the rule belongs to, and the terms that represent the rule pattern. There should be no duplicate rule patterns within a specification. In order to compare a rule head with other rule heads we have defined the following relation: `rulePattern` : `string` $\rightarrow$ (`list`(`Term`) $*$ `scope`), which allows us to declare rule patterns and query the scope graph for all patterns that a pattern needs to be compared to. The relation maps the name of the corresponding constraint to a tuple of the list of terms representing the pattern and the scope in which the corresponding constraint is declared. The scope is necessary, because in Statix you are allowed to define a constraint that shadows a constraint in an imported module.

4.2.1 Comparing two rule patterns

In order to compare two rule patterns you need to compare all terms from left to right and in the end you should be able conclude whether these two patterns match on a different set of inputs or not. Comparing an integer or string term is straightforward, but comparing variables requires the context in which the variables are located. When you compare the following three patterns: `rule`(`a`, `a`), `rule`(`b`, `b`), `rule`(`c`, `d`), the first two should be concluded to be equal, but the third is different, this is because where variable `a` in the first pattern is used variable `b` is used consistently in the second pattern, but in the third pattern two different variables are used. This is why during a pattern comparison we pass along a list of string tuples that bind two variables, such that further occurrences of these variables can be compared correctly.

A Fragment of the Statix code that implements the pattern comparison is given in Figure 4.6. The `comparePatterns` constraint is given two list of terms, representing the rule patterns and a list of string tuples, representing the previously mentioned mapping between variables. The constraint results in a tuple containing an integer, which encodes the boolean result of the comparison, and the mappings of all variables encountered. Since Statix doesn't have built in booleans and if statements, we use integers and pattern matching to define different branches of a comparison. The `compareTerms` constraint uses pattern matching to compare all possible combinations of Statix terms, Figure 4.6 shows the cases for integer terms and tuple terms as an example. The `comparePatternsHelper` rules show how a result of `compareTerms` is used to either conclude that a pattern is different or the resulting mappings are passed along to the comparison of the remaining terms.

```
signature
  sorts
    Mappings = list((string * string))

  sorts Term constructors
    Int: int -> Term
    Tuple: list(Term) -> Term
rules
  comparePatterns : list(Term) * list(Term) * Mappings -> (int * Mappings)

  comparePatterns([], [], ms) = (0, ms).
  comparePatterns([], _, ms) = (1, ms).
  comparePatterns(_, [], ms) = (1, ms).
  comparePatterns([x|xs], [y|ys], ms) = comparePatternsHelper(compareTerms(x, y, ms), xs, ys).

  comparePatternsHelper: (int * Mappings) * list(Term) * list(Term) -> (int * Mappings)

  comparePatternsHelper((1, ms), _, _) = (1, ms).
  comparePatternsHelper((0, ms), xs, ys) = comparePatterns(xs, ys, ms).

  compareTerms : Term * Term * Mappings -> (int * Mappings)

  compareTerms(Int(x), Int(x), ms) = (0, ms).
  compareTerms(Tuple(xs), Tuple(ys), ms) = comparePatterns(xs, ys, ms).
  ...
  compareTerms(_, _, ms) = (1, ms).
```

Figure 4.6: Fragment of the rules used for comparing rule patterns in Statix

4.2.2 Comparing all rule patterns

We will now list the steps we have used to compare a rule head to all relevant patterns during its type checking:

1. Declare the rule pattern in the scope graph using its constraint name.

2. Resolve all the rule patterns using a scope graph query that selects all patterns declared using the same constraint name from the current module and all imported modules.

3. Filter out the rule patterns of shadowed constraints by looking at whether the scope of the declarations matches the scope of the constraint for which the current pattern is being checked.

4. Compare all the remaining rule patterns to the current pattern and filter out all the patterns that are different.

5. Check whether a single rule pattern remains, if not add the appropriate error message to the rule head of the rule that is being type checked.

4.3 Scope Graph Extension Analysis

In this section we briefly discuss how we have adapted the preexisting implementation for scope graph extension permission analysis to use Statix. The reasons for why this analysis is necessary are given in section 3.2.3 and in Rouvoet et al. (2020). Since we only adapted the previous implementation we will only touch on what we needed to include in the Statix analysis and won't provide many details on how the analysis is performed.

