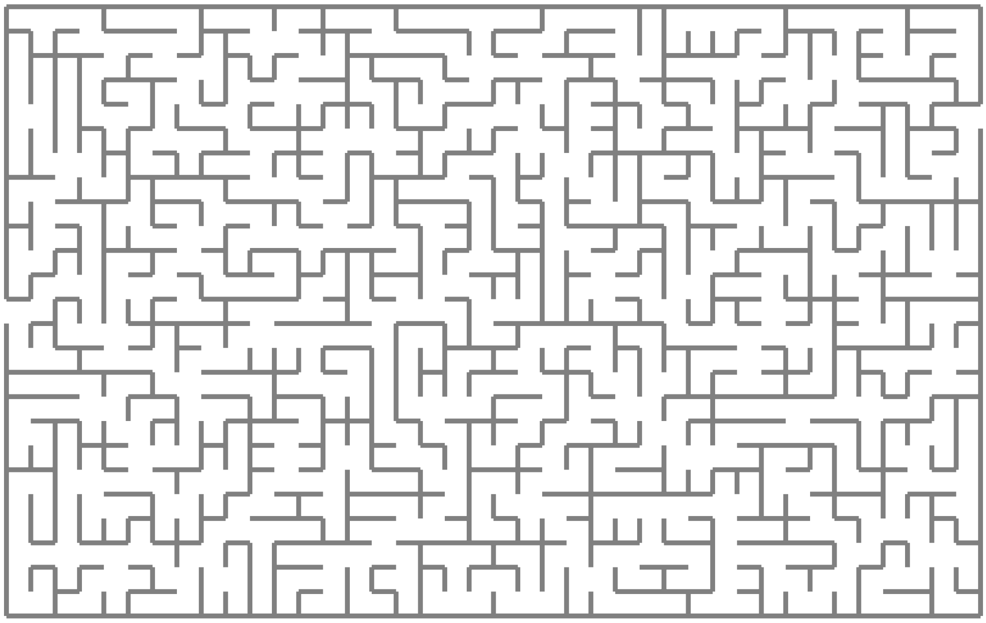
Modernizing the WebDSL Front-End: A Case Study in SDF3 and Statix

Version of October 25, 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: `maxdekrieger@gmail.com`

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. A. Van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. G. H. Cockx, Faculty EEMCS, TU Delft
University Supervisor:	Ir. D. M. Groenewegen, Faculty EEMCS, TU Delft
Thesis Advisor:	$\boxed{\text{Prof. dr. E. Visser, Faculty EEMCS, TU Delft}}$

Preface

Preface here.

Max Machiel de Krieger
Delft, the Netherlands
October 25, 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 popularity of programming languages change and developers tend to have preferences for one language over the other. In addition to preference, the implementation of a language and the tools that come with it can greatly boost the productivity of developers, if done well.

Another key to boost the productivity of software engineers is abstractions. Abstractions allow developers to think in terms closer to the domain rather than the implementation. In other words, the ideal level of abstraction increases the focus on the what, and steers away from the how. In this thesis, we will focus on a *domain-specific language* (DSL). In contrast to a general-purpose language such as Java, C, or Python, a domain-specific language does not intend to provide solutions for problems from all domains, but instead focus on a single domain. This restriction allows for a high level of abstraction in the language itself, in an attempt to boost developer productivity. Examples of popular domain-specific languages are CSS for styling web pages and SQL for efficient database querying.

In this thesis, we are using the domain-specific language *WebDSL* as a large case study for the languages *SDF3* and *Statix*. WebDSL is a domain-specific language for developing web applications, developed and maintained by the Programming Languages research group of the Delft University of Technology.

When inspecting the implementation of a programming language, the process is split up in multiple parts such as parsing, static analysis, code generation and optimization. The parsing, desugaring and static analysis is often called the front-end of a programming language, and this is the part developers face directly. The code generation and code optimization is called the back-end, and is required to make the programming language operational. While the back-end of a programming langauge makes it work, the front-end plays a large role in how developers experience 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).

Because of the language-based approach of WebDSL for encoding domain concepts, many features that would be a library or an external tool in a general purpose language, are linguistically integrated into WebDSL. Examples of such features are fuzzy search and defining the data model. The linguistic integration of these features allows for better consistency checking and more precise error descriptions.

Currently, the WebDSL implementation is composed of multiple definitions in meta-languages supported by the *Spoofax Language Workbench* (Kats and Visser 2010). Spoofax is an environment in which multiple meta-DSLs are used to declaratively specify a programming language. WebDSL is developed in Spoofax. In particular, the WebDSL syntax is defined in SDF2 and the desugaring, typechecking, optimization and code generation is defined in the term transformation language Stratego. In the current Stratego implementation of the WebDSL, the compilation steps are not clearly separated, which poses a threat to the readability and maintainability of the WebDSL language.

Continuous improvement of the Spoofax language workbench has introduced more meta-languages specialized in different parts of the language development chain. In this thesis, we will be modernizing the WebDSL front-end, by using the Spoofax meta-languages SDF3 to specify and disambiguate the syntax from which a parser is generated, and Statix to declare the static semantics from which a typechecker is automatically generated.

1.1 Why WebDSL as case study?

WebDSL is an interesting case study for SDF3 and Statix because of two main reasons. Firstly, WebDSL has a large amount of language features inspired by multiple paradigms of programming languages. As a consequence, the resulting SDF3 and Statix specifications are arguably the largest specifications to date. Accompanied by the large amount of publicly available source code for evaluation purposes, we aim to make observations about the elegancy of the resulting specifications and the scalability of their performance. Since this thesis is a case study, we cannot make general claims about the performance of SDF3 and Statix, but only reveal and analyse results of the WebDSL specification.

Secondly, WebDSL contains language features that have never been modelled in Statix before. Specifically, those features are:

- Extension of built-in types
- Generated functions and classes
- An unconventional module and scoping system

With the implementation of the above features in Statix, we aim to contribute to assessing whether Statix is capable of modelling the static semantics of all reasonable programming languages.

1.2 Contributions

In this thesis, the following contributions are made.

- We present a modernized WebDSL front-end through an implementation of its grammar in SDF3 and its analysis in Statix and document the challenges of this process.
- We assess the coverage of Statix and SDF3 by attempting to model all language features of WebDSL, evaluating the result on existing test suites, and give qualitative feedback on how to further improve the coverage and increase the elegancy of definitions.
- We assess the performance of Statix and SDF3 by benchmarking the new WebDSL front-end with large codebases of existing applications.

1.3 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.

TO-DO:

- Paragraph with strong points of WebDSL

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 conclusе 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, Bach Poulsen, 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, A. Tolmach, 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.

```
signature
  sorts
    Application
    Exp

  constructors
    Application : Exp          -> Application
    True        :                 Exp
    False       :                 Exp
    Int         : string       -> Exp
    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.

```
signature
  sorts
    TYPE

  constructors
    BOOL : TYPE
    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().
```

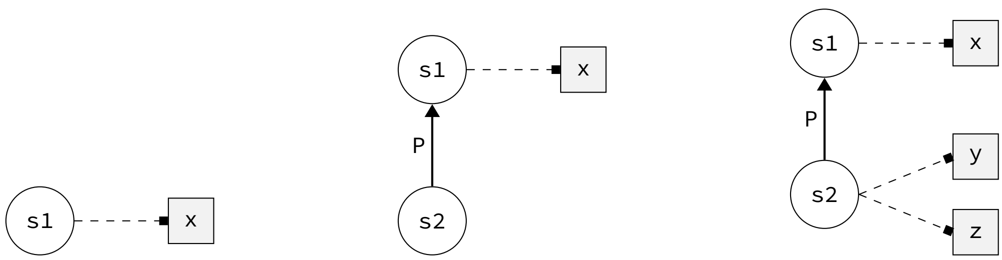
Figure 4.4: Statix predicates and rules for typing booleans, integers and addition

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, A. Tolmach, et al. 2015). Scope graphs are built out of three components: scopes, edges and declarations.



(a) (b) (c)

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.

```
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

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.

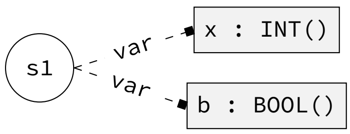


Figure 4.7: A scope graph containing a single scope with two declared variables

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).

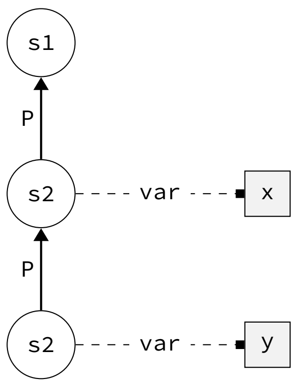
Figure 4.9 shows a possible input and the constructed scope graph after the constraints have been solved.

```
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.

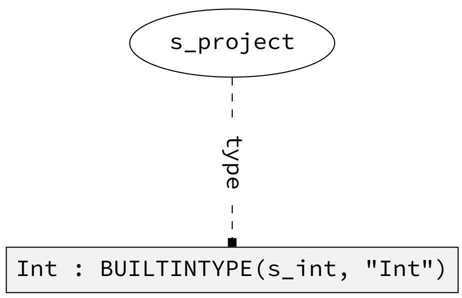
4.2.1 Built-in Types and Constant Expressions

Constant expressions such as strings, integers and booleans form the building blocks of more complication constructs. For reasons explained later (see section Section 4.6.2), 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.
```

```
typeOfExp(s, Const(StringConst(String(str)))) = t :-
  resolveType(s, "String") == [(_, (_, t))],
  stringPartsOk(s, str).

stringPartsOk maps stringPartOk(*, list(*))
stringPartOk : scope * StringPart
stringPartOk(s, StringValue(_)).
stringPartOk(s, InterpExp(exp)) :- typed(s, exp).
stringPartOk(s, InterpValue(InterpSimpleExp(simple_exp))) :- { T }
  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.

