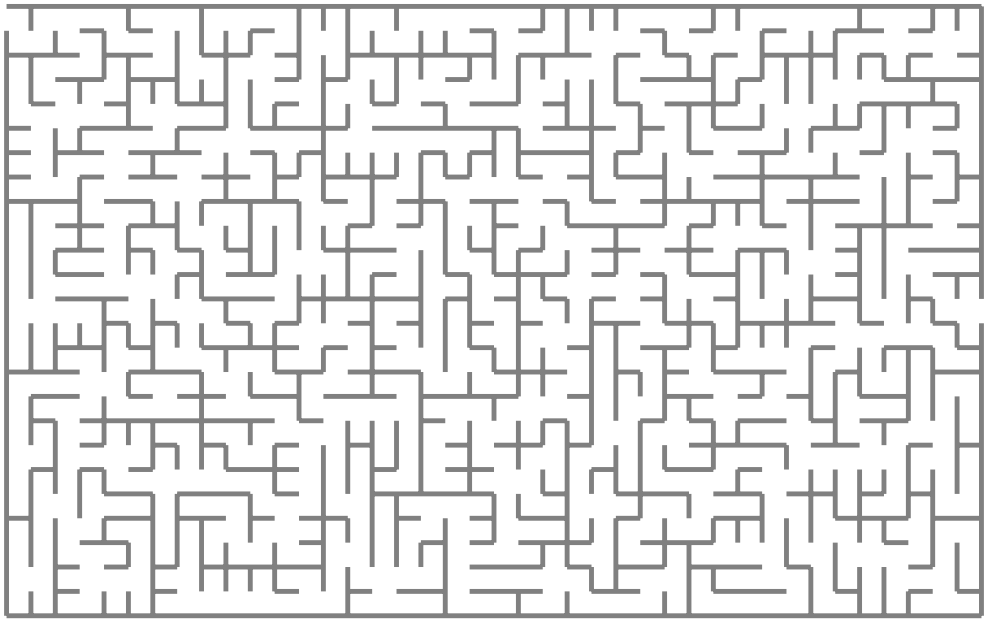
Modernizing the WebDSL front-end: A case study in SDF3 and Statix

Version of May 16, 2022



Max Machiel de Krieger

Modernizing the WebDSL front-end: A case study in SDF3 and Statix

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Max Machiel de Krieger
born in Delft, the Netherlands

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

Cover picture: Random maze.

Modernizing the WebDSL front-end: A case study in SDF3 and Statix

Author: Max Machiel de Krieger
Student id: 4705483
Email: M.M.deKrieger@student.tudelft.nl

Abstract

WebDSL is a domain-specific language for web programming that is being used for over ten years. As web applications evolved over the past decade, so did WebDSL. A complete formal specification of WebDSL has been *TO-DO: check if missing or not updated* since its original development. With the introduction of Statix in the Spoofax Language Workbench, a declarative language that generates a typechecker, we made an elegant and practical formal semantics for WebDSL.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Katsifodimos, Faculty EEMCS, TU Delft
University Supervisor:	Ir. D. M. Groenewegen, Faculty EEMCS, TU Delft
Expert:	Ir. A. Zwaan, Faculty EEMCS, TU Delft

Preface

Preface here.

Max Machiel de Krieger
Delft, the Netherlands
May 16, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programming is an essential skill that is increasingly important in diverse disciplines (Rafalski et al. 2019). To this end, many different programming languages exist, each with different properties and advantages. Over time, the populatity of programming languages change and developers tend to have preferences for one language over the other. In addition to language design choices, the implementation of a language and the tools that come with it can greatly boost the productivity of developers, if done well.

Add content about general purpose languages and domain-specific languages

When inspecting the design and implementation of a programming language, the different components can be classified in two boxes: the front-end and the back-end. The front-end is the part of the programming language that the developer faces directly, it consists of components such as the syntax and early feedback on written code. The back-end makes the programming language operational with, for example, code generation and opmtimization. While the back-end of a programming langauge makes it work, the front-end plays a large role in how developers experiences a programming language. Early feedback in the form of good error messages and hints are required to make the interaction with a programming language efficient (Becker et al. 2019).

Add content about challenging features in (domain-specific) language development

Programming languages are constantly evolving, requiring updates to its specification and implementation. One such language is WebDSL. WebDSL is a domain-specific language for developing web applications, developed at and maintained by the Programming Languages research group of the Delft University of Technology.

Because of its academic nature, many research projects added features to the language, all contributing to the success of existing WebDSL applications. With these features, WebDSL is the programming language in which applications are developed with thousands of daily users. The downside of having many different contributors adding new features and a small group of maintainers, is that the development experience, that is a result of the front-end, leaves much to be desired. Currently, the WebDSL implementation is composed of multiple definitions in meta-DSLs supported by the Spoofax Language Workbench: the syntax is defined in SDF2 and the desugaring, typechecking, optimization and code generation is defined in the term transformation language Stratego. The interleaving of all the latter processes in the same Stratego environment poses a threat to the readability and maintainability of the WebDSL language.

Add content about our approach here: using Statix and SDF3, see if the languages are fit for our purposes

Add research questions here

1.1 Contributions

In this thesis, we will be focusing on modernizing the WebDSL front-end, by implementing the syntax definition in SDF3 and the static analyses in Statix and documenting the challenges faced in this process. In this work, the following contributions are made:

- We present a modernized WebDSL front-end through an implementation of its grammar in SDF3 and its analyses in Statix.
- We document the challenges and solutions of implementing the new WebDSL front-end.
- We assess the completeness of Statix and SDF3 by attempting to model all language features of WebDSL.
- We assess the performance of Statix and SDF3 by benchmarking the new WebDSL front-end with large codebases of existing applications.
- We provide qualitative feedback regarding the development experience with SDF3 and Statix. *TO-DO: Change "development experience" to something more descriptive*

1.2 Outline

The rest of this thesis is structured as follows. In Chapter 2 we describe WebDSL, its features and its current implementation. Next, Chapter 3 and Section 4.1 go in detail about the new implementation of the WebDSL front-end in SDF3 and Statix respectively. The result of this implementation is evaluated in Chapter 5 and compared with related work in Chapter 6. Finally, Chapter 7 concludes this thesis.

Chapter 2

WebDSL

In this chapter, we describe WebDSL. WebDSL is a domain-specific language for developing web applications. The language incorporates ideas from various web programming frameworks and produces code for all tiers in a web application (Groenewegen, Chastelet, and Visser 2020). Ever since its introduction over 10 years ago (Visser 2007), WebDSL has been the subject of many published papers (cite some papers here) and on top of that, is the programming language underpinning several applications used daily by thousands of users. Examples of WebDSL applications include but are not limited to:

- **WebLab**: An online learning management system, used by the Delft University of Technology.
- **conf.researchr.org**: A domain-specific content management system for conferences, used by all ACM SIGPLAN and SIGSOFT conferences.
- **researchr.org**: A platform for finding, collecting, sharing, and reviewing scientific computer science related publications.

The rest of this chapter showcases the different aspects of WebDSL and zooms in on its non-trivial features. First, in Section 2.1 we will describe how WebDSL offers functionality for creating web user interfaces. Next, in Section 2.2 we illustrate how the langauge manages data models. Thirdly, Section 2.3 contains information about WebDSL's solution for access control and in Section 2.4 we highlight interesting aspects of its general-purpose object oriented function code. We concluse this chapter by going in detail about WebDSL's current implementation in Section 2.5.