The scope graph extension analysis consists of two main parts, checking whether the scope that is being extended is instantiated and checking whether the extension of a scope is allowed within the context in which it is located. Verifying whether a scope is instantiated is done by solving a set of constraints that get generated by going through the ASTs of Statix rules. These constraints state whether a variable declaration or reference either provides scope extension permission or requires it. They are solved by a custom fixed point solver. In order to solve these constraints it is required that variable references can correctly be resolved to their declarations. To ensure that this is possible we set the **ref** AST property of every variable reference to its declaration when resolving it in Statix, as seen in figure 4.7.

```
typeOfVariable(s, Var(id)) = T :- {id'}
    resolveVariable(s, id) == [(_,(id', T))|_] | error $[Variable [id] not defined],
    @id.ref := id'.
```

Figure 4.7: Using the **ref** AST property

The scope extension analysis that is reliant on context uses barriers to determine when a scope can and cannot be extended. The barriers are used to section of parts of a scope-graph. An example of when such a barrier is introduced is within the higher order constraint of a scope graph query, only scopes within this context are allowed to be extended since the other scopes cross the barrier. To model these barriers in Statix we use two relations `barrier` : `scope` and `varBarrier` : `string` → `scope`. The barrier relation allows for the declaration of scope graph barriers and the varBarrier relation is used to indicate what outer scope can be reached from a variable declaration without crossing a barrier. Comparing the scope that can be reached from the scope of a variable reference to the scope that is obtained by querying varBarrier determines whether that variable reference could be that of a scope that gets extended.

4.4 Normalization

In this section we discuss how bootstrapping affects the normalization step of the Statix compiler. The transformations that occur during the normalization step don't need to be changed themselves, but they are dependent on the analysis result, which in the bootstrapped version is a Statix analysis result, while in the original version it is an NaBL analysis result. This means that the transformations need to be modified to use the Statix analysis result and the Statix analysis result needs to provide the information necessary for the transformations. In the remainder of the section we address how we have adapted specific parts of the normalization step.

4.4.1 Transformation of Sugared Constructs

The transformations of sugared constructs are dependent on the type information of terms and constraints. In these transformations the AST nodes of sugared constructs are replaced with core constructs, for an example see 3.4 in section 3.3. In the original version of the

Statix compiler the transformations are implemented by first annotating the AST with the type information from the NaBL analysis result and then using the annotations to do the normalization. This meant that in the bootstrapped version we can annotate the AST with the type information from the Statix analysis result and as long as the type information is consistent there is no need to further change the transformation steps.

To enable the correct annotation of the AST we set the **type** AST property of all term and constraint AST nodes to its type during the analysis step. In Figure 4.8 you can see how the **type** AST property is used during type checking within the example from section 4.1.8. In the rule head the AST node is ascribed to variable a and in the final premise the type is bound to the node. The AST node can then successfully be annotated in the normalization step by retrieving the type using the Statix API in Stratego.

```
termOk(s, a@ListTail(hs, tail)) = LIST(T) :-
  T == listTermOk(s, hs),
  LIST(T) == termOk(s, tail),
  @a.type := LIST(T).
```

Figure 4.8: Using the **type** AST property

4.4.2 Specialization of Scope Graph Queries

In Zwaan (2022) a specialization of scope graph queries is added to the normalization step of the Statix compiler, in order to speed up the solving process of a Statix definition. In order to transform scope graph queries to their specialized versions, some type information from the analysis result is needed. One of these requirements is access to a list of all available edge labels to the query. We have implemented this by querying for all edge labels during the type checking of a scope graph query and adding the result to the AST node of the query using a labels AST property.

Chapter 5

Evaluation

In this chapter we evaluate the bootstrapped implementation of Statix. First we describe the process of validating its correctness. Then we have a look at its performance and finally we discuss the main takeaways from the bootstrapping process. The code used for the evaluation accompanies the bootstrapped implementation (Janssen et al. 2023).

5.1 Correctness

In this section we discuss how we have validated whether the bootstrapped Statix compiler produces the same result as the original compiler. The validation process is important, because bootstrapping should not result in different behavior of the compiler. This is because if the compiler would give different results, existing language projects that use Statix might become invalid or produce unwanted changes to the language.

5.1.1 Unit testing

In order to test whether the bootstrapped Statix compiler has interchangeable type checking behavior we use unit tests to verify specific aspects of the type system of Statix. These tests are written using the Spoofax Testing language (SPT) (Kats, Vermaas, and Visser 2011), which is part of Spoofax that allows language developers to express tests and provides a framework to execute them.

Unit tests are important, because they allow the type analysis of the bootstrapped Statix compiler is equivalent to that of the original compiler. Correct Statix definitions should pass the analysis, while definitions that contain type errors should not pass the analysis. Whether a Statix definition contains an error is stored in an analysis result, this result is specific to language in which the static analysis is expressed, in the original compiler NaBL2 and in the bootstrapped compiler Statix. This means we cannot directly compare these analysis results, but we use unit tests to compare them instead. Tests that pass using the original compiler should also pass for the bootstrapped compiler.

The original Statix compiler already had an accompanying test suite in SPT, which forms as a basis for the test suite we use to validate the correctness of the bootstrapped Statix compiler. This test suite consists of both tests that verify the correct type checking of Statix as well as the correct solving of a Statix constraint.

While the original test suite does include tests that check Statix definitions that violate its set typing rules, it is far from complete. This is why we have extended the test suite such that there is a test case for almost every possible typing error that could occur in Statix. An overview of the test suite can be seen in Table 5.1. An example of a unit test is given in Figure 5.2.