```
"Hello world" // value
"Hello ~( 1 + 2 )" // exp
"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* /* The filter will be expanded throughout the chapter */
10              and { x' :- x' == (x, _) }
11              min $ < P
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

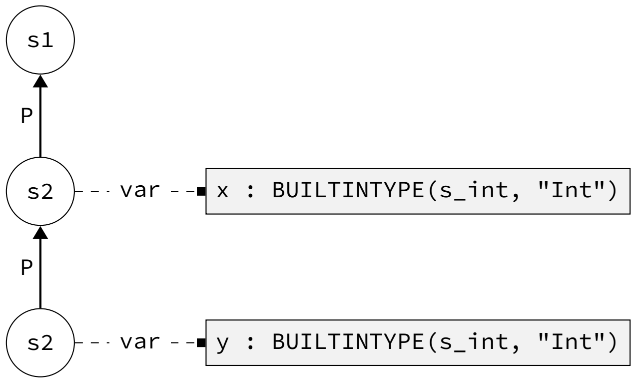
```
P*
F*
(
    (EXTEND? (INHERIT EXTEND?)*)
  | (DEF? (IMPORT | IMPORTLIB)?)
)
```

```
stmtOk : scope * scope * Statement

stmtsOk : scope * list(Statement)
stmtsOk(_, []).
stmtsOk(s, [stmt | tail]) :- {s_decl s_next}
  new s_decl, s_decl -P-> s,
  new s_next, s_next -P-> s_decl,
  stmtOk(s, s_decl, stmt),
  stmtsOk(s_next, tail).

stmtOk(s, s_decl, VarDecl(x, sort)) :- { t }
  t == typeOfSort(s, sort),
  inequalType(t, UNTYPED()) | error $[Unknown type [sort]] @sort,
  declareVar(s_decl, x, t),
  @x.type := t.
```

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

```
function f() : Int {
  var x := 1;
  var y := 2;
  return y;
}
```

(a)



(b)

Figure 4.18: Variable declarations example using a separate declaration scopes

more details in later sections when their use is discussed.

Figure 4.19 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.

Typing of the addition expression

The typing rules for most binary operations such as conjunction is trivial: the resulting value is of boolean type, with the constraint that both operators must be of boolean type. However, the range of values in WebDSL is greater than only natural numbers and booleans. WebDSL supports other numeric types such as `Float`s and `Long`s, as well as string types and multiple subtypes of strings such as `Secret`, `Text` and `WikiText`. The addition operator supports most of these values, and the typing of this operator is not as trivial as boolean conjunction. For example: the addition of two strings results in a string, the addition of a string and an integer results in a string value and the addition of a boolean and a string is not supported.

To calculate the return type of addition, we introduce a functional rule that calculates

```
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

```
lubForAdd : TYPE * TYPE -> TYPE
lubForAdd(T1, T2) = lubForAddNumeric(T1, T2).
lubForAdd(t@BUILTINTYPE("String", _), _) = t.
lubForAdd(_, t@BUILTINTYPE("String", _)) = t.

lubForAddNumeric : TYPE * TYPE -> TYPE
lubForAddNumeric(_, _) = UNTYPED().
lubForAddNumeric(t@BUILTINTYPE("Int", _)     , t) = t.
lubForAddNumeric(t@BUILTINTYPE("Long", _)    , t) = t.
lubForAddNumeric(t@BUILTINTYPE("Float", _)   , t) = t.
lubForAddNumeric(t@NATIVECLASS("Double", _) , t) = t.

// implicit widening from int to long
lubForAddNumeric(BUILTINTYPE("Int", _)       , t@BUILTINTYPE("Long", _))    = t.
lubForAddNumeric(t@BUILTINTYPE("Long", _)    , BUILTINTYPE("Int", _))       = t.

// implicit widening from float to double
lubForAddNumeric(t@NATIVECLASS("Double", _) , BUILTINTYPE("Float", _))     = t.
lubForAddNumeric(BUILTINTYPE("Float", _)     , t@NATIVECLASS("Double", _)) = t.
```

Figure 4.24: Least-upper-bound rules for addition

the least-upper-bound of two types: `lubForAdd : TYPE * TYPE -> TYPE`. The implementation of this rule is given in Figure 4.24. The functional rule `lubForAddNumeric` is reused in other contexts, in particular when generating the constraints for comparison with operators such as greater-than, to check if two types are comparable.

4.2.4 Boolean Logic in Statix

So far, most of the WebDSL's static semantics are expressible in Statix. However, the elegance of the Statix definition is sometimes lost due to code duplication. For example, logical negation and disjunction of predicates are not natively expressible in Statix, and require boilerplate code to function. To tackle this challenge, we introduced a notion of explicit boolean results for predicates that are reusable. The implementation in Statix is shown in Figure 4.25. The figure shows a predicate from before (`typeCompatible : TYPE * TYPE`) now changed to return an explicit result: `typeCompatibleB : TYPE * TYPE -> BOOL`. Additionally, we scope

```
signature
  sorts
    BOOL   // used as return values of functional rules

constructors
  TRUE : BOOL
  FALSE : BOOL

rules
  // return a TRUE() or FALSE() value instead of failing/passing constraint
  typeCompatibleB : TYPE * TYPE -> BOOL

  // scope this explicit results in a predicate to avoid having to work
  // with boolean computation everywhere
  typeCompatible : TYPE * TYPE
  typeCompatible(T1, T2) :- typeCompatibleB(T1, T2) == TRUE().
```

Figure 4.25: Boolean computation results in Statix

this boolean result in a predicate `typeCompatible(T1, T2) :- typeCompatibleB(T1, T2) == TRUE().` such that existing references can be left unchanged.

As hinted before, the explicit return values of functional rules open up new possibilities for expressing constraints. One instance of where this is necessary, is expressing the semantics of an equality check in WebDSL. For the expression `A == B` to type check, the types have to be compatible. The naive implementation would be to define the constraint `typeCompatible(T_A, T_B)`. However, type compatibilty is not symmetrical while the equality check should be: `A == B` $\iff$ `B == A`. An example of type compatibilty not being symmetrical is when dealing with entity inheritance (see Section 4.3.1). To properly define the static semantics for the equality expression in Statix, we need the newly defined boolean computation rules. The result is shown in Figure 4.26.

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 4.27 and an example of a simple program with entity definition plus its scope graph is shown in Figure 4.28.

The `declareType` and `resolveType` rules as introduced in Figure 4.11 need to be updated to 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 4.29. **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 4.29). 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 4.27.

```
or  : BOOL * BOOL
orB : BOOL * BOOL -> BOOL

or(b1, b2) :- orB(b1, b2) == TRUE().

orB(_, _) = FALSE().
orB(TRUE(), _) = TRUE().
orB(FALSE(), TRUE()) = TRUE().

// (e1 == e2)
typeOfExp(s, Eq(e1, e2)) = t :- { T1 T2 }
  t == bool(s),
  typeOfExp(s, e1) == T1,
  typeOfExp(s, e2) == T2,
  or(
    typeCompatibleB(T1, T2),
    typeCompatibleB(T2, T1)
  ).
```

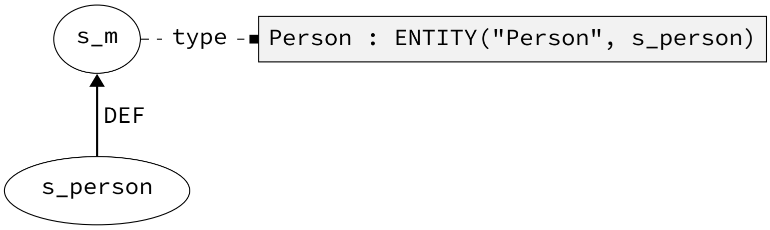
Figure 4.26: Using boolean computation results in Statix for the equality expression

```
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 4.27: The Statix rules for declaring entities

```
module m
  entity Person {
    // no properties
    // for now...
  }
```

(a)



(b)

Figure 4.28: An example of entity definition in WebDSL

```
declareType : scope * string * TYPE
declareType(s, name, t) :-
  !type[name, t] in s,
  resolveType(s, name) == [(_, (_, t))]
    | error $[Type [name] is defined multiple times] @name.

resolveType : scope * string -> list((path * (string * TYPE)))
resolveType(s, name) = typesOf(ts) :-
  query type filter P* DEF?   // resolving a type may
                              // optionally follow DEF edge label
                and { t' :- t' == (name, _) }
                in s |-> ts.
```

Figure 4.29: `declareType` now shows an error when two types with the same name are declared and `resolveType` may optinally follow a `DEF` edge label

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 4.27 calls the `declEntityBody` predicate, of which the implementation is shown in Figure 4.30 and an example of an entity definition with two properties is shown in Figure 4.31.

Entity properties are declared under the variable relation inside the entity scope, such that functions inside entities can reference their own properties without using the `this` prefix. The `this` construct is supported, but not necessary. Declaring properties in this way, allows us to reuse the already existing rules such as those against duplicate definition, without duplicating the code for another relation.

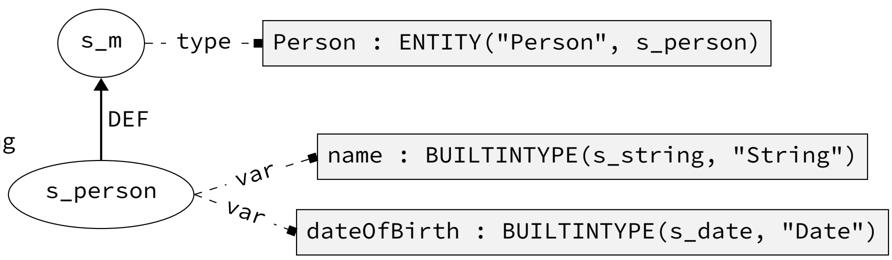
When instantiating an entity, the properties declared in the entity body may be given a value in the instantiation expression. To express this in Statix, an entity instantiation first retrieves the scope of the entity. If the scope cannot be retrieved, it means that the entity is unknown at the position of the expression, so either the entity was never declared or it is not imported correctly. Secondly, all instantiated properties must be declared under the `var` relation of the entity scope. An example of the declaration and scope graph of an entity declaration is shown in Figure 4.31. A part of the Statix rules for instantiating entities is shown in Figure 4.32.