2.1 User Interfaces

Introduction

2.1.1 Building blocks and Syntax

Domain specific language for web applications -> the UI is how the user interacts with the application.

Page is the entry point, arguments are clean URL parameters.

Templates are reusable components that can be inserted on pages or in other templates.

Short example with three boxes next to each other (WebDSL code left, resulting HTML right, resulting UI bottom):

Functionalities for in example:
- Pages
- Templates
- Navigate
- Text
- Divs
- HTML elements

2.1.2 Request processing and Action Code

With the building blocks of the previous subsection, only static pages can be made.

Need HTML forms and submits to manipulate data.

WebDSL abstracts over the usual manual request processing by using forms, inputs and action code.

Functionalities for in example:
- Form
- Multiple input sorts (boolean, string, text)
- Action with different redirects based on boolean, pass string to new page

2.1.3 Template Overriding and Overloading

2.1.4 Dynamically scoped redefines

2.1.5 Ajax

2.2 Data Model

- Syntax
- Inheritance
- Extending entities

2.3 Access Control

- Syntax
- Inferred visibility
- Nested rules
- Pointcuts

2.4 Functions

- Syntax
- Entities as classes
- Hooks for entity setters
- Extending functions

2.5 Current Implementation

2.5.1 Spoofax Language Workbench

- History
- Goal
- Achievements

2.5.2 Current Implementation of WebDSL

- Large Stratego specification where desugaring, static analysis, optimization and code-generation are interleaved (exaggeration?)
- Side effects using dynamic rules.
- Unexpected consequences of changes due to limited static analysis in untyped setting.

Go over some interesting WebDSL features and how they are implemented:

- Access control
- Template overloading and overriding
- Entity extension

2.6 Modernization goal

- A complete and maintainable SDF3 and Statix specification of WebDSL.
- Gather insight into the capabilities, elegance and performance of SDF3 and Statix.
- (Incrementalization for free leveraging the parallel Statix solver)

Chapter 3

WebDSL in SDF3

Goal: New specification of WebDSL grammar. Stay compatible with all existing WebDSL code; as few breaking changes as possible.

Goal: Large case study for SDF3.

Outline of chapter

3.1 WebDSL Grammar Specification

The current grammar of WebDSL is specified in SDF2, the predecessor of SDF3.

The WebDSL grammar specification consists of <> files with <> productions in total.

Parts of the syntax are deprecated but still mainainted for backwards compatibility reasons.

Some productions added for the sake of autocompletion.

Reason to switch to SDF3:
- Modern spoofax does not support SDF2 anymore?
- SDF3 more performant?

3.2 Introduction to SDF3

Able to declaratively specify the complete syntax of a programming language in SDF3, and a parser, highlighter and pretty-printer gets generated from this specification.

3.2.1 Syntax

- Lexical sorts
- Context-free sorts
- Constructors
- Injections
- Optional sorts
- Repitition

3.2.2 Disambiguation

Possibilities:
- Prefer/avoid annotations on constructors (deprecated)
- Declare priorities of nested constructors
- Reject keywords
- Reject nesting of certain constructors

3.3 Migration from SDF2 to SDF3

There is a tool to migrate SDF2 specifications to SDF3 specifications but it does not work in all cases. Some work needs to be done to prepare the SDF2 specification for the migration, and some work needs to be done on the resulting SDF3 specification to make sure it is as usable as the old SDF2 specification.

3.3.1 Preparing the WebDSL SDF2 definition for migration

The SDF2 to SDF3 migration tool does not accept ”sorts” sections.

Alternations must be removed from the SDF2 specification. Solution is to introduce a separate sort for the alternation:

Before:
(”B” | ”C”) -> A cons(”A”)

After:
BorC -> A cons(”A”)
”B” -> BorC cons(”B”)
”C” -> BorC cons(”C”)

Restrictions (both context-free and lexical) produce an error during transformation and must be manually copied.

Mixed languages and parameterized imports are currently not supported in SDF3, so WebDSL code cannot be mixed with Stratego/Java code in the new SDF3 syntax definition.

Mixed languages were only used in the compiler, except for HQL which is used in WebDSL code. Fortunately, the HQL syntax is not used elsewhere and could be transformed to be a part of the WebDSL syntax natively.

3.3.2 Manual Tweaking of Generated WebDSL SDF3

Missing and duplicate constructors

In SDF3, the constructors are a much more key part of the productions than in SDF2, where constructors are defined as a `cons("MyConstructor")` annotation on the production. In the WebDSL SDF2 definition, some constructors were missing and there were many duplicate constructors that denoted alternative syntax for the same construct, essentially providing syntactic sugar.

In the newly generated SDF3, duplicate constructors had to be changed, in order for them to be unique. Additionally, missing constructors had to be added, preferably even for injections for a reason we will touch on later in Section 3.4.

Priority chains

To indicate priority amongst context-free productions, both SDF2 and SDF3 use the concept of priority chains, but the SDF2 variant requires a repitition of the production inside the chain, whereas SDF3 uses a reference to the sort with corresponding constructor. This causes the SDF2 priority chains to not be migrated to the SDF3 priority chains.

The only way to tackle this issue is to manually re-enter the priority chains in SDF3.

Transferring comments

Of a lesser importance, but highly recommended for the readability of a syntax definition is the comments, that are parsed as layout and are therefore not tranfered to the generated SDF3 files. Again, there is no way around this and they have to be manually transferred.

Template productions

A major change in SDF3 compared to SDF2 are template productions, that allow for nice pretty printing and syntactic code completion. The productions in the generated SDF3 files are all template productions, but do not have the proper surrounding layout and indentation because there is no way to extract this information from the SDF2 source. This had to be manually added to the generated SDF3 productions where applicable.

Deeply embedding HQL

Previously, the syntax definition of HQL was a standalone definition, and was used in the WebDSL SDF2 through parameterized imports. SDF3 has no support for this feature, so as discussed in Section 3.3.1, the language has to be transformed to be a part of the WebDSL syntax.

Deeply embedding the HQL syntax in the WebDSL syntax causes some errors to arise on duplicate names of sorts and constructors, this had to be fixed manually.

3.4 Preparation for Statix

With the intention to use Statix for implementing the WebDSL static analyses, the grammar sorts and constructors have strict requirements. Statix is a strongly typed language and requires all input to adhere to the declared sorts and constructors.

3.4.1 Sorts and Constructors in Statix

Statix takes an abstract syntax tree as input.

In the signature definition of Statix rules, it must be stated what the input and output sorts are. The implementation of the rules are defined over the constructors that belong to the sorts in the rule's signature.

Demo: Left top a few SDF3 productions, right top an abstract syntax tree, bottom statix sorts, constructors and a few rules.

All sorts and constructors that rules are defined over, have to be defined in the Statix code. In our case, a complete redefinition of all sorts and constructors in the Statix code is necessary to statically analyze the all WebDSL language features.

Unlike SDF3 and Stratego, Statix is statically typed and does not support injections or polymorphism in its constructors, which leaves some abstract syntax trees generated by the parser unable to serve as input for static analysis.

3.4.2 Statix Signature Generator

As mentioned and demonstrated in Section 3.4.1, Statix requires a definition of the constructors and sorts of the language to be analyzed. This definition exists in SDF3, but is not compatible with the Statix semantics since no injections are allowed. To prevent manual redefinition of the sorts in Statix code, the Statix Signature Generator is developped. This tool takes the SDF3 definition as input, and generates importable Statix files that contain the sorts and constructors from the syntax definition. For the Statix Signature Generator to work properly, Additional well-formedness requirements exist for the SDF3 definition.