	Number of Tests
Terms	26
Sorts & Constructors	8
Constraints	36
Rules	37
Relations & Scope graph labels	11
Queries	33

Table 5.1: An overview of the unit tests used to validate type checking

```
test incorrect strings rule [[
  module test

  rules
    rule: string * string
    rule("hi", 9).
]] analysis fails
```

Figure 5.2: Unit test testing the Statix type system

5.1.2 Correct import behavior

A part of the type analysis of Statix involves adding edges to imported modules to the scope graph, such that imported constraints, sorts and other definitions can be correctly resolved in the analysis of a module. SPT unfortunately does not support tests that contain multiple Statix modules, this is why we tested for correct and incorrect imports manually using the integrated development environment Eclipse. When the analysis result contains an error, this is shown in Eclipse.

We tested whether all constructs that can be imported are correctly resolved within the module that contains the import, as well as whether duplicate imported constructs are not allowed. Statix supports transitive imports for sorts, constructors and constraints, which means we also tested for imports that get resolved through multiple import scope graph edges. Transitive imports allow for the possibility of cyclic and diamond imports, for which scope graph examples are shown in Figure 5.3.

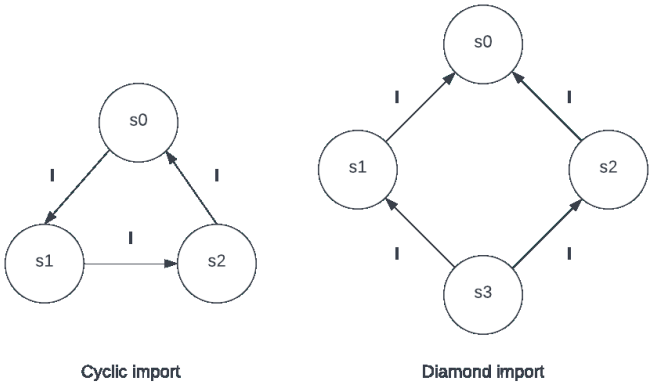

Figure 5.3: Scope graph examples of import structures

5.1.3 Equivalence checking

In addition to checking whether the type analysis is correct using unit tests, we check whether the transformations which use the analysis result stay consistent and the produced A-Term stays equivalent. This means that the normalized Statix definition used by the Statix solver in order to perform analysis does not change no matter what compiler version is used. Using another meta-language for the analysis shouldn't change the result of the transformations that rely on the analysis.

The contents of an A-Term are described in section 3.3. In reality two A-Terms will not be perfectly equivalent, because during the normalization of Statix certain sugar constructs get transformed into core constructs by introducing new variables, which will have some unique random name. This means we have to check for alpha-equivalence instead of full equivalence when comparing normalized Statix rules.

To perform this equivalence checking we used Stratego, since it allows for a straightforward comparison of the two compilation pipeline results. The steps taken to compare a single Statix AST are as follows:

1. Execute both methods of analysis on the AST and obtain the resulting two analysis results.

2. Perform the normalization step on a separate instance of the AST using both analysis results and their corresponding transformations in order to obtain two A-Terms.

3. Compare the lists of imports, edge labels, relations and scope graph extensions of both A-Terms by converting them to sets and checking for set equivalence.

4. Compare the lists of rules for alpha-equivalence.

The steps described above can be seen in Figure 5.4. The generated ATerm is used to statically analyze a language file to which the Statix specification on the left of the diagram belongs to. The ATerm is part of the input given to the Statix solver, as is described in section 3.1 and is shown in Figure 3.1.