Even though the concepts, rules and approach mentioned in this subsection are present in the Statix specification of WebDSL, we had to simplify the examples and shown Statix rules to hide the extra complexity added concepts such as inheritance, property annotations and type extension. Those concepts will be explained in detail in section Section 4.3 and Section 4.6.

```
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 4.30: Statix rules for declaring the entity body

```
module m
  entity Person {
    name        : String
    dateOfBirth : Date
  }
```



(a) (b)

Figure 4.31: An example of an entity definition with multiple properties

```
typeOfExp(s, ObjectCreation(x, prop_assignments)) = e :-
  definedType(s, x) == e,
  e == ENTITY(_, _),
  propAssignmentsOk(s, e, prop_assignments).

  propAssignmentsOk maps propAssignmentOk(*, *, list(*))
  propAssignmentOk : scope * TYPE * PropAssignment
  propAssignmentOk(s, ent@ENTITY(e, s_ent),
      PropAssignment(x, exp)) :- { propType expType }
    typeOfProperty(s, ent, x) == propType,
    typeOfExp(s, exp) == expType,
    typeCompatible(expType, propType).
```

Figure 4.32: Statix rules for instantiating an entity

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

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

Figure 4.33: Statix signature for pages and templates

4.2.6 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. The statix rules to implement these checks can be found in Figure 4.34 and an example of a module with a page and a template definition is shown in Figure 4.35. In the latter image, the argument type of template `t` is shortened to `String`, instead of its full version `BUILTINTYPE(s_string, "String")`.

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 4.36. The resolving of templates is similar to that of functions and will be explained later in Section 4.4.

The body of templates and pages consist of so called *Template elements*. The simplest template element is simply a text to be printed on the page. Next to plain text, hyperlinks to other pages can be created using the `navigate` element. If we take Figure 4.35 as basis, an example of a valid navigate call would be `navigate p() { "Go to p" }`. To type-check this, the code from Figure 4.36 can be used. Other examples of template elements are forms, nested template calls, and at the top of a template, variables can be initialized, followed by a block of computational statements that get executed when the template is being loaded.

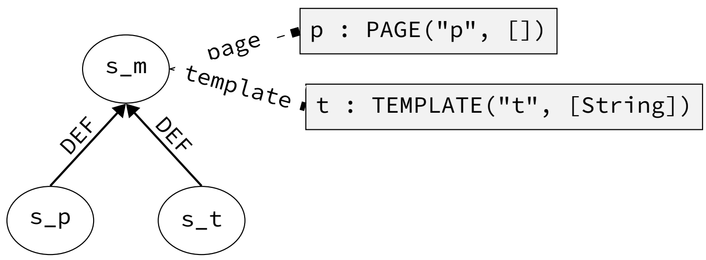
Apart from regular templates, WebDSL also has a notion of Ajax templates. Ajax tem-

```
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 4.34: Statix rules for declaring WebDSL pages and templates

```
module m
  page p {
    "Hello "
    t("World!")
  }

  template t(s : String) {
    ~s
  }
```

(a)



(b)

Figure 4.35: An example of a module with a page `p` and a template `t`

```
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 4.36: Statix rules for type-checking a page reference

plates can be used as building blocks of the user interface, just like regular templates. Additionally, Ajax templates also have a possibility of being replaced on a rendered page without reloading the whole page, for example to refresh the results of a poll on a page. This addition makes Ajax templates useful for more interactive and modern web applications.

Certain action code such as the `replace` and `refresh` statements are only supposed to work on Ajax templates, and not on regular templates. To this end we need to differentiate between them in the scope graph. We have chosen the most trivial way of implementing this, namely adding an additional argument in the type constructor of a template: `TEMPLATE : string * list(TYPE) * BOOL -> TYPE`. The last boolean argument indicates whether the template is an Ajax template or not. When resolving templates we can now resolve only Ajax templates by adding a `TRUE()` to the filter statement of the query.

4.2.7 Functions

In WebDSL, a function is a sequence of statements that perform some sort of computation and can return a value. The type of the return value must be stated in the function header and is part of the signature. The implementation of the declaration and resolving of functions is similar to that of templates, as explained in Section 4.2.6, and therefore will not be repeated here.

An additional characteristic of functions that is similar to templates, is the use of parameters. The parameters with their corresponding types have to be declared statically. The parameters are readable from the function body, but never writable or overridable by a local variable. Additionally, the name of parameters may shadow the name of definitions outside the function such as entity properties or global definitions. To enforce these constraints, we introduce a new edge label `F` for embedding the function scope in their surrounding scope, which is either global or within an entity. Using this new edge label, the shadowing rules can be adjusted to properly check the listed semantics. The result is shown in Figure 4.37 and an example of a WebDSL snippet with the resulting scope graph is shown in Figure 4.38. In the example, parameter `x` of function `f` shadows the globally declared `x`.

Apart from globally declared functions, functions may also be part of an entity. In this case, functions can be called similar to how entity properties are referenced. Lastly, entity functions may have the `static` annotation, which is similar to static class functions in the Java programming language. Static functions may be called without having an instantiated entity.

Possible TO-DO:

- List/explain interesting action code type checking

4.2.8 Access Control

When developing any application that will be used in practice, access control is an important part of the system. It controls which user is allowed to see what data, what actions can be executed. Generally, this is implemented through a log-in system where different user accounts are given different rights. In all popular programming languages, developing a system access control is the responsibility of the developer, either through manual coding or using frameworks and libraries. In WebDSL however, access control is embedded in the language and all pages are protected by default.

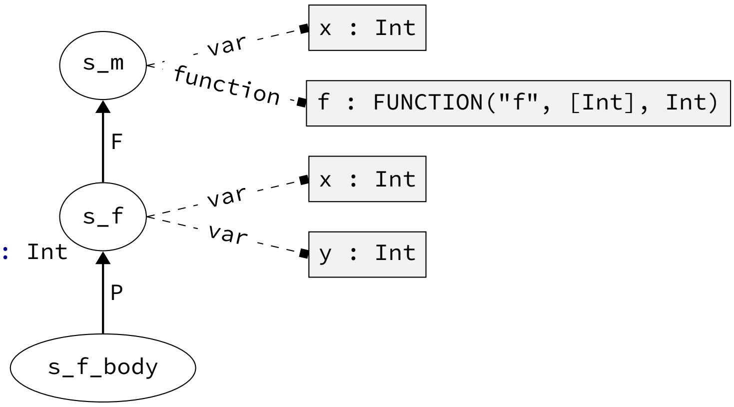
Concretely, the developer is able to declare what entity represents a user in the system, and what data the user needs to show to log in. In the rest of the WebDSL code, the globally available security context is extended with the properties `principal` which references the logged in user, and `loggedIn` which is true if the user has logged in. If the developer has not specified what entity represents a user, the security context is available but does not have these properties. An example of WebDSL code with resulting scope graph is shown in

```
functionOk : scope * Function
functionOk(s_outer,
    Function(name, FormalArgs(args), OptSortSome(returnSort), Block(stmts)))
      :- { argTypes returnType s_function s_body }

  // embed the function scope with edge label F
  new s_function, s_function -F-> s_outer,

  // declare parameters in function
  argTypes == typesOfArgs(s_outer, args),
  declareParameters(s_function, zipArgTypes(args, argTypes)),

  // create the function body and generate constraints
  new s_body, s_body -P-> s_function,
  stmtsOk(s_body, stmts, returnType),

  // declare the function in the outer scope
  returnType == typeOfSort(s_outer, returnSort),
  declFunction(s_outer, name, argTypes, returnType).

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

// a definition is only duplicate in a line of P edges
noDuplicateVarDefs : scope * string
noDuplicateVarDefs(s, x) :-
  query var filter P*
            and { x' :- x' == (x, _) }
            in s |-> [_].
```

Figure 4.37: Statix rules for function parameters and variable shadowing

```
module m
  var x : Int := 1

  function f(x : Int, y : Int) : Int
  {
    return x + y;
  }
```

(a)

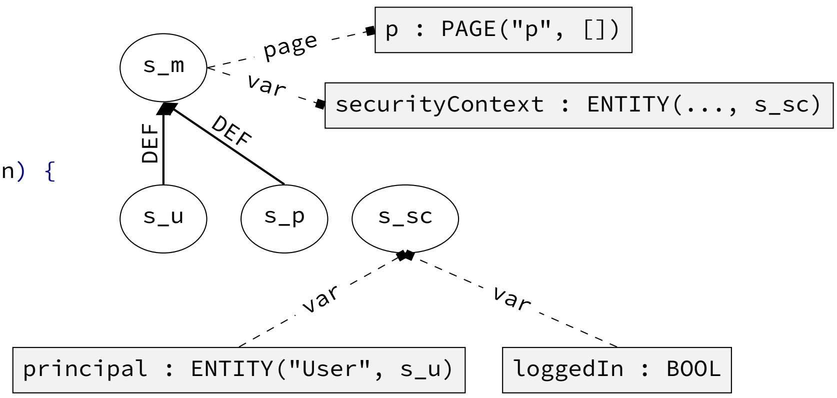


(b)

Figure 4.38: An example of a function with parameters

```
module m
  entity User {
    username : String
    password : Secret
  }

  principal is User
    with credentials
    username, password

  page p {
    if (securityContext.loggedIn) {
      "Welcome!"
    } else {
      "Log in first!"
    }
  }
```

(a)



(b)

Figure 4.39: An example of access control in WebDSL. The entries related to entity `User` are omitted for brevity

Figure 4.39 and the Statix rules used to achieve this are shown in Figure 4.40. In the code for extending the security context with two additional properties, the same mechanics are used as for entity and built-in type extension. An explanation can be found later in Section 4.6.

Possible TO-DO:

- References to pages and templates
- Wildcards
- Pointcuts?