Explicitly Declare Sorts

The parser generator that takes an SDF3 definition as input, is able to extract the sorts names from productions. Unfortunately, this is not the case for the Statix Signature generator so all sorts used in productions must be declared explicitly in `context-free sorts` and `lexical sorts` blocks.

TO-DO: Find reason and describe here

TO-DO: Example here with before and after with context-free and lexical sort blocks

Injections

With the semantics of Statix' constructors and sorts, it is not possible to model injections.

TO-DO: Example of injection here and impossibility of modelling in Statix

The Statix Signature Generator deals with simple injections (from one sort to one sort) by explicating the injections, making a more verbose version of the constructors and sorts.

TO-DO: Example here of simple injection explicated

Even though this functions properly, it is hard to read in the Statix rules. For this reason, we changed the WebDSL SDF3 to contain less injections by inserting constructors with descriptive names where injections were.

TO-DO: Show result

Optional Sorts

As mentioned in Section 3.2.1, SDF3 has built-in support for optional sorts, resulting in `Some(_)` and `None()` terms.

The resulting terms cannot be translated to Statix signatures, since this would mean a lot of duplicate Some and None constructors belonging to different sorts.

To resolve this challenge, the SDF3 definition must be altered make the Some and None constructors unique per sort. This leads to a much more verbose syntax definition:

TO-DO: Example here with two syntax definitions and resulting ASTs: one with built-in optional sorts, other with verbose optional sorts.

Disambiguation

As mentioned in Section 3.2.1, ambiguous code fragments lead to abstract syntax trees with the `amb(_, _)` term. Similar to optional sorts, this cannot be translated to Statix and therefore it is crucial that the syntax is disambiguated properly.

Next to this, the `prefer` and `avoid` annotations for disambiguation were heavily used in the WebDSL SDF2 definition. The annotations are supported in SDF3, but the support is likely to be dropped in a future release.

For the two reasons listed above, we reimplemented disambiguation through a combination of multiple SDF3 features. This process is explained in Section 3.5.

3.5 Disambiguation

Since the `amb(_,_)` constructor is not declarable in Statix, having an ambiguity in the AST leads to the analysis not executing. This increases the need for disambiguation.

Challenges and solutions:

- Keywords in WebDSL: SDF3 template options not optimal.
- String interpolation: Convert to one String constructor with a list of parts.
- Optional separators: In SDF2 multiple productions could have the same constructor, in SDF3 multiple constructors make for an increase in reject and desugaring rules.
- Optional alias vs. cast expression: use non-transitive priority rule.

3.6 Reflection on SDF3

Chapter 4

WebDSL in Statix

In this chapter, we elaborate on the implementation of the WebDSL static semantics in Statix, using the examples from Chapter 2 as a basis. We start this chapter by introducing the meta-DSL Statix. Once the goal and basics of Statix are stated, we describe the implementation of the type system that is the core of WebDSL. Next, we address and discuss the challenges faced while implementing non-trivial WebDSL features in Statix and lastly we reflect on the developer experience of using Statix to implement static analyses.

4.1 Introduction to Statix

Statix is a constraint-based declarative language for the specification of type systems, introduced in 2018 (Antwerpen et al. 2018). Since then, the meta-DSL Statix has become a part of the Spoofax Language Workbench and allows language developers to implement static analyses to provide language-specific feedback to developers on written code.

A Statix specification consists of rules over terms that define constraints. Additionally, Statix rules build and query a *scope graph* (Neron et al. 2015) that provides a language-agnostic representation of a program. A scope graph consists of nodes and edges that can be used to for example model the lexical scope of variables.

4.1.1 Language Signature

Consider a language consisting of booleans, integers and addition, for which we want to create a type-checker with Statix. First, Statix requires us to declare all types and sorts that we will be using in the rules. These Statix constructor names have to match the constructors of the input term (the AST). The Statix code that declares the sorts and constructors of our example language is shown in Figure 4.1. When writing a Statix specification for a language implemented in the Spoofax language workbench, it is a common practice to have the Statix signature generated from your SDF3 specification by the Statix signature generator (see Section 3.4.2), to prevent code duplication.

```
1  signature
2    sorts
3      Application
4      Exp
5
6    constructors
7      Application : Exp       -> Application
8      True        :              Exp
9      False       :              Exp
10     Int         : string    -> Exp
11     Add         : Exp * Exp -> Exp
```

Figure 4.1: Language signature in Statix

So far, our specification consists of two sorts. The `Application` sort defines the entry point of our language, it has one constructor with an identical name. Next, the sort `Exp` describes what expressions are allowed. It has four constructors: the Boolean values `True` and `False`, `Int` which requires an integer literal as subterm, and `Add` which takes two nested expressions as subterms. Examples of valid input according to our defined signature are shown in Figure 4.2.

```
Application(True())                    // true
Application(Int("42"))                 // 42
Application(Add(Int("40"), Int("2")))  // 40 + 2
Application(Add(Int("40"), False()))   // 40 + false
```

Figure 4.2: Valid input terms for the described language

4.1.2 Semantic Types

Not all of the valid input terms according to our signature are well-typed. For example, the last term shown in Figure 4.2 features an addition of the integer literal `40` and the Boolean value `False`. Using Statix' constraint solving capabilities, we would like to give feedback to the programmer that the input is ill-typed.

Given the code in Figure 4.1, our Statix specification does not yet generate any constraints. Constraints that we would like to generate using Statix rules are firstly that a program must be well-typed and secondly, in order for an addition expression to be well-typed, its two subterms must be of integer type.

To reason about the types of expressions and use them in constraints, we must first define them in our specification, as shown in Figure 4.3. To distinguish input sorts and constructors from semantic types that we will use in our constraints, those sorts and constructors are defined in upper-case. With the new `TYPE` sort that has two constructors: `BOOL` and `INT`, we can start generating constraints on input terms.

```
1 signature
2   sorts
3     TYPE
4
5   constructors
6     BOOL : TYPE
7     INT  : TYPE
```

Figure 4.3: Statix signature for Boolean and integer types

4.1.3 Predicates and Rules

Figure 4.4 lists the Statix predicates and rules required to generate the constraints we want to be satisfied in order for a program to be well-typed.

```
1 rules
2
3   applicationOk : Application
4   applicationOk(Application(e)) :- { T }
5     typeOfExp(e) == T.
6
7   typeOfExp : Exp -> TYPE
8   typeOfExp(True()) = BOOL().
9   typeOfExp(False()) = BOOL().
10  typeOfExp(Int(_)) = INT().
11  typeOfExp(Add(e1, e2)) = INT() :-
12    typeOfExp(e1) == INT(),
13    typeOfExp(e2) == INT().
```

Figure 4.4: Statix predicates and rules for typing booleans, integers and addition

The type of all Statix predicates must be explicitly declared, for example the `applicationOk` predicate on **line 3** specifies that all rules of `applicationOk` match exactly one constructor `Application`. An instantiation of the `applicationOk` predicate is on **line 4**. In prose English it would read "An application is well-typed, given that for some type `T`, the expression `e` has type `T`".

The other Statix rule in our small example specification is a *functional predicate*, meaning that it returns a value. All but the last rules of the `typeOfExp` predicate compute a `TYPE` for a given expression, without conditions. The last rule of the example does have two conditions, in prose English it would read "`e1` plus `e2` is of type `INT`, given that `e1` is of type `INT` and `e2` is of type `INT`".