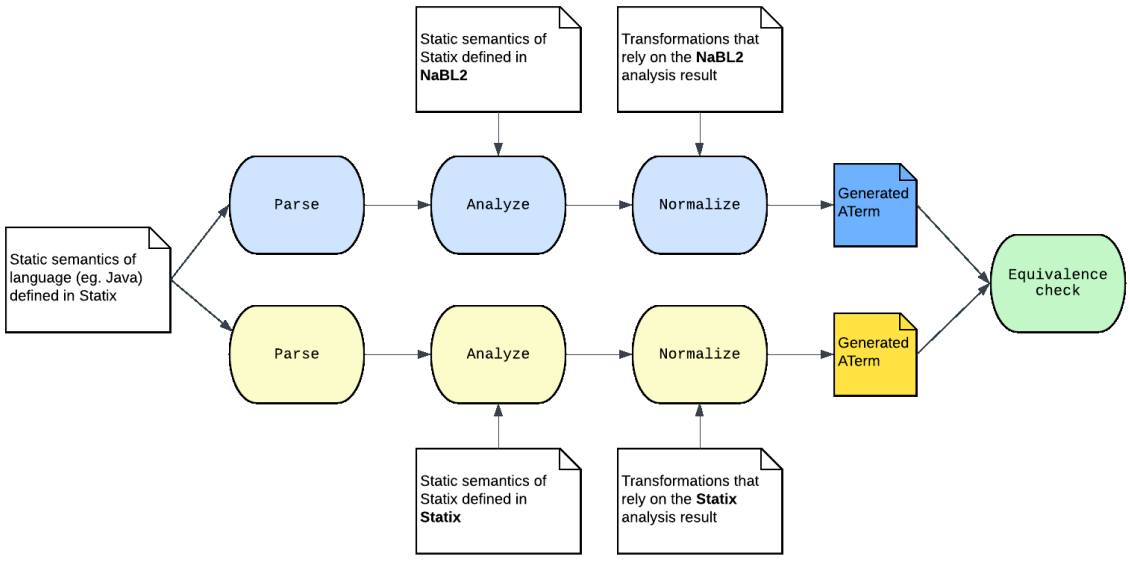

Figure 5.4: The equivalence check progress represented by a diagram

We used the comparison described above in order to compare the normalized Statix specifications of already established language projects as well as the Statix specification of Statix we created ourselves. These projects of differing size and complexity act as a test set of realistic Statix definitions. In Table 5.5 details can be found about the size of the Statix specifications. The following information is given:

- The number of Statix modules in which the rules of the Statix specification are given. In brackets we give the number of modules that contain the signatures of the language constructs, which are generated from the grammar.
- The number of import edges in the scope graph that is created from the Statix constraints.
- The number of constructors within the Statix specification, of which a majority are the generated language construct signatures.
- The number of rules within the Statix specification.
- The number of lines of all the Statix modules combined (including signatures).

Language	Tiger	Chicago	Java	Statix
Modules (Signatures)	1 (14)	14 (15)	63 (66)	9 (10)
Import edges	23	73	637	75
Constructors	78	76	530	305
Rules	86	96	974	347
Lines	850	964	6950	1983
Equivalent	✓	✓*	-	✓*

* Equivalent when specialized queries are disregarded

Table 5.5: Numbers that reflect the size of several Statix specifications

We briefly describe the language projects that were used as input for the equivalence check:

Tiger

Tiger is a functional language used as an example in the book Modern Compiler Implementation in ML (Appel 1998). The language supports constructs such as branching, loops, arrays and records. The Statix specification for Tiger is given in a single module (excluding language construct signatures).

Chicago

Chicago is an experimental language that was created to test Statix. The language includes constructs such as functions, records, modules and imports. The Statix specification is divided into 14 modules and includes a majority of the language constructs of Statix.

Java

Java is a high-level object-oriented programming language which is frequently used in practice. The existing Statix specification provides the static semantics of Java version 8. This specification is significantly larger than the other 2, consisting of 63 modules.

The equivalence check was completely successful on the Tiger specification and mostly successful on the Chicago and Statix specifications. The only differences that were observed from the equivalence check were regarding the specialization of scope graph queries (Zwaan 2022). This is a relatively new addition to the Statix compiler and if have the equivalence check focus on the original queries only, no differences are observed. We cooperated with the author of the paper on specialization of scope graph queries and he had the following explanation for the observed results:

> The Statix backend uses compile-time optimization on queries (Zwaan 2022). At the heart of the optimization is a derivative-based approach to generate a deterministic finite-state automaton (DFA) from a regular expression (Owens, Reppy, and Turon 2009; Brzozowski 1964). Debugging has shown that the Statix implementation is non-deterministic. Sometimes, for the same input, different but equivalent DFAs are generated. Usually, it is the case that one of the DFA's has a duplicate state (similar to Owens, Reppy, and Turon (2009) fig. 4). This results in some redundancy in the representation of optimized queries, This is easy to observe for humans, but beyond the scope of this project to actually fix, as the root cause of the non-determinism is not yet identified.

Using the Statix specification of Statix from our bootstrapped compiler as an input for the equivalence check is similar to the method of *sound* bootstrapping proposed in (Konat, Erdweg, and Visser 2016). In this method you try to reach a fixpoint when comparing the binaries of multiple bootstrapping iterations. When we use the bootstrapped Statix specification as an input for the equivalence check, we are essentially comparing two iterations of bootstrapping, namely the first iteration that uses the NaBL2 specification of the baseline compiler to type check Statix files and a second iteration that uses the Statix specification of the first iteration to type check its Statix files. Instead of comparing binaries we compare the compiled Statix files, but we do reach a fixpoint after one iteration.

We were not able to perform the equivalence check on the Java specification however. This is due to the fact that solving the Java Statix specification using our bootstrapped compiler did not terminate. We looked into what was causing this issue and came to the following conclusion:

Statix allows for transitive imports of sorts, constructors and constraints. The Java Statix specification is made up of a large number of Statix modules and contains a complex import structure, that results in a dense scope graph containing many diamond imports (Figure 5.3). Combining these two facts results in a large number of possible paths when resolving queries, this is because the amount possible non-cyclic paths has an upper bound that grows exponentially relative to the size of the scope graph. The large number of possible paths produces a large overhead for the current Statix solver. When you combine this with the large amount of references that exist in the Java Statix specification it becomes clear why the analysis can not be solved within a reasonable amount of time.