```
principalDefOk : scope * string * list(string)
principalDefOk(s, ent, properties) :-
  { s_ent entityName credentialTypes t }
  definedType(s, ent) == t@ENTITY(entityName, s_ent),
  principalPropertyTypes(s_ent, properties, ent) == credentialTypes,
  compatibleCredentialTypes(properties, credentialTypes),
  declSecurityContext(s, t, credentialTypes).

compatibleCredentialTypes maps compatibleCredentialType(list(*), list(*))
compatibleCredentialType : string * TYPE
compatibleCredentialType(x, s) :-
  isStringCompatibleType(s).

declSecurityContext : scope * TYPE * list(TYPE)
declSecurityContext(s, principalType, credentialTypes) :-
  { s_extend_security_context }
  new s_extend_security_context,
  declProperty(s_extend_security_context, "securityContext"
    , "principal", principalType),
  declProperty(s_extend_security_context, "securityContext"
    , "loggedIn", bool(s)),
  declareExtendScope(s, "securityContext", s_extend_security_context),
  extendScopes(resolveExtendScope(s, "securityContext")
    , s_extend_security_context).
```

Figure 4.40: Statix rules for declaring the access control principal

4.3 Advanced Entity Features

4.3.1 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 Figure 4.41.

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.

The variable resolving query as listed in Figure 4.43 reflects the addition of the `INHERIT` label. The addition of `INHERIT*` in the query filter makes all variables declared in ancestors reachable, but the shadowing rule as declared after the `min` keyword ensures correct shadowing behaviour, namely that local variables are preferred over variables defined in ancestors.

Overwriting Functions

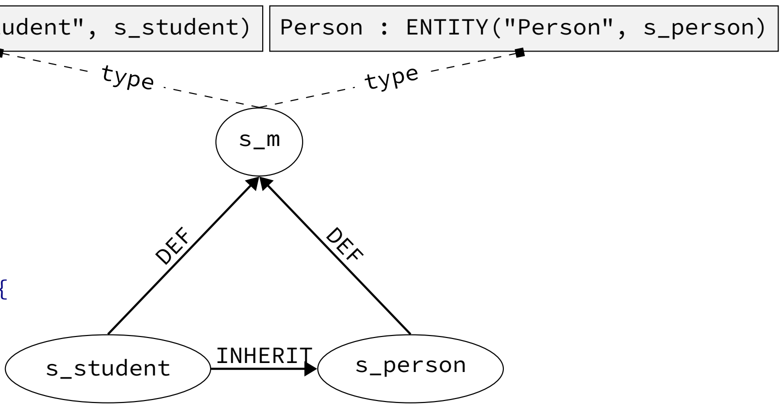
Generally, defining two functions with the same name and same argument types is not allowed in WebDSL. Entity functions are an exception to this such that entity function definitions shadow global function definitions. With the introduction of inheritance there comes

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

rules
  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'.
```

Figure 4.41: Entity inheritance Statix rules

```
module m
  entity Person {
    // no properties
  }

  entity Student : Person {
    // no properties
  }
```

(a)



(b)

Figure 4.42: An example of entity definition in WebDSL

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

Figure 4.43: The query that specifies what variables can be resolved, updated to reflect entity inheritance

```
// previously (local entity scope only)
resolveEntityFunction(s, x) = ps :-
  query function filter e
                 and { x' :- x' == (x, _) }
                 min
                 in s |-> ps.

// new (allow resolving to ancestors)
resolveEntityFunction(s, x) = ps :-
  query function filter INHERIT*
                 and { x' :- x' == (x, _) }
                 min /* */
                 in s |-> ps.
```

Figure 4.44: Statix rules for allowing entity function calls to resolve to definitions in their ancestors

```
resolveEntityFunction(s, x) = ps :-
  query function filter INHERIT*
                 and { x' :- x' == (x, _) }
                 /* prioritize local scope over inheritance */
                 min $ < INHERIT
                 /* shadow when function name and argument types match */
                 and {
                   (f, FUNCTION(args, _, _)),
                   (f, FUNCTION(args, _, _))
                 }
                  in s |-> ps.
```

Figure 4.45: Statix rules for resolving entity functions that allow overriding

another exception, namely that sub-entities are allowed to override function definitions of their ancestors.

Previously, the resolving of entity functions was done using a query that resolves within the entity scope only. With the introduction of entity inheritance, the path well-formedness over edge labels was changed such that functions from ancestors are also in scope. Changing `filter e` to `filter INHERIT*` accomplishes this. Both the previous and resulting queries are shown in Figure 4.44.

This query definition is adequate when sub-entities do not override 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 the desired outcome is only one result, namely the overridden 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 Figure 4.45.

Entity Type Compatibility

A perk of having the notion of inheritance in the WebDSL language, is that it allows for better abstraction and less code duplication. An example of this is a function definition,

```
typeCompatibleB(ENTITY(s_sub), ENTITY(s_super)) = inherits(s_sub, s_super).

inherits : scope * scope -> BOOL
inherits(s_sub, s_super) = nonEmptyPathScopeList(ps) :-
  query () filter INHERIT*
           and { s :- s == s_super }
           min $ < INHERIT
           in s_sub |-> ps.

nonEmptyPathScopeList : list((path * scope)) -> BOOL
nonEmptyPathScopeList(_)        = FALSE().
nonEmptyPathScopeList([(_,_)]) = TRUE().
```

Figure 4.46: Statix rules for entity type compatibility that support inheritance

where the argument type is an entity. This function can be called with an argument of the entity type, or one of its sub-entities. To know if the given type is compatible with the required type, we require a predicate that defines this compatibility. We have created such a predicate while implementing general type compatibility in Section 4.2.3, 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 Figure 4.46. 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`. An example of a scope graph with entity inheritance is shown in Figure 4.42.

4.3.2 Property Annotations

So far, the Statix specification can validate entities, their properties and their functions. Since the goal is to never manually touch the database specification, we would like to entity properties to be more expressive by for example specifiying default values, or put a constraint on the possible values of a property. WebDSL uses *property annotations* for this. Figure 4.47 shows an WebDSL code of an entity with properties that have annotations.

Many property annotations do not influence the scope graph. An example of this is the `default = <exp>` annotation, where Statix only needs to check whether the given expression is compatible with the property type. For the `length = <exp>` annotation, the same holds, except that the expression must now have type `Int`.

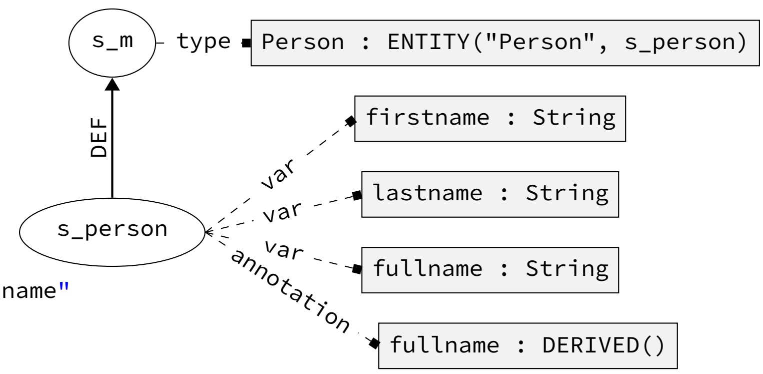
An interesting property to point out is the derived property, as shown in for property `fullname` of the `Teacher` entity in Figure 4.47. While this is not strictly an annotation, it does change something in the scope graph. A derived value can be calculated from other properties of the entity and does not have to be stored. It's value can also not be changed directly. The latter property is something we need to store in the scope graph, such that we can give an error when the developers attempts to assign a value directly to a derived property. For this, a new relation is introduced in Statix, which allows us to declare annotations on properties in the scope of an entity. An example of this is shown in Figure 4.48. When assigning a variable, the left-hand side of the assignment can now be checked for mutability. The implementation of this checks whether the entity property that is referenced on the left-hand side has the `DERIVED()` annotation. To prevent code duplication, we chose to re-use the `DERIVED()` property for function parameters, which can only be referenced but never changed in the function body.

```
module m
  entity Course {
    key : String (default="change-me")
    ects : Float (validate(ects >= 0, "ECTS may not be lower than 0"))
    teacher : Teacher (not null)
  }

  entity Teacher {
    firstname : String
    lastname : String (length = 255)
    courses : [Course] (inverse=teacher)

    temporaryNumber : Int (transient)
    fullname : String := getFullName()

    function getFullName() : String {
      return "~firstname + ~lastname";
    }
  }
```

Figure 4.47: Examples of entity property annotations

```
module m
  entity Person {
    firstname : String
    lastname  : String
    fullname  : String :=
      "~firstname + ~lastname"
  }
```

(a)



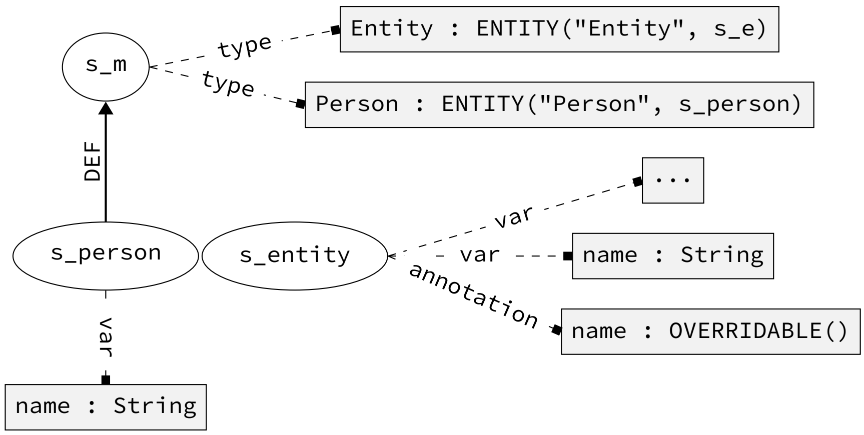
(b)