4.1.4 Building and Querying Scope Graphs

When we expand our small example language with let-bindings and we want to add typing rules for this new construct, we come across a new feature in Statix. To facilitate typing rules for name binding, Statix uses *scope graphs* (Neron et al. 2015). Scope graphs are built out of three components: scopes, edges and declarations.

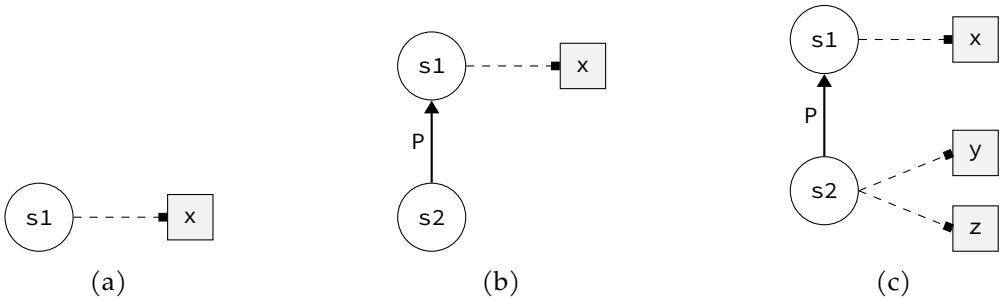


Figure 4.5: Scope graph examples

Figure 4.5 showcases three examples of scope graphs. Figure 4.5a consists of a single scope `s1` with declaration `x` that could be a model of a module with a single global variable `x` declared inside. The second example, Figure 4.5b, consists of two scopes: a root scope `s1` with again a declaration of `x`, and a scope `s2` with an outgoing edge to `s1` labeled `P`. The `P` label is often used to denote the relation of a lexical parent scope. In this example, `s2` could for example model an empty function declared in module `s1`. The last example again has two scopes, with one declaration in `s1` and two declarations in `s2`. This could model the same

program as described previously, but now with two local variable declarations inside the function body of `s2`.

The first step in implementing let-bindings in Statix is adding the signature. In addition to the new constructors on **line 3 and 4**, we now introduce an edge label `P` and the relation `var`. The edge labels defined in the constructor provide the set of allowed labels to use in rules later on. The relation `var` on **line 11** specifies that any declaration made under the `var` relation in a scope, maps an identifier to its type.

For illustration purposes, when we want to encode a single scope with two variable declarations, `x` of type `INT` and `b` of type `BOOL`, its scope graph would be as shown in Figure 4.7.

```
1  signature
2    constructors
3      Let : string * Exp * Exp -> Exp
4      Var : string             -> Exp
5
6    name-resolution
7      labels
8        P // to denote parent scope
9
10   relations
11     var : string * TYPE
```

Figure 4.6: Statix signature for let-bindings

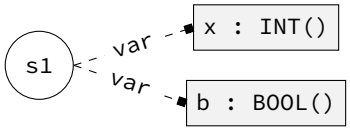


Figure 4.7: A scope graph containing a single scope with two declared variables

In Statix, scopes can be passed around as data. When we are evaluating an expression in our extended language, we now also want to pass the current scope. If the current input term that we are generating constraints for is a let-binding, we want to create a new scope, link it to the previous one, declare the variable in the new scope and evaluate the expression. To generate constraints for a variable expression, we want to query the scope graph and get its type. The Statix rules to reflect this are shown in Figure 4.8.

Figure 4.8 showcases various previously unexplained constructs:

- **Line 4** creates a new scope `s`. This scope is the root scope since it is created once at the start of an application and is not linked to any other scope.
- **Line 7** shows the new signature of the `typeOfExp` functional predicate. Given a scope and an expression, the rules of `typeOfExp` will compute the type of the expression.
- **Line 9-14** gives the typing rule of a let-binding. Given that the let-binding is of form `let x = e1 in e2`, the rule:
  - computes the type of `e1` on line 10;
  - creates a new scope `s_let` on line 11 for the body of the let to evaluate in;
  - declares variable `x` with associated type `T1` in the newly created scope `s_let`;
  - computes the type of `e2` and this is the result of the rule.
- **Line 16-21** holds the implementation of the variable typing rule. It executes a query with the following properties:
  - It only returns entries in the `var` relation (line 17)
  - It may follow zero or more `P` edge labels to other scopes (line 17);
  - It only returns declarations under the same identifier as `x` (line 18);

```
1  rules
2    applicationOk : Application
3    applicationOk(Application(e)) :- { s T }
4      new s,
5      typeOfExp(s, e) == T.
6
7    typeOfExp : scope * Exp -> TYPE
8    // ... previous rules
9    typeOfExp(s, Let(x, e1, e2)) = T2 :- { s_let T1 }
10     typeOfExp(s, e1) == T1,
11     new s_let,
12     s_let -P-> s,
13     !var[x, T1] in s_let,
14     T2 == typeOfExp(s_let, e2).
15
16   typeOfExp(s, Var(x)) = T :-
17     query var filter P*
18               and { x' :- (x', _) == (x, _) }
19               min $ < P
20               and true
21               in s |-> [(_, (_, T))].
```

Figure 4.8: Statix rules for let-bindings

- It prefers local declarations over declarations for which P edges must be followed (line 19);
- Shadowing according to the shadowing rules of line 19 is enabled (line 20);
- The query starts in the passed scope `s` (line 21);
- The result may only be one declaration (line 21).

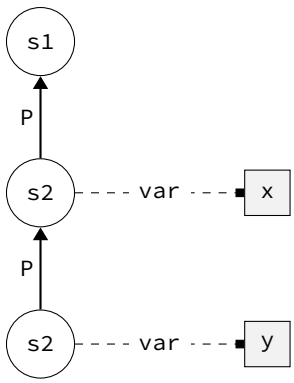
Figure 4.9 shows a possible input and the constructed scope graph after the constraints have been solved.

```
let x = 1 in
  let y = x in
    x + y
```

(a)

```
Application(
  Let(
    "x", Int("1"), Let(
      "y", Var("x"), Add(
        Var("x"),
        Var("y")
      )
    )
  )
)
```

(b)



(c)

Figure 4.9: Constructed scope graph after the example specification solved its contraints

4.2 Encoding the WebDSL Basics

The WebDSL language adheres to a structure similar to many popular programming languages. A WebDSL application consists of multiple files. At the topmost level in a file, there is a module or *unit* declaration. Within a module, multiple *sections* of *definitions* exist, such as pages, templates, entities and functions. A function consists of consecutive *statements* such as variable assignment (`var n := 2`). At the innermost level, these statements contain *expressions* that form the basis the WebDSL type system.

```
rules
  projectOk : scope
  unitOk    : scope * Unit
  sectionOk : scope * Section
  defOk     : scope * Definition
  typeOfExp : scope * Exp -> TYPE
```

Figure 4.10: Predicates that form the basis of the WebDSL Statix specification

To define well-typedness of the mentioned constructs, the Statix predicates as shown in Figure 4.10 form the backbone of the WebDSL Statix specification.

4.2.1 Built-in Types and Constant Expressions

Constant expressions such as strings, integers and booleans form the building blocks of more complication constructs. For reasons explained later (see section Section 4.4.2), a built-in type such as string is not declared as `STRING : TYPE` but instead as `BUILTINTYPE : scope * string -> TYPE`, where the instantiation of the string type is as follows: `BUILTINTYPE(s, "String")`.