5.2 Performance

In order to gain an impression on how the bootstrapped version of the Statix compiler performs, we conducted some benchmarking. We looked at the effect of several elements of the Statix compiler and compared the performance of the new version of the compiler to the NaBL2 based version of the compiler.

5.2.1 Benchmark setup

The benchmarking is performed using a Java script, in which we can register a time for each of the separate steps of the Statix compiler. The benchmarks are executed on a HP ZBook Studio G3 with an Intel i7-6700HQ processor. Every benchmark consisted of ten warm-up iterations and ten measurement iterations.

We use a total of six different versions of the Statix compiler for benchmarking. Both the non-bootstrapped NaBL2 based version of the compiler and the bootstrapped Statix based version of the compiler are used in order to compare them. For both types of the compiler we have three different versions, where in some versions features of the compiler are left out in order to see their influence on the compile time. The following three versions were used:

- A version of the compiler in which there are no rules for checking duplicate declarations of sorts, constructors, relations, constraints as well as checking whether there are no duplicate rule patterns.
- A version of the compiler which does have rules for checking duplicate declarations, but doesn't contain rules for checking whether there are duplicate rule patterns.
- A version of the compiler that has all features of the Statix compiler, including duplicate rule pattern checking.

As input for the benchmarking we used the Tiger and Chicago language projects, which we describe in section 5.1.3 and for which you can see details in Table 5.5. These two language projects are similar in size, but their biggest difference is that in the Tiger project all rules are given in one Statix module, while in the Chicago project the rules are spread out over multiple modules and a structure of imports is used. This results in the Chicago project having a more complex scope graph during analysis.

Version used	**Analysis time** (s)		**Normalization time** (s)	
	NaBL2	Statix	NaBL2	Statix
Not any duplicate checking	1.38	1.50	0.26	0.15
No duplicate rule pattern checking	1.46	1.66	0.25	0.15
All duplicate checks	1.51	2.03	0.24	0.15

Table 5.6: Results of benchmarking the Statix compiler on the **Tiger** language project

Version used	**Analysis time** (s)		**Normalization time** (s)	
	NaBL2	Statix	NaBL2	Statix
Not any duplicate checking	1.90	0.92	0.33	0.19
No duplicate rule pattern checking	1.96	4.06	0.32	0.21
All duplicate checks	2.33	54.77	0.34	0.25

Table 5.7: Results of benchmarking the Statix compiler on the **Chicago** language project

5.2.2 Results

In Tables 5.6 and 5.7 and Figures 5.8 and 5.9 the results of the benchmarking on the Tiger and Chicago language projects are shown.

Firstly we can observe that the analysis time of the bootstrapped compiler is often slower than the NaBL2 version of the compiler. This could suggest that the back-end of the Statix is