Figure 4.48: An example of a derived property in an entity

Another interesting annotation to mention, is the `inverse = <var>` annotation as shown for the `courses` property of the `Teacher` entity in Figure 4.47. The inverse annotation is introduced to prevent data duplication in the database. To continue with the example of `Teacher` and `Course` of Figure 4.47, the `Course` table saves the corresponding teacher, and when a teacher is fetched from the database, the `courses` property is instantiated according to the data in the `Course` table. When specifying `inverse=teacher`, the Statix specification has to validate that the entity mentioned in the property type (`Course` in this case) has the `teacher` property, and that the type of that `teacher` property is equal to the type of the entity that the inverse annotation was declared in (`Teacher` in this example). To prevent a situation where none of the two entities is responsible for saving the data, a double inverse annotation is not allowed. To enforce this, another constraint has to be added, namely that the `teacher` property of `Course` does not have an inverse annotation. The only way to reliably check this is to save the annotation `INVERSE()` in the scope graph.

```
module m
  entity Person {
    name : String
  }
```

(a)



(b)

Figure 4.49: An example of overriding the `name` property

4.3.3 Entity Hierarchy and the Name Property

Similar to `Object` in the Java programming language, WebDSL also has a root of the entity hierarchy, namely `Entity`. If a defined entity does not explicitly inherit from another entity, it will automatically inherit from `Entity`. This built-in superclass is convenient to store properties that all entites will have out of the box, such as the property `id` of type `UUID` and the property `created` of type `DateTime`. User-defined entities are not allowed redefine such properties, but they may to edit the values. An exception to this rule is the property `name` of type `String` that all entities have by default, but may be overridden once by sub-entities.

To achieve the overridability, we can re-use the property annotations in the scope graph as explained in previous section. The `name` property of the built-in entity `Entity` gets a `OVERRIDABLE()` annotation declared in the scope of `Entity`, and when attempting to catch duplicate property definitions, we must discard properties from parents that have the `OVERRIDABLE()` property. An example of the overriding is shown in Figure 4.49 and the Statix code to discard overridable properties is shown in Figure 4.50. In the Statix rules, there are now two predicates to prevent duplicates: one for within the entity, and another one to check inherited properties. The difference is rules allows for better error messages, but most importantly allows us to only discard properties with the `OVERRIDABLE()` annotation from inherited properties. At the point of writing this thesis, the built-in name property is the only use case for the `OVERRIDABLE()` annotation and it is not possible to mark a property as overridable by code.

4.4 Function and Template Overloading

Function overloading is the practice of defining multiple functions with the same name, that differ in argument types or in the amount of arguments. Before running the WebDSL program, the compiler determines what instance of the function will be run, based on the types of the function call arguments. WebDSL supports overloading for both functions and templates, and their implementation in the static analysis is the same. The concepts and solutions explained in this section are therefore applicable for both. An example of template overloading is shown in Figure 4.51. In the rest of this section we will talk about functions but the concept is exactly the same for templates.

Overloading complicates the static analysis in two ways. The first and easiest to implement is that functions may now be defined multiple times with the same name, as long as the amount of arguments and argument types do not exactly match. The Statix rules that achieve this are shown in Figure 4.52. The essence of the rules is that all functions with the relevant name are retrieved, and the declarations with argument types exactly matching

```
declProperties maps declProperty(*, *, list(*), list(*))
declProperty : scope * string * string * TYPE
declProperty(s, ent, x, sortType) :-
  validPropertyName(x),
  declareVar(s, x, sortType),
  resolveLocalProperty(s, x) == [_]
    | error $[Property [x] of entity [ent] is defined multiple times] @x,
  noDuplicateVarDefsInSuper(s, x)
    | error $[Cannot override existing entity property [x]] @x.

noDuplicateVarDefsInSuper : scope * string
noDuplicateVarDefsInSuper(s_sub, x) :- { xs nonOverridable }
  resolveProperty(s_sub, x) == xs,
  withoutAnnotation(xs, OVERRIDABLE()) == nonOverridable,
  amountNonOverridableOk(nonOverridable).

amountNonOverridableOk : list((path * (string * TYPE)))
amountNonOverridableOk(_) :- false.
amountNonOverridableOk([]).
amountNonOverridableOk([_]).
```

Figure 4.50: Statix rules for overridable entity properties

```
module m

  entity Animal {
    name : String
  }
  entity Cat : Animal {
    breed : String
  }

  template description(a : Animal) {
    "~a.name"
  }
  template description(c : Cat) {
    "~c.name (Breed: ~c.breed)"
  }
```

Figure 4.51: An example of defining overloaded templates in WebDSL

```
// predicate that defines when there are overlapping function signatures
noDuplicateFunDefs : scope * string * list(TYPE)
noDuplicateFunDefs(s, f, ts) :- { ps }
  resolveFunction(s, f) == ps,
  amountOfFunDeclsWithArgs(ps, ts, 0) == 1.

// helper function for noDuplicateFunDefs that counts the amount
// of functions with a given name and argument types
amountOfFunDeclsWithArgs : list((path * (string * TYPE)))
  * list(TYPE) * int -> int
amountOfFunDeclsWithArgs([], _, n) = n.
amountOfFunDeclsWithArgs([(_, (_, FUNCTION(_, types, _, _))) | tail], types, n)
  = amountOfFunDeclsWithArgs(tail, types, i) :- i #= n + 1.
amountOfFunDeclsWithArgs([_ | tail], types, n)
  = amountOfFunDeclsWithArgs(tail, types, n).
```

Figure 4.52: Statix rules for allowing overloaded function definitions

with the relevant types are counted. The resulting number should be 1, namely the newly declared function.

Now that the static analysis allows for overloaded functions and templates to be defined, the code that typechecks function calls and template calls should be updated to reflect the new changes. The semantics of resolving the correct overloaded function or template are listed below. A practical example of how these rules work is shown in Figure 4.53.

1. Retrieve all function signatures with the matching name from the scope graph
2. Filter the result to end up with function signatures with matching arity (amount of arguments) and compatible argument types.
3. If the filtered result is exactly one signature: this is the function that will be called
4. If the filtered result is more than one signature: choose the signature with the "most specific" argument types:
   - If there are exactly matching types, always choose this one.
   - Otherwise; count the amount of `INHERIT` edges that have to be taken from the given expression types to the function argument types, and choose the signature with the least total edges taken.

The implementation of resolving the correctly overloaded function according to the listed semantics is shown in Figure 4.54. The essence of the semantics is encoded in the Statix rules, but the brevity and elegancy is lost due to many helper predicates being required to transform the data into the correct forms to perform the queries that calculate the inheritance edges, and filter the function signatures accordingly. Note that the `typeOfFunctionCall` predicate is not specific to functions, because the signature requires a string and a list of expressions, which causes the predicate to be re-usable for resolving template calls.

Possible TO-DO:

- Change implementation to throw an error when an overloaded function is not "strictly better" than another (e.g. another one has a more specific argument type for at least one of the arguments).

```
module m

  entity Animal {
    name : String
  }
  entity Cat : Animal {
    breed : String
  }

  template description(a : Animal) {
    "~a.name"
  }
  template description(c : Cat) {
    "~c.name (Breed: ~c.breed)"
  }

  page p {
    var a := Animal{ name := "Alice" }
    var c := Cat{ name := "Charlie", breed := "Sphynx" }

    description(a) // will output "Alice"
    description(c) // wil output "Charlie (Breed: Sphynx)"
  }
```

Figure 4.53: An example of referencing overloaded templates in WebDSL

4.5 Placeholders, Actions and Submitting Forms

Forms are the basis of most information systems on the web. WebDSL has linguistically integrated forms through inputs and actions. The static analysis of referencing actions and placeholders is unlike variable referencing, due to their scoping semantics. Actions abd placeholders are defined in pages or templates and may be referred to in the rest of the page or function body. Unlike variables, actions and placeholders do not follow the declare-before-use principle (explained in Section 4.2.2). Actions and placeholders may be defined anywhere in the body, and referenced from anywhere in the body, but there is a twist. Inside the action body, the statements and expressions may reference variables defined in the template or page body, and thus the scope of the action must be linked to the scope of the template of page in some way.

Summarized, actions and placeholders cannot be declared in the scope they are defined in, because that scope follows a declare-before-use regime, but the body of an action must have access to the scope it was declared in. As a solution to this challenge, the Statix rules for declaring a page or template were altered to not only create a scope for its body, but create an additional scope where placeholders and actions will be declared. This scope is passed along such that variables are queryable using the regular function body scope, and querying placeholders and actions is available using the additional scope. An example of an action and its representation in the scope graph is shown in Figure 4.55.