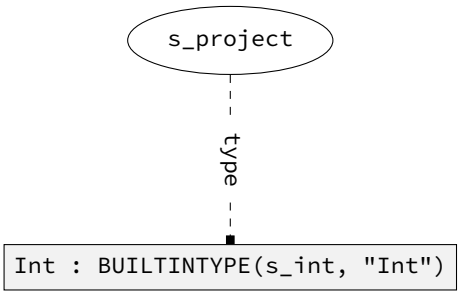
These built-in types are declared in a scope that is reachable from almost every location, the project scope, once per analysis. All WebDSL type declarations are made under the `type` relation, which associates the human readable type name with a `TYPE` term: `type : string * TYPE`. The part of the Statix specification to achieve this, and the resulting scope graph are shown in Figure 4.11.

```
projectOk(s_project) :-
  declareTypeBuiltIns(s_project).
  // ...

declareTypeBuiltIns : scope
declareTypeBuiltIns(s) :-
  declareType(s, "Int",
    BUILTINTYPE(new, "Int")).
  // ...

declareType : scope * string * TYPE
declareType(s, name, t) :-
  !type[name, t] in s.
```

(a)



(b)

Figure 4.11: Declaring built-in types in the project scope

To retrieve a built-in type when evaluating a constant expression, we need to query the scope graph and resolve the type associated with the string representation. For example, the typing rules of an integer constant are listed in Figure 4.12. The integer typing rule introduces a constraint

```
typeOfExp(s, Const(Int(_))) = t :-
  resolveType(s, "Int") == [(_, (_, t))].

resolveType : scope * string
  -> list((path * (string * TYPE)))
resolveType(s, name) = ts :-
  query type filter P*
            and { t' :- t' == (name, _) }
            in s |-> ts.
```

```
1 typeOfExp(s, Const(StringConst(String(str)))) = t :-
2   resolveType(s, "String") == [(_, (_, t))],
3   stringPartsOk(s, str).
4
5 stringPartsOk maps stringPartOk(*, list(*))
6 stringPartOk : scope * StringPart
7 stringPartOk(s, StringValue(_)).
8 stringPartOk(s, InterpExp(exp)) :- typed(s, exp).
9 stringPartOk(s, InterpValue(InterpSimpleExp(simple_exp))) :- { T }
10   typeOfSimpleExp(s, simple_exp) == T.
```

Figure 4.14: WebDSL string typing rules

that the scope graph must contain a single type declaration associated with `"Int"` under the `type` relation. The result of the `resolveType` functional predicate on **line 2** should be a list containing one entry, namely the pair that we declared in Figure 4.11. Other WebDSL constant expressions such as booleans, longs and floats have similar typing rules.

The typing of perhaps the most common constant expression, a string, has an additional condition to be well-typed. Because string interpolation is possible, the constructor of a WebDSL string contains multiple parts that may impose additional constraints. A demonstration of the different interpolated parts is shown in Figure 4.13 and the complete typing rules are shown in Figure 4.14. The parts can be a simple string value which imposes no additional constraints, they can be a complete interpolated expression which requires the expression to be typed, or lastly they can be a "simple" expression which is directly inlineable.

```
1 "Hello world" // value
2 "Hello ~( 1 + 2 )" // exp
3 "Hello ~x.y" // simple exp
```

Figure 4.13: WebDSL string interpolation examples

Now that all the typing rules for constants are implemented, typing rules for unary and binary operators are a step towards more complicated expressions. While it might seem trivial, we might require additional construct functional predicates for determining type compatibility or determining the resulting type of an expression.

4.2.2 Variables

Similar to other imperative languages, WebDSL allows the use of variables to store values. These variables can be defined on multiple levels, such as in the module, within a function or at the top of a page/template definition. Additionally, functions may be embedded in entities, allowing direct access to entity properties as variables without having to prefix it with the `this` keyword.

The basic variable declaration and resolving rules are shown in Figure 4.15. Given a scope `s`, the declaration rule will make a declaration in `s` of variable `x` with associated type `t`.

```
1 declareVar : scope * string * TYPE
2 declareVar(s, x, t) :-
3   !var[x, t] in s,
4   noDuplicateVarDefs(s, x)
5     | error $[A variable named [x] already exists in this scope].
6
7 resolveVar : scope * string -> list((path * (string * TYPE)))
8 resolveVar(s, x) = ps :-
9   query var filter P* /* We will explain the filter later in this section */
10            and { x' :- x' == (x, _) }
11            min $ < P /* We will explain the shadowing later in this section */
12            and true
13            in s |-> ps.
```

Figure 4.15: WebDSL variable declaration and resolving

The implementation of variable typing is similar to the example of let-bindings in Section 4.1.4. One difference between the let-bindings and WebDSL variables is that the introduction of consecutive statements in WebDSL requires a structure that defines declare-before-use semantics, to prevent backwards- or self-references such as shown in Figure 4.16.

```
function f() {
  var a := b;
  var b := b;
}
```

Figure 4.16: WebDSL requires declare-before-use of variables

Figure 4.17 shows how the scope graph is constructed when there are consecutive statements. To catch declare-before-use related errors, a new scope is created for each statement (**line 6 and 7**). When constraints are generated for a constraint (such as on **line 11**), it has access to two scopes. Scope `s` denotes the scope of the current statement. Any scope graph queries will be executed in this scope. Example: the type of this statement is queried starting in scope `s` on **line 12**). Scope `s_decl` denotes the scope of the next statement. Any scope graph declarations will be made this scope. Example: a variable declaration is being made in scope `s_decl` on **line 14**).

Using this tactic, a statement can never access declarations made by itself or by the next statements, it can only access declarations from previous statements.

An example of how this structure influences the building of scope graphs, a visualization of a function, accompanied by the scope graph of its body is shown in Figure 4.18.

Another difference between the let-binding rules from an earlier example and WebDSL variables is the complexity of the shadowing rules. The WebDSL variable shadowing rules which we reverse-engineered from the current compiler and static analysis implementation, state that the same variable identifier may be used multiple times, but never twice in the "environment". Such environments are: module scope, entity properties, functions, templates, etc. If a variable reference has multiple declarations in reach, the closest one according to the shadowing rules will be picked. The regular expression that defines the reachability of variables (left out in **line 9** of Figure 4.15) is shown in Figure 4.19.