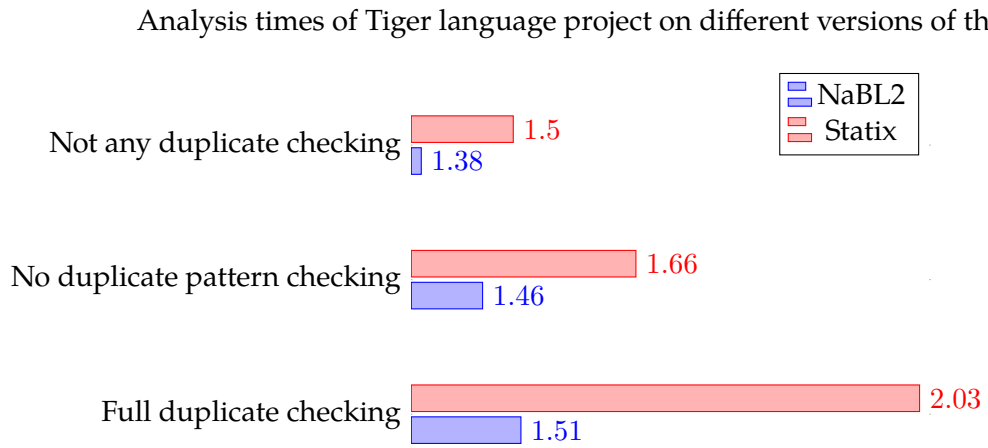

Figure 5.8: Compiler versions compared using Tiger language project

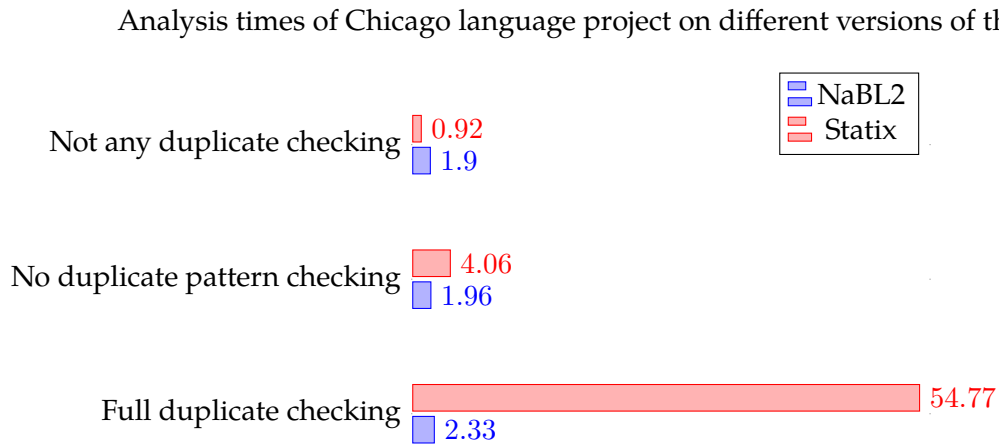

Figure 5.9: Compiler versions compared using Chicago language project

not as optimized as that of NaBL2, but it could also caused by the nature of our specification in Statix. Further research is required to determine what is causing the difference.

A second observation from the results is the fact that doing any form of duplicate checking requires more time during the analysis phase of the compiler, but this time increases a lot more for the Statix based version of the compiler compared to the NaBL2 based version. In the results for the Tiger and Chicago project there is a larger increase in analysis time when duplicate declaration checking is added to the compiler and an even larger increase when duplicate rule pattern checking is added. This can be explained by the fact that the duplicate rule pattern checking we have defined in the bootstrapped compiler (See section 4.2) requires a lot of constraints to be solved in order to compare every possible rule combination. In the NaBL2 version of the compiler the duplicate rule pattern checking analysis is not defined using NaBL2, but is checked as part of the custom analysis section of the compiler (section 3.2.1) using a functional approach in Stratego (Visser 2003), which is faster than solving a large number of constraints.

The increase in analysis time in the Chicago results is a lot greater than the increase in the Tiger results. The likely explanation for this behavior is that the Chicago project has a more complex scope graph, caused by the higher number of modules and import edges shown in

Table 5.5. This means that scope graph queries are harder to resolve, resulting in a larger analysis time.

Finally we can see that the normalization time is larger in the NaBL2 version of the compiler, but is barely affected by whether there is duplicate checking. It was expected that the normalization time wouldn't get affected by the presence of duplicate checking, because this is purely part of the analysis and none of the transformations in the normalization phase use that information. It is however interesting to see that doing the same transformations in the bootstrapped compiler takes less time for both language projects used. This could mean that using the Statix analysis object is faster than using the NaBL2 analysis object, but the more likely explanation is that the NaBL2 version includes deprecated features which required transformations, meaning there are extra traversals of the AST.

5.3 Bootstrapping process

In this section we summarize how we handled the challenge of bootstrapping Statix. We explain some of the choices that were made during the process and try to express the lessons learned from our experience.

Since the main focus of our bootstrapping process was making sure that the compiler was still correct and there was already a reference implementation, we could use existing tests as well as write new tests for the original compiler to verify the bootstrapped compiler. This meant we were able to use an approach that resembles test-driven development.

While reworking the transformations in the normalization step such that they function with the new analysis result, we came to the conclusion that most of the transformations of the original compiler can be reused as long as the type system stays the same. This is due to the fact that these transformations took an AST that was annotated with the types of terms and constraints as input and then relied on these type annotations to perform transformations instead of the analysis result. So as long as we could obtain the same type annotations from the analysis result, these transformations did not need to be altered.

During the definition of the static semantics of Statix in Statix we tried two approaches: looking at the existing static semantics definition in NaBL2 and translating this to Statix and studying the documentation of Statix and come up with a Statix definition from scratch. We found that the latter approach works better than the former, since the first approach requires you to get familiar with NaBL2 and the existing definition which is language specific, so adopting a similar approach in Statix might not work well. However the preexisting specification does provide a complete set of static semantics, so studying it after you have defined your own set of semantics can aid in making sure that your specification is complete as well.

Chapter 6

Related work

In this chapter we discuss work that is related to this thesis. We explore research related to the Statix meta-language as well as the topic of bootstrapping meta-languages.

6.1 The Statix Meta-language

In this section we will discuss papers that have contributed to or used the Statix meta-language. We mention what they focus on, summarize their approach and discuss their results.

The Statix meta-language was introduced by Antwerpen, Poulsen, et al. (2018). The aim of this paper is to prove that the scope graph framework introduced by Néron et al. (2015) supports the modeling of interesting name binding patterns in programming languages, such as structural and parameterized types. It accomplishes this by viewing the scopes in a scope graph as an actual type. The scope graph model is extended with scope relations and scope graph resolution queries with their own visibility policies. All these extensions of the scope graph model are then incorporated in Statix, for which the paper provides the declarative semantics. The scopes-as-types approach and the Statix language are evaluated using case studies of languages that contain the following name binding patterns: structural subtyping, parametric types and generic class types, showing that the scope graph model has the potential to the potential to standardize the treatment of name-binding in programming languages.

Our work obviously heavily builds on Scopes as types, since our work is in a large part a case study on whether the Statix language, which has incorporated the scope graph model, can be used for a constraint-based declarative language, namely Statix.