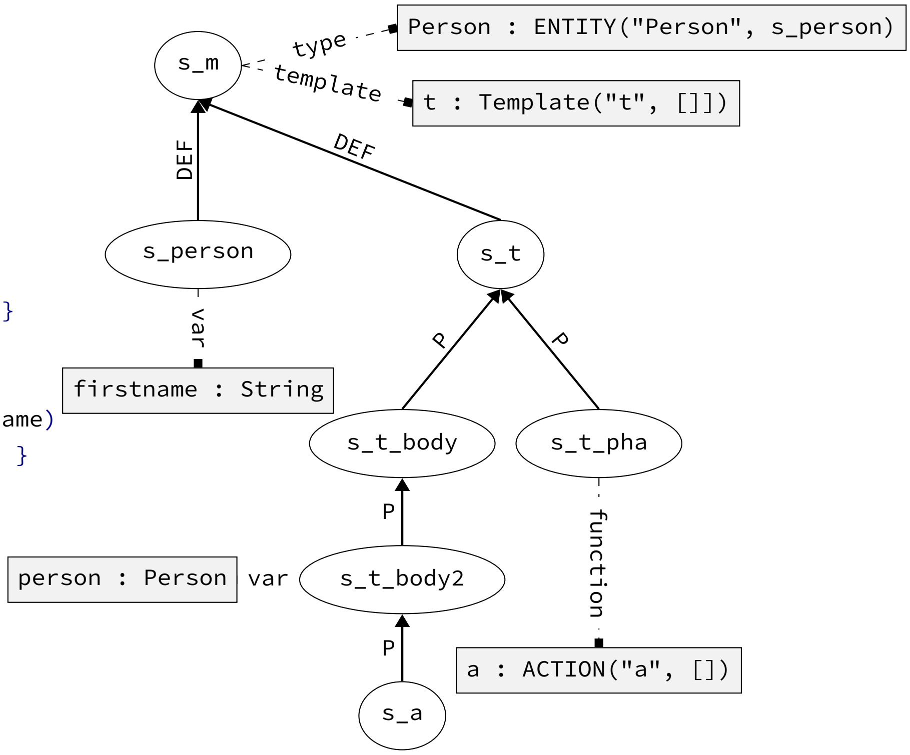
The updated Statix rules for typechecking a page or template definition, and defining actions in shown in Figure 4.56. The Statix rules contain a separate predicate `templateActionOk` where the last boolean parameter indicates whether the action should be declared or not, and this is passed to `optionallyDeclareTemplate`. The declaration is optional because it is possible to create a form with a submit button that defines an action inline; an anonymous

```
typeOfFunctionCall : scope * string * list(Exp)
  * list((path * (string * TYPE))) -> TYPE
typeOfFunctionCall(s, f, args, funSigs) = t :- { argTypes f' result }
  argTypes == typesOfExps(s, args),
  result == mostSpecificSigs(
    argTypes
    , typeCompatibleSigs(funSigs, argTypes
  )),
  [(f', FUNCTION(_, _, t, _))] == result
    | error $[Cannot resolve function [f] with compatible argument types] @f.

// function that gets all functions/templates with matching name
// and compatible argument types
typeCompatibleSigs : list((path * (string * TYPE))) * list(TYPE)
  -> list((string * TYPE))
/* implementation not shown for brevity */

// function that prunes the list of compatible signatures
// to a list of most specific signatures
mostSpecificSigs : list(TYPE) * list((string * TYPE)) -> list((string * TYPE))
// In case no functions are compatible, return empty list
mostSpecificSigs(args, []) = [].
// In case of only one compatible signature, return that
mostSpecificSigs(args, fs@[_]) = fs.
mostSpecificSigs(args, sigs) = mostSpecificSigs_helper(args, sigs,
  matchingSigs(stripRefTypes(args), sigs)).

// helper function for mostSpecificFunSigs that returns
// the exactly matching signatures if they exists,
// else return the most specific (least inheritance) signatures
mostSpecificSigs_helper : list(TYPE) * list((string * TYPE))
  * list((string * TYPE)) -> list((string * TYPE))
mostSpecificSigs_helper(args, sigs, matching) = matching.
mostSpecificSigs_helper(args, sigs, [])       =
  filterLeastInheritanceAmount(
    minOfList(inheritanceAmounts)
    , zipInheritanceAmountWithSig(inheritanceAmounts, sigs)) :-
  inheritanceAmounts == inheritanceAmounts(args, sigs).

// at least ten more helper predicates are not shown that calculate
// the amount of inheritance edges and perform the filtering
```

Figure 4.54: Statix rules for resolving overloaded function calls

```
module m

entity Person {
  firstname : String
}

template t {
  var person := Person{}

  form {
    input(person.firstname)
    submit a() { "Save" }
  }

  action a() {
    person.save();
    return p();
  }
}
```

(a)



(b)

Figure 4.55: An example of defining and referencing actions

action in some sense. The anonymous inline action definition can now be type-checked by re-using the `templateActionOk` predicate.

The Statix rules which implement the placeholders and actions code behave correctly because of Statix' scheduling algorithm as first described by in the original Statix paper (Antwerpen, Bach Poulsen, et al. 2018) and further characterized by Rouvoet et al. (Rouvoet et al. 2020).

4.6 Type Extension

4.6.1 Entity Extension

4.6.2 Built-in Type Extension

4.7 Module system

4.8 Pre-analyzed built-in library

4.9 String Manipulation in Statix

```
defineTemplateOk(s, DefineTemplate(mods, t
    , FormalArgs(args), _, elements))
  :- {fargTypes s_template s_pha s_body}
  new s_template, s_template -DEF-> s,
  argTypes == typesOfArgs(s, args),
  declareParameters(s_template, zipArgTypes(fargs, argTypes)),
  new s_pha, s_pha -P-> s_template,    // scope for placeholders and actions
  new s_body, s_body -P-> s_pha,       // scope for template body
  declareTemplate(s, t, argTypes, isAjaxTemplate(mods)),
  overriddenElementExists(s, Template(), t, isAjaxTemplate(mods)),
  templateElementsOk(s_body, s_pha, elements).

templateElementOk(s, _, s_pha
  , Action2TemplateElement(Action(_, a, FormalArgs(args), Block(stmts))))
  :- templateActionOk(s, s_pha, a, args, stmts, TRUE()).

templateActionOk : scope * scope * string
  * list(FormalArg) * list(Statement) * BOOL
templateActionOk(s, s_pha, a, args, stmts, declare)
  :- {s_fun s_fun_body argTypes}
  new s_fun, s_fun -P-> s,
  argTypes == typesOfArgs(s, args),
  declareParameters(s_fun, zipArgTypes(args, argTypes)),
  new s_fun_body, s_fun_body -P-> s_fun,
  optionallyDeclareAction(s_pha, a, args, argTypes, declare),
  stmtsOk(s_fun_body, stmts, PAGE(_, _)).

optionallyDeclareAction : scope * string * list(FormalArg) * list(TYPE) * BOOL
optionallyDeclareAction(_, _, _, _, FALSE()).
optionallyDeclareAction(s, a, args, ts, TRUE())
  :- declareAction(s, a, args, ts).
```

Figure 4.56: Statix rules for declaring actions

Chapter 5

Evaluation

In this chapter we evaluate the newly introduced SDF3 and Statix specifications for WebDSL. The new specifications have two concrete use-cases, namely serving as a case study for SDF3 and Statix, and being used on a daily basis by WebDSL developers. For both purposes it is useful to gather information about how the specifications behave in various situations. As a result of the case study, we want to show strengths and weaknesses of SDF3 and Statix based on information from the specifications, and for the WebDSL developers we would like to decide whether the new specifications are ready to be used in practice.

For both specifications, we will evaluate their correctness and performance on existing test suites, as well as WebDSL code that is used in practice. Then, we conclude this chapter by discussing the usability of the modernized implementation in practice.

5.1 Evaluating the WebDSL SDF3 Specification

Evaluating the SDF3 specification of the WebDSL grammar is done in two parts: its correctness and its performance in terms of generated parse tables and their run time. In this section we will use the current implementation of the WebDSL grammar in SDF2 as the reference grammar for correctness and performance.

5.1.1 Correctness

In this thesis we do not formally prove the correctness of the new grammar. Instead, we parse test suites that are intended for the current SDF2 specification and observe whether the files are parsed correctly and construct an AST without ambiguities. Additionally, we parse open source WebDSL applications that are used in practice and again observe whether the files parse correctly.

The test suite consists of 231 WebDSL snippets, ranging from single expressions to complete functioning applications. To re-use this test suite for the SDF3 specification, we converted the snippets into SPT tests. The existing syntax test suite is not a complete test suite of all syntax constructs but mostly contain syntax fragments which were problematic in the past to serve as a regression test suite. For the sake of completion, we extended the SPT test suite, leading to a new total of 1118 SPT tests, where the newly added test have an expected AST result, instead of only expecting the snippets to parse correctly. The result of running the WebDSL SDF3 specification on the syntax test suite is shown in Table 5.1.

In addition to the test suite, we used two open-source WebDSL applications for verifying that the new parser generated from the SDF3 specification does not suddenly fail or see ambiguities in existing applications:

	Parsed succesfully	Parsed with ambiguities	Failed to parse
Original test suite	231	0	0
Extension	887	0	0
Total	1118	0	0

Table 5.1: Results of parsing the syntax test suite with the WebDSL SDF3 specification.

- **Reposearch[1]:** A source code search engine that helps to find implementation details and example usages. Reposearch consists of 16 main files, 19 library files and 1 standard library file, totalling at 8,722 lines of code spread over 36 files.

- **YellowGrass[2]:** A tag based issue tracker similar to GitHub Issues, complete with access control and used daily by WebDSL developers. YellowGrass consists of 54 WebDSL files plus 20 WebDSL library files and 1 standard library file, coming to a total of 12,898 lines of code spread over 75 files.

Using the parser generated from the WebDSL SDF3 specification, all files of both projects parsed succesfully without ambiguities.

One thing to note in discussing correctness of the WebDSL SDF3 completeness is that, while the results are promising, the SDF3 specification has introduced many new sorts and constructors for disambiguation purposes, and to comply with the Statix Signature Generator expectations. The effect of this change is that we cannot automatically guarantee correctness of the disambiguation, because the resulting AST from the SDF3 definition is different compared to the SDF2 definition. Instead, we manually inspected the ASTs of handpicked snippets and no incorrect results were found.

5.1.2 Performance

The performance of a parser of a programming language is essential due to the rest of the compilation chain depending on its output. A requirement to use the parser generated by the new SDF3 specification in practice, is that its run time should not increase substantially.