The edge label `P` as introduced in Figure 4.17 is the edge label used for linking consecutive statements together. The other edge labels such as complicate this regular expression, and will be will be explained in

```
P*
F*
(
    (EXTEND? (INHERIT EXTEND?)*)
  | (DEF? (IMPORT | IMPORTLIB)?)
)
```

```
1 stmtOk : scope * scope * Statement
2
3 stmtsOk : scope * list(Statement)
4 stmtsOk(_, []).
5 stmtsOk(s, [stmt | tail]) :- {s_decl s_next}
6   new s_decl, s_decl -P-> s,
7   new s_next, s_next -P-> s_decl,
8   stmtOk(s, s_decl, stmt),
9   stmtsOk(s_next, tail).
10
11 stmtOk(s, s_decl, VarDecl(x, sort)) :- { t }
12   t == typeOfSort(s, sort),
13   inequalType(t, UNTYPED()) | error $[Unknown type [sort]] @sort,
14   declareVar(s_decl, x, t),
15   @x.type := t.
```

Figure 4.17: WebDSL statements use different scopes for querying and declaring data from the scope graph

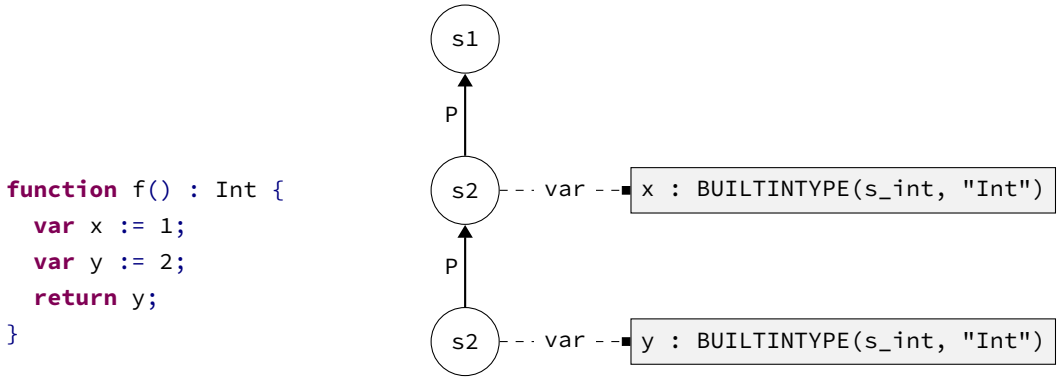


Figure 4.18: Variable declarations example using a separate declaration scopes

more details in later sections when their use is discussed.

Figure 4.19 above defines what data is reachable from any point in the scope graph, but we also want some restrictions of declarations. The same environment such as function body or an entity definition may never declare the same variable twice. To achieve this, **line 4** of Figure 4.15 uses the helper predicate `noDuplicateVarDefs`. The implementation of this predicate is straight-forward and shown in Figure 4.20. The predicate queries the current scope and checks whether all scopes reachable using only `P` edge labels, results in a list containing only one entry.

4.2.3 Type Compatibility

WebDSL has a notion of type compatiblity. For example, the WebDSL superclass of all entities is conveniently called `Entity`. When assigning a value to a variable that requires type `Entity`, passing an instance of a user-defined entity such as `Person` or `Project` also suffices.

```
noDuplicateVarDefs : scope * string
noDuplicateVarDefs(s, x) :-
  query var filter P*
            and { x' :- x' == (x, _) }
            in s |-> [_].
```

Figure 4.20: The same variable identifier may only be declared once in an environment

```
entity Person {}

function f() {
  var e : Entity := Person{}; // all user-defined entities are compatible with Entity
  var d : Date := now(); // now() produces a value of type DateTime
  var p : Person := null; // null is compatible with many types
}
```

Figure 4.21: Examples of type compatibility in WebDSL

```
typeCompatible : TYPE * TYPE
// By default, two types are not compatible
typeCompatible(T1, T2).
// Same type is always compatible
typeCompatible(T, T).
```

Figure 4.22: WebDSL type compatibility predicate and general rules

In this case, type `Person` is compatible with type `Entity`, but not the other way around. Type compatibility is not limited to entities. For instance, all WebDSL date types (`Date`, `Time`, `DateTime`) are compatible with each other. As a last example, null is compatible with many types. The examples given above are shown in Figure 4.21.

To encode the type compatibility as shown in Figure 4.21 in Statix, we need a predicate that tells us, given two types A and B, if type A is compatible with with B. The signature and its general rules are shown in Figure 4.22.

With only the basic rules from Figure 4.22, we have created the equality (`==`) from Statix in predicate form. The advantage of listing it like this, is that we can now add rules to make it fit the WebDSL type system. To continue the example of `null` being compatible with every type, we can add the rules shown in Figure 4.23 to achieve this. An example of how to use our new `typeCompatible` predicate is also given on **line 8** of Figure 4.23.

TO-DO:

- Add (plus) LUB

4.2.4 Boolean Logic in Statix

TO-DO:

- Show sort and constructors
- Examples of where it's necessary

```
typeOfExp(_, Null()) = NULL().
typeCompatible(NULL(), _).

// example of usage:
stmtOk(s, VarDeclInit(x, sort, exp), _) :- { sortType expType }
  sortType == typeOfSort(s, sort),
  expType == typeOfExp(s, exp),
  typeCompatible(expType, sortType)
    | error $[Expression [exp] is not of type [sort], got type [expType]] @exp,
  declareVar(s, x, sort),
  @x.type := t.
```

Figure 4.23: Compatibility of the null expression encoded in Statix

```
signature
  constructors
    // an entity constructor has two subterms:
    // - the entity name
    // - the scope of the entity where all the properties and
    //   functions are declared
    ENTITY : string * scope -> TYPE

rules
  defOk(s_module, EntityNoSuper(entity_name, body)) :- { s_entity }
    // a new scope for the entity is created and linked to the module scope
    // using the `DEF' (for definition) edge label
    new s_entity, s_entity -DEF-> s_module,

    // the new entity is declared as type in the module scope
    declareType(s_module, entity_name, ENTITY(entity_name, s_entity)),

    // finally a helper rule is called that properly handles
    // the entity body definitions (properties, functions, etc.)
    declEntityBody(s_entity, entity_name, body).
```

Figure 4.24: The Statix rules for declaring entities

4.2.5 Entities and Properties

Entities form the basis of the type system and data structure in a WebDSL application. Using Hibernate as an object-relational mapping (ORM) tool, instances of entities can be persisted without explicit communication with a database management system. Entities typically have multiple properties which values are persisted, and functions that can be called and will be executed in the scope of the instantiated entity. Entity properties and entity functions together form the entity body declarations.

In the WebDSL type system, entities are declared in the scope of the module they are defined in. An entity is a type in the WebDSL type system, similar to built-in types such as `String` and `Int`. The Statix code to declare entities is shown in Figure 4.24 and an example of a simple program with entity definition plus its scope graph is shown in Figure 4.25.

The `declareType` and `resolveType` rules as introduced in Figure 4.11 need to be updated to

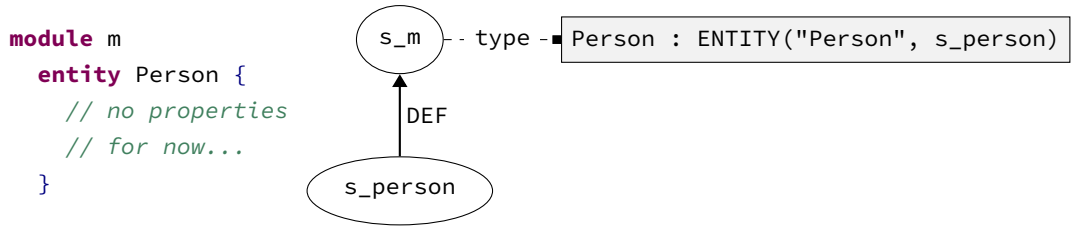


Figure 4.25: An example of entity definition in WebDSL

```
1  declareType : scope * string * TYPE
2  declareType(s, name, t) :-
3    !type[name, t] in s,
4    resolveType(s, name) == [(_, (_, t))]
5      | error $[Type [name] is defined multiple times] @name.
6
7  resolveType : scope * string -> list((path * (string * TYPE)))
8  resolveType(s, name) = typesOf(ts) :-
9    query type filter P* DEF?   // resolving a type may
10                               // optionally follow DEF edge label
11             and { t' :- t' == (name, _) }
12             in s |-> ts.
```

Figure 4.26: `declareType` now shows an error when two types with the same name are declared and `resolveType` may optinally follow a `DEF` edge label

work as intended for resolving and declaring entities. To prevent duplicate entity definitions, the `declareType` rule is extended with one additional rule as shown in Figure 4.26. **Line 4** was added to `declareType`, to make sure when you declare a new type or entity, its name is unique.