The paper by Rouvoet et al. (2020) expands on the work of Antwerpen, Poulsen, et al. (2018) and focuses on the execution of Statix. The main purpose of the paper is to ensure answer stability of name resolution queries during the solving of Statix constraints. The paper distills and refines the core aspects of Statix into Statix-core and provides operational semantics such that queries are scheduled in a way that answer stability is guaranteed. A proof is given that these operational semantics are sound, which relies on the type system of Statix having a permission to extend a scope.

The permission to extend a scope has been incorporated into the Statix compiler as mentioned in section 3.2.3. In order we to successfully bootstrap the Statix compiler we had to make sure that this analysis was still functional after bootstrapping, the explanation on how we accomplish this is given in section 4.3.

We will now briefly mention other papers that work on or with Statix. The paper by Zwaan (2022) looks at speeding up Statix based type checkers by specializing scope graph resolution queries using partial evaluation. Pelsmaeker et al. (2022) presents a language parametric code completion editor service that relies on a Statix specification and the Statix

constraint solver. Misteli (2020) looks at how you can develop language parametric refactorings by creating a program model using the static semantics specification of a language in Statix.

6.2 Bootstrapping Meta-languages

In this section we will look at papers that focus on bootstrapping meta-languages of language workbenches.

The paper by Konat, Erdweg, and Visser (2016) provides a detailed analysis of the bootstrapping problem of language workbenches and provides a method for sound bootstrapping. The Spoofax language workbench and more specifically the syntax specification language SDF3 are used to show the intricate dependencies encountered when bootstrapping a meta-language using a language workbench. A sound method for bootstrapping is described that relies on meta-language compilation reaching a fixpoint. Besides this method, the paper also provides the details on having an interactive bootstrapping environment that supports breaking changes and how to decompose those. The approach has been implemented in the Spoofax language workbench and evaluated by bootstrapping eight interdependent meta-languages.

As mentioned in section 5.1.3, we believe that our method of validating the correct bootstrapping of Statix is similar to the sound bootstrapping proposed in Konat, Erdweg, and Visser (2016). We did not make use of an interactive bootstrapping environment to compare the binaries bootstrapping iterations, but used a method that checks for equivalence between compiled specifications.

Prinz and Alexander Shatalin (2019) discuss the requirements of bootstrapping a language workbench and therefore bootstrapping individual meta-languages. It shows that in order for a successful bootstrapping of a meta-language to occur, it is important to take into account the dependencies and references it has to other meta-languages of the language workbench. An example is given of the bootstrapping of the LanguageLab language workbench (Gjøsæter and Prinz 2015), and it is shown that there are dependencies and references between the structure (grammar), generator (transformations) and editor meta-languages, which need to be dealt with during different steps of the bootstrapping process.

As discussed in chapter 3 the Statix language also has dependencies to multiple meta-languages, which are the structure and generator meta-languages and itself. Our work doesn't go in depth on how these dependencies would affect the overall bootstrapping of the Spoofax language workbench, but instead focuses more on the specifics on how a constraint-based language for static semantics is bootstrapped.

Prinz and Mezei (2020) the observations made about the bootstrapping situation of language workbenches in Prinz and Alexander Shatalin (2019) are confirmed by comparing the bootstrapping of four language workbenches, namely Eclipse Modeling Framework (EMF) (Steinberg et al. 2009), JetBrains Meta Programming System (MPS) (Pech, Alex Shatalin, and Völter 2013), LanguageLab (Gjøsæter and Prinz 2015) and Dynamic Multi-Layer Algebra (DMLA) (Mezei et al. 2019). In this paper several observations are made about the bootstrapping of language workbenches and one of them is that at first glances meta-languages have a very tight connection between one and other, which would make bootstrapping difficult, but at the concept level there are often a few core concepts that are self-referential and mutually referencing each other.

Chapter 7

Conclusion

In this chapter we will draw conclusions based on our work and give recommendations for future work.

The purpose of this thesis was to obtain a correct bootstrapping of the Statix meta-language. We accomplished this by providing a Statix specification of the static semantics of Statix as well as by adapting the transformations within the Statix compiler. We have shown that the Statix compiler within the Spoofax language workbench can successfully be bootstrapped without its behavior changing from the reference solution. This shows that Statix can be used to define the static semantics of a declarative constraint-based language.

Although we claim to have a correctly bootstrapped Statix compiler, it is currently not yet a viable option to use in practice. This is due to the fact that using the bootstrapped compiler to compile more complex large Statix language projects takes an excessive amount of time. The reason for this is that Statix supports transitive imports which can cause an exponential growth in the amount of resolution paths to consider and the current Statix back-end is not prepared to deal with that.

7.1 Future work

We recommend that in order to start using the bootstrapped Statix compiler within Spoofax, future work should focus on improving the performance of the compiler. We believe that this can be achieved by critically looking at the Statix constraint solver and adapting it such that the solving time can not exponentially grow. As well as looking at improving the Statix back-end, the static semantics specification of Statix in Statix could be adapted to improve the performance. Once the performance has been improved such that larger language projects such as the Java project can be built, the correctness validation method used in our work could be used on these projects as well in order to verify whether the correctness is still valid for larger projects.