Grammar specifications in SDF2 and SDF3 are not interpreted directly. Both formalisms generate a parse table, which is interpreted by the parser implementation JSGLR[3]. JSGLR is an implementation of SGLR parsing in Java, used within the Spoofax Language Workbench. Because of this architecture, it is insightful to inspect the generated parse tables and highlight the differences, as well as comparing the run times of both parsers on the test suite and existing applications.

Parse table from	States	Gotos	Max gotos per state	Actions	Max actions per state
SDF2	10,449	179,454	510	62,127	107
SDF3	12,866	244,688	821	525,728	2,491

Table 5.2: Data about the size of the parse tables generated from the WebDSL SDF2 and SDF3 grammar specifications.

The parse table generated from the SDF3 specification has more states, gotos and actions than the parse table from the SDF2 specification. Even though the described grammar did not change, it is implemented differently, leading to the increase in parse table size. Given

[1]https://codefinder.org/, Source code: https://github.com/webdsl/reposearch/
[2]https://yellowgrass.org/, Source code: https://github.com/webdsl/yellowgrass/
[3]https://github.com/metaborg/jsglr

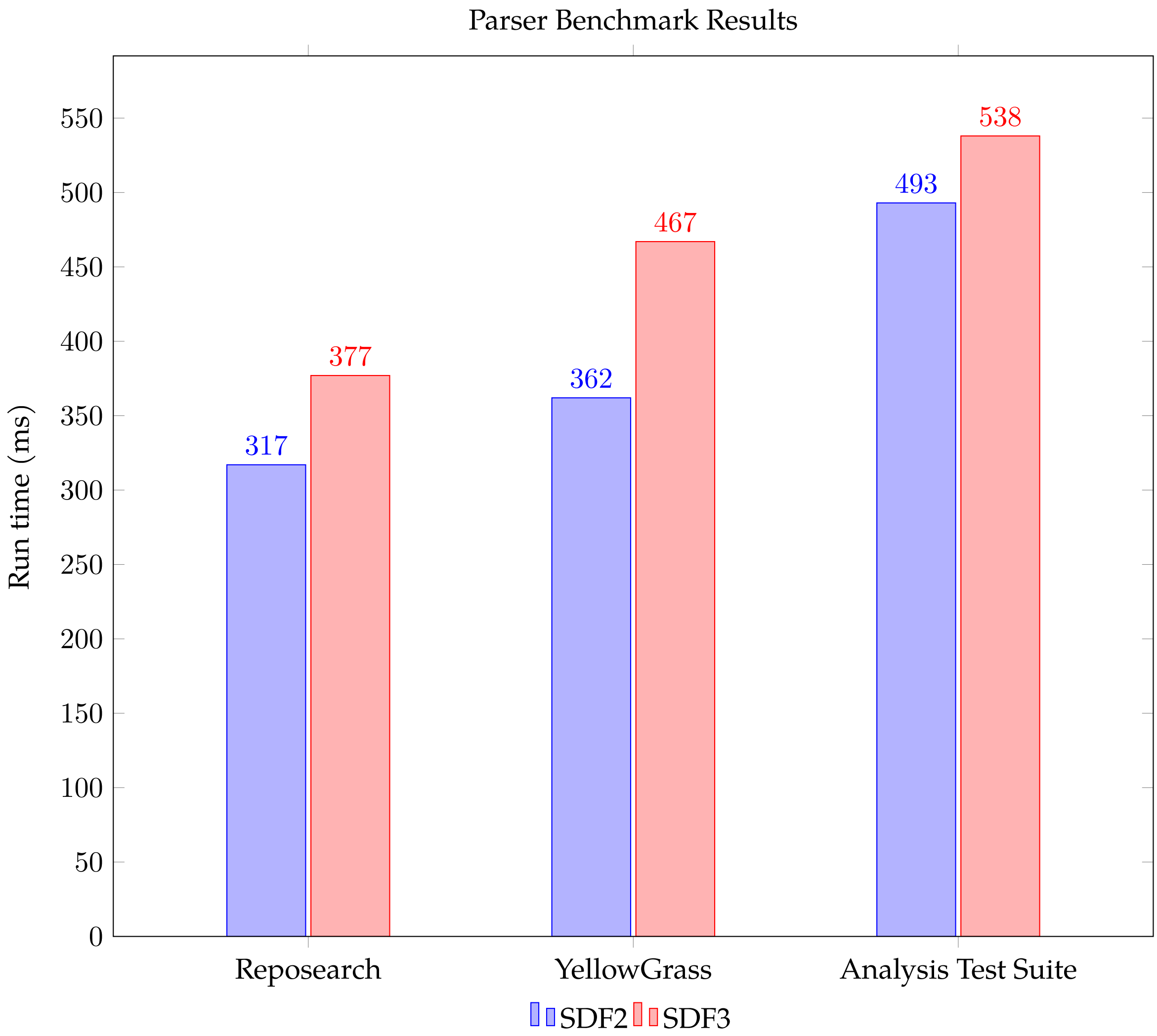


Figure 5.3: Run time of WebDSL SDF2 definition vs. SDF3 definition

that the parse table of the SDF3 specification is larger, we expect that this has a negative effect on the run time. To see the impact of the larger parse table on the run time of the parser, we executed the evaluation on Reposearch, YellowGrass and all files in the analysis test suite, which contains complete WebDSL programs as opposed to the syntax test suite. The analysis test suite consists of 521 small files, with a total of 19.644 lines of code.

To execute the evaluation, we used a 2019 MacBook Pro running macOS Montery 12.2. The machine has a 2,3 GHz 8-core Intel Core i9 with 64 GB RAM available, of which 8 GB was dedicated to the evaluation scripts. The evaluation scripts[4] were configured to parse the described files with the SDF2 parse table, as well as the SDF3 parse table using the JSGLR1 parser implementation. Using the Java Microbenchmark Harness[5], we timed the run time of the parsers using 5 warmup iterations and 10 regular iterations.

The result of benchmarking the run time of the syntax definitions is shown in Figure 5.3. Similar to the growth of the parse table generated from the SDF3 definition, the run time has also increased.

[4]https://github.com/metaborg/jsglr2evaluation
[5]https://github.com/openjdk/jmh

```
context-free syntax

  "for" "(" Id ":" Sort "in" Exp OptFilter ")"
    "{" TemplateElement* "}" ForSeparator -> TemplateElement {cons("For")}

  "separated-by" "{" TemplateElement* "}" -> ForSeparator{cons("ForSeparator")}
                                          -> ForSeparator{cons("None")}
```

(a)

```
context-free sorts

  TemplateElement ForSeparator

context-free syntax

  TemplateElement.For = <
    for ( <VarId> : <Sort> in <Exp> <OptFilter> ) {
      <TemplateElement*>
    } <ForSeparator>
  >

  ForSeparator.ForSeparator     = <separated-by { <TemplateElement*> }>
  ForSeparator.ForSeparatorNone = <>
```

(b)

Figure 5.4: Defining a WebDSL for-loop in SDF2 and SDF3

5.1.3 Maintainability

Two of the goals of introducing SDF3 as successor of SDF2 in the syntax formalism family, were to support more declarative syntax definition and to make the syntax definitions more readable and understandable (Souza Amorim and Visser 2020). Figure 5.4 shows snippets of the SDF2 and SDF3 specifications that define a WebDSL for-loop. Souza Amorim and Visser argue that SDF3 syntax is more similar to other grammar formalisms such as EBNF (Backus et al. 1963) and for this reason we argue that the WebDSL syntax definition in SDF3 is easier to read and understand than its predecessor in SDF2.

However, being easier to read and understand does not automatically make the new syntax definition easier to maintain. The compliance with Statix and its Signature Generator[6] imposes constraints on the grammar, such as disallowing optional sorts, which in the worst case causes the amount of sorts in the grammar to double as described in Section 3.4.2. Additionally, disambiguation without the `prefer` and `avoid` keywords, as described in Section 3.5, removes the unperformant post-parse filters but does create the need for more sorts which artificially complicate the grammar definition.

5.2 Statix

Static consistency checking through static analysis is one of the core aspects of WebDSL (Hemel et al. 2011). However, since no formal semantics of WebDSL are described, we rely

[6]https://www.spoofax.dev/howtos/statix/signature-generator/

on the current implementation of the static analysis in Stratego as the ground thruth.

The evaluation of the Statix specification of WebDSL consists of three parts. First, we list results on the correctness; whether the static analysis allows well-formed programs to pass and whether it gives the correct feedback for erroneous programs. Next, we evaluate the performance in terms of run time on the applications Reposearch and YellowGrass. Lastly, we make qualitative observations on the maintainability of the new Statix specifications.

5.2.1 Correctness

One of the goals of analyzing source code before compiling and running it, is to provide early feedback to the developer regarding possible errors. The range of errors that can be caught early is extensive in WebDSL compared to other programming languages, because of the linguistic integration of user interfaces, request handling, access control and the data model.

For evaluating the correctness of the Statix specification, we first run the static analysis on Reposearch and YellowGrass. Both applications are well-formed programs without errors, therefore the desired result of the static analysis is to report no errors. The result of analyzing the programs with the Statix specifications is shown in Table 5.5. The Statix specification found a handful of errors that are caused by an unimplemented WebDSL feature in Statix, namely static code template expansion. This feature cannot be implemented in Statix due to the absence of String manipulation features, as we discussed in Section 4.9. Apart from this feature, the WebDSL Statix specification considers the applications to be well-typed which is the desired result.

Project	Files	Lines of code	Errors
Reposearch	36	8.722	4
Yellowgrass	75	12.898	6

Table 5.5: Results of running the static analysis on Reposearch and Yellowgrass.

It is trivial to write a program that does not analyse anything and therefore never give an error, and it would technically suit our goal of analysing Reposearch and YellowGrass without errors. To make sure the Statix specification gives feedback when it encounters an incorrect program, we run the Statix static analysis on the analysis test suite of the current implementation in Stratego. In total, the test suite consists of 521 small programs, testing different aspects of the WebDSL language. 273 files contain a correct program and expect the analysis to give no errors, while 248 programs contain in incorrect program where the static analysis must give specific feedback. The expectation as in those 248 files are listed as first lines in the file as comments, and can be one or more of the following.