In addition to the added constraint to the `declareType` rule, we added an optional `DEF` edge label that may be followed when querying the scope graph for a type (**line 9** of Figure 4.26). The `DEF` (short for defintion) is used to link the scope of top-level elements, such as entities and functions, to the module scope. This can be seen in **line 13** of Figure 4.24.

So far, there has been no reason to query for types inside the entity body because we have always worked with empty entities. In practice, entities are filled with properties and functions. **Line 20** of Figure 4.24 calls the `declEntityBody` predicate, of which the implementation is shown in Figure 4.27.

TO-DO:

- Entity body declarations
- Instantiation

4.2.6 Functions

TO-DO:

- Declaration and resolving
- Prevent duplicates
- Parameters

```
declEntityBody maps declEntityBodyDeclaration(*, *, list(*))
declEntityBodyDeclaration : scope * string * EntityBodyDeclaration

// entity property
declEntityBodyDeclaration(s, ent,
    Property(x, propkind, sort, PropAnnos(annos))) :- { sortType }

  // resolve the type of the property
  sortType == typeOfSort(s, sort),

  // there are some restrictions on property types
  sortType != UNTYPED()
    | error $[Cannot resolve type [sort]] @sort,
  sortType != VOID()
    | error $[Property type 'Void' not allowed] @sort,
  sortType != REF(_)
    | error $[Reference type is not allowed in property] @sort,
  isValidTypeForPropKind(propkind, sort, sortType),

  // declare the property as variable in the entity scope
  declareVar(s, x, sortType),

  // use a helper predicate to check for the uniqueness of
  // the property name
  resolveLocalProperty(s, x) == [_]
    | error $[Property [x] of entity [ent] is defined multiple times] @x.
```

Figure 4.27: Statix rules for declaring the entity body

```
signature
  constructors
    PAGE     : string * list(TYPE) -> TYPE
    TEMPLATE : string * list(TYPE) -> TYPE

  relations
    page     : string * TYPE
    template : string * TYPE
```

Figure 4.28: Statix signature for pages and templates

4.2.7 Pages and Templates

The user-inferface of a WebDSL application is built out of *pages* and *templates*. A page defines a path that is able to be requested by the browser while a template is a reusable component that can be part of a page or nested in other templates.

The name of a page must be unique, while a template can be defined multiple times for different argument types (*overloading*), but never multiple times for the same argument types. Page and template definitions must be declared in the scope graph such that they can be referenced or checked for uniqueness. The statix rules to implement these checks can be found in Figure 4.29.

```
declarePage : scope * string * list(TYPE)
declarePage(s, p, ts) :-
  !page[p, PAGE(p, ts)] in s,
  resolveTemplate(s, p) == []
    | error $[Multiple page/template definitions with name [p]] @p,
  resolvePage(s, p) == [_]
    | error $[Multiple page/template definitions with name [p]] @p.

declareTemplate : scope * string * list(TYPE)
declareTemplate(s, t, ts) :-
  !template[t, TEMPLATE(t, ts)] in s,
  resolvePage(s, t) == []
    | error $[Multiple page/template definitions with name [t]] @t,
  filterTemplateResultsArgs(resolveTemplate(s, t), ts) == [_]
    | error $[Multiple page/template definitions with name [t] and argument types [ts]] @t.
```

Figure 4.29: Statix rules for declaring WebDSL pages and templates

```
pageCallOk : scope * string * list(Exp)
pageCallOk(s, p, args) :- {argTypes ts}
  pageType(s, p) == PAGE(_, ts)
    | error $[There is no page with signature [p]] @p,
  argTypes == typesOfExps(s, args),
  typesCompatible(argTypes, ts)
    | error $[Given argument types not compatible with page definition] @args.

// root page is always accessible from all locations
pageCallOk(_, "root", []).
```

Figure 4.30: Statix rules for type-checking a page reference

TO-DO:

- Show code and resulting scope graph with a page and a template

Type-checking a page reference is easier than the that of a template, since a page definition cannot be overloaded. In order for a page reference to be well-typed, the page must be defined exactly once, and the types of the passed arguments must be compatible with the parameter types of the page. The Statix rules that ticks those boxes is shown in Figure 4.30.

TO-DO:

- Explain resolving templates later
- Ajax templates
- Template elements

4.2.8 Access Control

TO-DO:

- Principal
- References to pages and templates
- Wildcards
- Pointcuts?

4.3 Inheritance

Linking the Scopes

The implementation of inheritance requires the scope of the sub- and super-entity to be connected such that Statix queries can resolve to declarations from the super-entity when necessary. To achieve this, we introduce an edge label `INHERIT` as shown in listing *TO-DO*.

```
name-resolution
  labels
    INHERIT // inherit edge label for subclasses
```

Declarations of sub-entities will generate constraints as shown in listing *TO-DO*.

```
defOk(s_global, Entity(x, super, bodydecs)) :- {s_entity super' s_super}
  resolveEntity(s_global, super) == [(_, (super', ENTITY(s_super)))],
  new s_entity, s_entity -INHERIT-> s_super,
  noCircularInheritance(s_entity),
  declEntity(s_global, s_entity, x, bodydecs),
  @super.ref := super'.
```

First of all, the super-entity refered to in the declaration must refer to an existing entity in the scope graph. Secondly, the new scope belonging to the sub-entity `s_entity` is linked to the scope of the super class `s_super` via an `INHERIT` edge. Finally, some additional constraints are generated to make sure no circular inheritance exists and constraints for the entity body declarations of the sub-entity are generated.

Previously, the resolving of variables was done using the query as shown in listing *TO-DO*

```
resolveVar(s, x) = ps :-
  query var filter P* F* IMPORT*
            and { x' :- x' == (x, _) }
            min $ < P, P < F, F < IMPORT
            and true
            in s |-> ps.
```

The new query in listing *TO-DO* reflects the addition of the `EXTEND` label. The addition of `INHERIT*` in the query filter makes all variables declared in ancestors reachable.

```
resolveVar(s, x) = ps :-
  query var filter P* F* INHERIT* IMPORT*
            and { x' :- x' == (x, _) }
            min $ < P, P < F, F < INHERIT, INHERIT < IMPORT
            and true
            in s |-> ps.
```

Overwriting Functions

Generally, overwriting functions is not allowed in WebDSL. Entity functions are an exception to this such that entity function definitions shadow global function definitions. With the introduction of inheritance there comes another exception, namely that sub-entities are allowed to overwrite function definitions of their ancestors.