Another suggestion for future work is looking at whether the bootstrapped Statix compiler can be used to add generics to the Statix language. It has been shown that Statix can be used to express parametric types (Antwerpen, Poulsen, et al. 2018) and this would be a welcome addition to the language.

Finally another direction for future work would be incorporating the bootstrapped version of Statix as part of the boostrapping of the latest version of the Spoofax language workbench, Spoofax 3.

Bibliography

Antwerpen, Hendrik van, Pierre Néron, et al. (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, et al. (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.

Appel, Andrew W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press. ISBN: 0-521-58274-1.

Brzozowski, Janusz A. (1964). "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4, pp. 481–494.

Gjøsæter, Terje and Andreas Prinz (2015). "LanguageLab - A Meta-modelling Environment". In: *SDL 2015: Model-Driven Engineering for Smart Cities - 17th International SDL Forum, Berlin, Germany, October 12-14, 2015, Proceedings*. Ed. by Joachim Fischer et al. Vol. 9369. Lecture Notes in Computer Science. Springer, pp. 91–105. ISBN: 978-3-319-24911-7. DOI: 10.1007/978-3-319-24912-4_8. URL: http://dx.doi.org/10.1007/978-3-319-24912-4_8.

Janssen, Boris et al. (Apr. 2023). *Thesis Boris Janssen: Bootstrapped Statix*. Version thesis. DOI: 10.5281/zenodo.7799225. URL: https://doi.org/10.5281/zenodo.7799225.

Kats, Lennart C. L., Rob Vermaas, and Eelco Visser (2011). "Integrated language definition testing: enabling test-driven language development". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, pp. 139–154. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048080. URL: http://doi.acm.org/10.1145/2048066.2048080.

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.

Konat, Gabriël, Sebastian Erdweg, and Eelco Visser (2016). "Bootstrapping Domain-Specific Meta-Languages in Language Workbenches". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Bernd Fischer and Ina Schaefer. ACM, pp. 47–58. ISBN: 978-1-4503-4446-3. DOI: 10.1145/2993236.2993242. URL: http://doi.acm.org/10.1145/2993236.2993242.

Mezei, Gergely et al. (2019). "Towards Mainstream Multi-level Meta-modeling". In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*. Ed. by Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic. SciTePress, pp. 481–488. ISBN: 978-989-758-358-2. DOI: 10.5220/0007580404810488. URL: https://doi.org/10.5220/0007580404810488.

Misteli, Philippe D. (2020). "Towards language-parametric refactorings". In: *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*. Ed. by Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix. ACM, pp. 213–214. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3398476. URL: https://doi.org/10.1145/3397537.3398476.

Néron, Pierre et al. (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.

Owens, Scott, John H. Reppy, and Aaron Turon (2009). "Regular-expression derivatives re-examined". In: *Journal of Functional Programming* 19.2, pp. 173–190. DOI: 10.1017/S0956796808007090. URL: http://dx.doi.org/10.1017/S0956796808007090.

Pech, Vaclav, Alex Shatalin, and Markus Völter (2013). "JetBrains MPS as a tool for extending Java". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. Ed. by Martin Plümicke and Walter Binder. ACM, pp. 165–168. ISBN: 978-1-4503-2111-2. DOI: 10.1145/2500828.2500846. URL: http://doi.acm.org/10.1145/2500828.2500846.

Pelsmaeker, Daniël A. A. et al. (2022). "Language-parametric static semantic code completion". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA, pp. 1–30. DOI: 10.1145/3527329. URL: https://doi.org/10.1145/3527329.

Prinz, Andreas and Gergely Mezei (2020). "The Art of Bootstrapping". In: *Model-Driven Engineering and Software Development*. Cham: Springer International Publishing, pp. 182–200. ISBN: 978-3-030-37873-8.

Prinz, Andreas and Alexander Shatalin (2019). "How to Bootstrap a Language Workbench". In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*. SciTePress, pp. 345–352. ISBN: 978-989-758-358-2. DOI: 10.5220/0007398203450352. URL: https://doi.org/10.5220/0007398203450352.

Rouvoet, Arjen et al. (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.

Souza Amorim, Luis 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.

Steinberg, Dave et al. (2009). *Eclipse Modeling Framework*. 2nd ed. Addison-Wesley.

Visser, Eelco (2003). "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. Ed. by Christian Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Springer, pp. 216–238. ISBN: 3-540-

22119-0. DOI: 10.1007/978-3-540-25935-0_13. URL: https://doi.org/10.1007/978-3-540-25935-0_13.

Zwaan, Aron (2022). "Specializing Scope Graph Resolution Queries". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, pp. 121–133. ISBN: 978-1-4503-9919-7. DOI: 10.1145/3567512.3567523. URL: https://doi.org/10.1145/3567512.3567523.