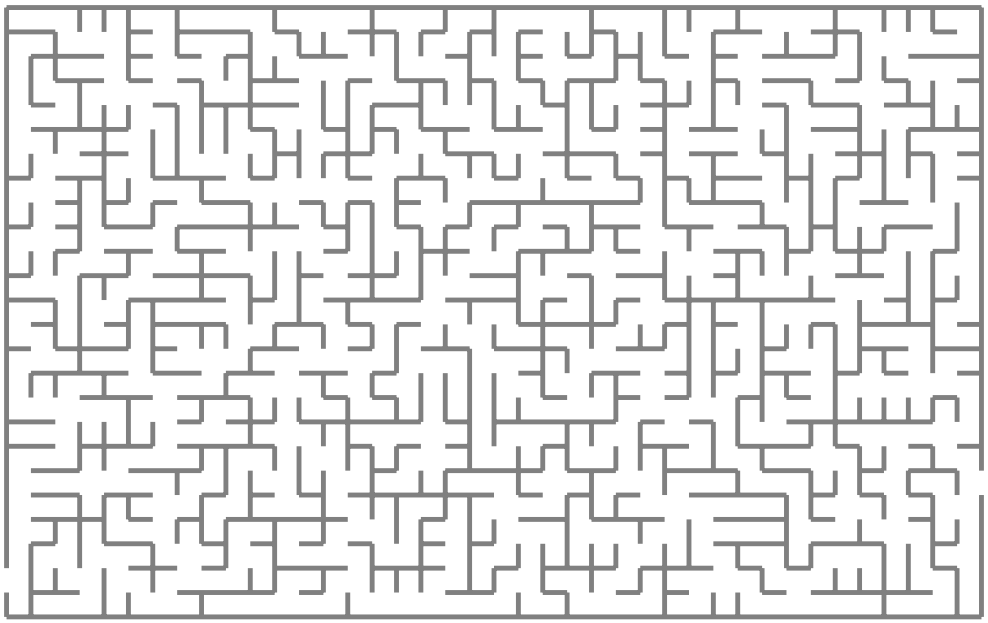
Master Thesis Template

Version of September 22, 2022



Boris Janssen

Master Thesis Template

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

© 2022 Boris Janssen.

Cover picture: Random maze.

Master Thesis Template

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

Abstract

Abstract here.

Thesis Committee:

Chair:	Prof. dr. C. Hair, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bee, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Dee, Faculty EEMCS, TU Delft
University Supervisor:	Ir. E. Ef, Faculty EEMCS, TU Delft

Preface

Preface here.

Boris Janssen
Delft, the Netherlands
September 22, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contributions

1.2 Thesis Outline

Introduction here.(Sústrik 2012)

Chapter 2

The Statix Meta-language

Statix (Antwerpen et al. 2018)

2.1 Statix as part of the Spoofax language workbench

Statix is a meta-language that is part of the Spoofax language designer workbench, 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 rules of a language that can be checked at compile time. Two other languages that are part of Spoofax and play an important role in language design are: SDF3, which is used specify the syntax of a language and Stratego, which is used to perform transformations on intermediate representations of a language.

The role Statix of in Spoofax is to specify a set of constraints on an abstract syntax tree (AST) of the language that represent the static semantics of the designed language. The intermediate representation is usually a parsed and possibly desugared AST of the designed language. The Statix specification gets normalized (see chapter 3) to a spec file format, which can together with the object language AST be fed to the Statix solver to be solved. The Statix solver results in an analysis result that contains type information, such as the types of terms and possibly error messages when the constraints can't be fully solved.

2.2 Introduction to Statix

2.2.1 Typing rules

2.2.2 Scope Graphs

Chapter 3

The Statix Compiler

3.1 Compilation pipeline

3.2 Static Analysis

3.3 Custom Analysis

3.4 Normalization

info here.

Chapter 4

Bootstrapping: What is it & Why do it?

4.1 What is bootstrapping?

4.2 Why bootstrap Statix?

info here.

Chapter 5

Bootstrapped Implementation

5.1 Typing & Name Binding of Statix

In this section we will describe the typing system of Statix and how this type system is used with regards to name binding and name resolution in Statix. We will cover the numerous names that can be defined in Statix and explain the motivation behind their name binding in the bootstrapped version of Statix. We we will also show how types in statix are used to type check statix using an example.

5.1.1 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 5.1 are defined. A majority of these types don't 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 made up 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 : SortId -> TType
```

Figure 5.1: Term Types of Statix

```
sorts
  IType

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

Figure 5.2: Predicate Types of Statix

5.1.2 User Defined Constraints

User defined constraints can't be expressed as term type, but they do have a type signature that consists of term types. This is why a set of types is defined to represent constraints, which can be seen in Figure 5.2. A basic constraint or predicate type is composed of a list of the term types of its parameters, whilst 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 ($constraintId \rightarrow IType$). All non mapping constraints in Statix are declared with their type signature, which makes their name binding straightforward. A constraint's type is resolved when either a rule for it is defined or it is used as an inline constraint or term.

5.1.3 Variables

Variables that are used in rules are defined by a relation that maps the variable identifier to a term type ($variableId \rightarrow TType$). Variables can be declared either by being part of a rule pattern or they can be introduced in an exists 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 be used 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 existentially quantified 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 an existentially quantified variable instead of an already derived type, but because Statix constraints are solved to a fixpoint, the remaining constraints will bind this variable to the correct type and the scope-graph declarations and queries will match.

5.1.4 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 ($sortId \rightarrow TType$), for a non-aliased sort this term type will be of type sort with the newly declared identifier as its parameter, while for aliased sorts 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

```
sorts
  LType

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

Figure 5.3: Label Types of Statix

$(constructorId * int \rightarrow (list(TType) * TType))$. 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 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.

5.1.5 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 can't have overlapping identifiers due to the fact that they both give names to elements of a scope-graph and the Statix solver doesn't have a way to differentiate between them. We could have also chosen to use two seperate relations, but that would mean that checking whether you are using a duplicate name requires two scope-graph queries instead of one, which is more efficient. To differentiate between relations and scope-graph edge labels we have defined a label type that you can see in Figure 5.3, and the relation maps identifiers to these label types $(identifier \rightarrow LType)$. 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.

5.1.6 Modules

The modules of Statix are also defined in the scope-graph using a relation that pairs the module identifier with a scope $(moduleId \rightarrow scope)$. The scope is the module scope in which all constraints, sorts, constructors, relations and labels get 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 isn't a functional relation from identifier to scope, because it is also queried using a scope to find the corresponding identifier such that it can be used for error messages or qualifying the name of a predicate or label reference.

5.1.7 Type Checking Example: ListTail Term

The types and scope-graph relations described in the previous subsections are used to type-check 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 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 won't type check. The statix code for this type checking behaviour can be seen in Figure 5.4.

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

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

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

Figure 5.4: Statix code relevant to type checking a ListTail term

5.2 Duplicate Pattern Checking

5.3 Scope Extension Behavior

5.4 Normalization

Chapter 6

Evaluation

6.1 Correctness

6.1.1 Unit testing

6.1.2 Equivalence checking

6.1.3 Integration testing

6.2 Performance

6.2.1 Error message behavior

6.2.2 Bench-marking

Chapter 7

Related work

Related work here.

Chapter 8

Conclusion

Conclusion here.

Bibliography

Antwerpen, Hendrik van 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.

Sústrik, Martin (2012). *ZeroMQ*. URL: http://aosabook.org/en/zeromq.html (visited on 02/16/2018).

Acronyms

AST abstract syntax tree

DSL domain-specific language