Previously, the resolving of entity functions was done using the query as shown in listing *TO-DO* below.

```
1  resolveEntityFunction(s, x) = ps :-
2    query function filter e
3                    and { x' :- x' == (x, _) }
4                    min
5                    in s |-> ps.
```

With the introduction of entity inheritance, the path well-formedness over edge labels should be tweaked such that functions from ancestors are in scope. Changing `filter e` to `filter INHERIT*` accomplishes this. The resulting query is in listing *TO-DO* below.

```
1  resolveEntityFunction(s, x) = ps :-
2    query function filter INHERIT*
3                    and { x' :- x' == (x, _) }
4                    min /* */
5                    in s |-> ps.
```

This query definition works perfectly when sub-entities do not overwrite functions. When a sub-entity does define a function that is already defined in one of its ancestors, resolving the entity function gives two results while we would like only one result, namely the overwritten function defined in the sub-entity. To tackle this challenge, we defined a Statix anonymous shadowing rule combined with a label order. This ensures that when two functions with the same name and argument types exist, only the most specific (i.e. the least inheritance edges) is returned. This is implemented as shown in listing *TO-DO*.

```
1  resolveEntityFunction(s, x) = ps :-
2    query function filter INHERIT*
3                    and { x' :- x' == (x, _) }
4                    /* prioritize local scope over inheritance */
5                    min $ < INHERIT
6                    /* shadow when function name and argument types match */
7                    and {
8                    (f, FUNCTION(args, _, _)),
9                    (f, FUNCTION(args, _, _))
10                   }
11                   in s |-> ps.
```

Entity Type Compatibility

A great perk of having inheritance in a language is writing code for that works for super-entities, and then executing this code with sub-entities. To know if the given entity type is compatible with the required entity type, we require a predicate that defines this compatibility. We have created such a predicate while implementing general type compatibility in subsection *TO-DO*, in the form of `typeCompatibleB : TYPE * TYPE -> BOOL`.

With the addition of entity inheritance, we need to expand this definition. To this end, we added the rules as shown in listing *TO-DO*. Given two entity scopes, the `inherits(s_sub, s_super)` predicate returns true when the query has one result. The query in the `inherits` rule requests all paths from scope `s_sub` to scope `s_super` consisting of only `INHERIT` edges. Such a path exists if and only if the entity belonging to scope `s_sub` inherits the entity belonging to `s_super`.

```
1  typeCompatibleB(ENTITY(s_sub), ENTITY(s_super)) = inherits(s_sub, s_super).
2
3  inherits : scope * scope -> BOOL
4  inherits(s_sub, s_super) = nonEmptyPathScopeList(ps) :-
5    query () filter INHERIT*
6               and { s :- s == s_super }
7               min $ < INHERIT
8               in s_sub |-> ps.
9
10 nonEmptyPathScopeList : list((path * scope)) -> BOOL
11 nonEmptyPathScopeList(_)       = FALSE().
12 nonEmptyPathScopeList([(_,_)]) = TRUE().
```

Built-in Supertype

4.4 Type Extension

4.4.1 Entity Extension

4.4.2 Built-in Type Extension

4.5 Function and Template Overloading

4.6 Module system

4.7 Pre-analyzed built-in library

4.8 Reflection on Statix

- Repeat reasons for using Statix
- What worked out as intended?
- What did not work as intended?
- What are the workarounds?
- Recommendations for improving Statix

Chapter 5

Evaluation

In this chapter we will evaluate the results of the work done in this thesis. First we will assess the correctness of the modernized WebDSL front-end by defining what correctness means in absence of a formal specification and evaluating accordingly. Next, we will share, inspect and reason about the performance of the new parser and static analyses. Lastly, we conclude this chapter by discussing the usability of the modernized implementation in practice.

5.1 Correctness

- Defining correctness in absence of a formal specification
- How correct is the implementation WebDSL
- Explain correctness
- Edge cases

5.2 Performance

- Explain metrics and methods
- Results
- Discuss results

5.3 Usability

- Lack of user-friendliness of the error messages generated by Statix
- Can the WebDSL Statix specification be used as formal specification?
- Maintainability of the Statix and SDF3 codebase

Chapter 6

Related work

6.1 Papers

- Static consistency checking of web applications with WebDSL (Hemel, Groenewegen, Kats, Visser). JSC 2011.
  - Importance of early feedback to developer.
  - Different inconsistencies that can be checked and how they are often checked.
  - Importance of linguistic integration to enable consistency checking at compile time.
  - How WebDSL is designed to enable early reporting of inconsistencies.
- Links: Web Programming Without Tiers (Cooper, Lindley, Wadler, Yallop). FCMO 2006.
  - Introduces the Links programming language.
  - Links generates all tiers of web application: client-side HTML and JavaScript, server-side OCaml and SQL for the database.
  - Links tackles the impedance mismatch problem of different input/output provided and expected by the different tiers.
  - Links is a strict, typed, functional language with all state saved in the client-side.
- Ur/Web: A Simple Model for Programming the Web (Chlipala). POPL 2015.
  - Introduces the Ur/Web programming language.
  - Similar to Links, Ur/Web is a functional programming language that generates code for all tiers of a web application.
  - Ur/Web introduces encapsulation and simple concurrency.
  - Encapsulation: treating key pieces of web applications as private state (?).

6.2 Papers to investigate

- Evolution of the WebDSL runtime
  - About the evolution of a programming language
  - Contains WebDSL details
  - Key difference: This thesis focusses on front-end of the WebDSL language, not the back-end

- Scopes as Types
  - About declaratively specifying static semantics
  - Contains Statix details and case studies
  - Key difference: This thesis contains a larger case study with more language constructs and different (practically motivated) requirements

Maybe?:

- Type errors for the IDE with Xtext and Xsemantics
  - https://www.degruyter.com/document/doi/10.1515/comp-2019-0003/html
  - Implements typechers for two small languages with the Xtext language workbench
  - Describes what to pay attention to when implementing a typechecker (error recovery, useful error messages, etc.)
  - Key difference: This thesis contains a larger case study and is written in Spoofax meta-languages

Chapter 7

Conclusion

7.1 Future work

- Use analysis results for back-end
- Incrementalization in back-end?

Bibliography

Antwerpen, Hendrik van et al. (Oct. 2018). "Scopes as Types". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

Becker, Brett A. et al. (2019). "Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE 2019, Aberdeen, Scotland Uk, July 15-17, 2019*. Ed. by Bruce Scharlau et al. ACM, pp. 177–210. ISBN: 978-1-4503-6895-7. DOI: 10.1145/3344429.3372508. URL: https://doi.org/10.1145/3344429.3372508.

Groenewegen, Danny M., Elmer van Chastelet, and Eelco Visser (2020). "Evolution of the WebDSL Runtime: Reliability Engineering of the WebDSL Web Programming Language". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. '20. Porto, Portugal: Association for Computing Machinery, pp. 77–83. ISBN: 9781450375078. DOI: 10.1145/3397537.3397553. URL: https://doi.org/10.1145/3397537.3397553.

Neron, Pierre et al. (2015). "A Theory of Name Resolution". In: *Programming Languages and Systems*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 205–231. ISBN: 978-3-662-46669-8.

Rafalski, Timothy et al. (2019). "A Randomized Controlled Trial on the Wild Wild West of Scientific Computing with Student Learners". In: *Proceedings of the 2019 ACM Conference on International Computing Education Research, ICER 2019, Toronto, ON, Canada, August 12-14, 2019*. Ed. by Robert McCartney et al. ACM, pp. 239–247. ISBN: 978-1-4503-6185-9. DOI: 10.1145/3291279.3339421. URL: https://doi.org/10.1145/3291279.3339421.

Visser, Eelco (2007). "WebDSL: A Case Study in Domain-Specific Language Engineering". In: *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 5235. Lecture Notes in Computer Science. Braga, Portugal: Springer, pp. 291–373. ISBN: 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3_7. URL: http://dx.doi.org/10.1007/978-3-540-88643-3_7.

Acronyms

AST abstract syntax tree

DSL domain-specific language

Appendix A

A