- Should give an error containing the message s, denoted by `//s`.
- Should give an error containing the message s exactly x times, denoted by `//#x s`.
- Should not show an error with message s, denoted by `//^s`.

The results of running the Statix specification on the analysis test suite is shown in Table 5.6 below.

The result of the analysis test suite shows that the Statix specification of WebDSL is not yet on the level of the current static analysis in terms of correctness. While there are more passing tests than failing tests, there is room for improvement in both categories of the test suite, but particularly in the incorrect programs. The result of the analysis test suite scales with the engineering effort put in, and cannot be solely explained by Statix' shortcomings which we will highlight in the next subsection.

	Test succeeded	Test failed
Correct programs	231	42
Incorrect programs	71	177
Total	302	219

Table 5.6: Results of the analysis test suite.

Discussion of Failed Tests

Failing tests can be divided in categories. While some categories of failing tests are able to be solved by more engineering effort, others are inherent to using Statix as a language for implementing a static analysis.

Error messages that are not specific enough This is not something fundamentally wrong in the Statix language, as it can be improved by more development effort on the Statix specification, but at the cost of the brevity of the specification.

Error messages that Statix can not generate Some tests require an error along the lines of `A function with Signature f(string, int) does not exist`. Statix is unable to format strings to produce such an error. Statix is able to use string concatenation in error messages, but the given expression will always be surrounded by quotes. Apart from the quotes, a lack of string manipulation functions in Statix prevents users from properly formatting AST terms back to human readable signatures.

Cascading errors This is mostly targeted by the test expectation where a certain error s should not be present (`//^s`). Statix as a language is not ideal for these test expectations, as errors are designed to show when the corresponding constraint fails. All generated constraints must either fail or succeed and there is no way to stop execution after a failed constraint, having the result of showing multiple errors. Consider the following expression:

```
var i : Int := nonexistingvariable + 2;
```

The error in this code snippet is that a non-existing variable is being referenced, but the expression cannot be typed properly, resulting in another error on the statement as a whole that says that the expression must be of boolean type.

Unimplemented features WebDSL has an extensive history and many features were added and deprecated over time. However, the deprecated features are still present in the analysis test suite, leading to failing tests. More development effort could make the tests pass.

5.2.2 Performance

Providing early feedback on written code to developers is an essential part of increasing productivity (Becker et al. 2019). For this reason, we want the static analysis to give quick and accurate results. To make claims about the run time of the WebDSL specification in Statix, we run the specification on two open-source applications: Reposearch and YellowGrass, and compare it to the run time of the current static analysis in Stratego.

To execute the evaluation, we used a 2019 MacBook Pro running macOS Montery 12.2. The machine has a 2,3 GHz 8-core Intel Core i9 with 64 GB RAM available, of which 8 GB was dedicated to the evaluation scripts. The evaluation scripts[7] were configured to analyze

[7] https://github.com/metaborg/statix-benchmark/

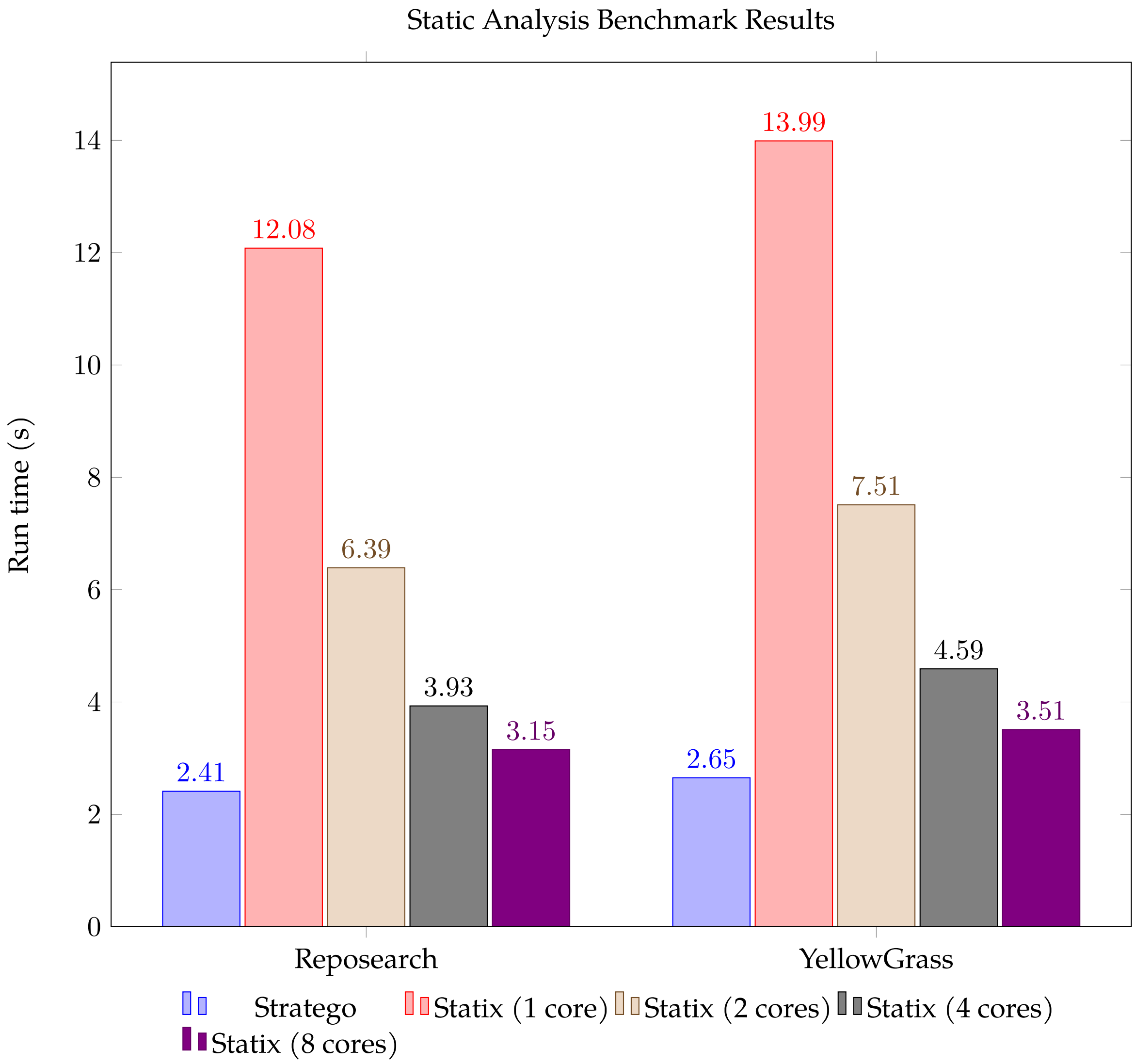


Figure 5.7: Run time of the WebDSL static analysis in Stratego vs. Statix

the two applications. Using the Java Microbenchmark Harness[8], we timed the run time of the Statix specification using 5 warmup iterations and 20 regular iterations. The run time of the current static analysis in Stratego was measured using a shell-script that executed the command-line tool shipped with the WebDSL compiler[9].

Figure 5.7 shows the result of evaluating the WebDSL static analysis in terms of run time. As opposed to the Stratego implementation which always runs on a single core, the Statix specification utilizes all available cores on a machine. Regardless of the amount of cores, the Statix run time scales with the largest file in the project. Due to performance increases such refactoring the WebDSL module system (Section 4.7) and pre-analyzing the WebDSL standard library (Section 4.8), the performance of the largest Statix specification to date is decent. We argue that it is an acceptable run time for use in practice,

5.2.3 Maintainability

The current WebDSL compiler uses Stratego to implement the static analysis. As stated in the introduction of this thesis, we argue that the implementation of many compiler steps

[8] https://github.com/openjdk/jmh
[9] https://github.com/webdsl/webdsl

(e.g. desugaring, type checking, optimization and code generation) in the same Stratego project without clear intermediate representations poses a threat to the maintainability of WebDSL. While the results of the correctness- and performance evaluation look promising, we argue that there is still room for improvement in terms of the maintainability of the Statix specification.

Where Stratego is a more general term transformation language, Statix is developped specifically for implementing static analysis using scope graphs. This smaller domain allows the language engineer to express certain concepts with ease. Examples include declaration, resolving, shadowing and overriding. Additionally, the declarative nature of Statix increases focus on the what, instead of the how. Its syntax reads closer to inference rules, with an predicate being the conclusion, and the constraints being the premises.

A positive aspect of defining the static analysis in Statix is that, as opposed to the more imperative and handcrafted implementation in Stratego, we are able to profit from all new features introduced in Statix such as concurrency (Antwerpen and Visser 2021), incrementality (Zwaan, Antwerpen, and Visser 2022) or better editor services based on the Statix specification (Pelsmaeker, Antwerpen, and Visser 2019).

Unfortunately, the elegancy and brevity of the WebDSL Statix specification is lost due to the amount of boilerplate code, helper functions and code duplication for specific error messages. A lack of provided list functions such as a *zip*, *map* or *filter* function requires multiple definitions of these, essentially duplicates but for different types.

Next, we argue that support for string manipulation would increase the quality of error messages, and reduce code duplication by being able to pass strings to other rules and adjusting them for a specific error message. Next to better error messages, this would also increase the range of language features Statix can support, as discussed in Section 4.9.

Lastly, the extensive amount of time required to analyse the Statix specification itself is a thorn in the side from a language engineer perspective, as it requires more than a minute to run the analysis on the WebDSL Statix specification.

In conclusion, while the maintainability of the WebDSL Statix specification is not perfect, it can be improved when the points above are adressed.

Chapter 6

Related work

In this chapter we discuss work related to compiler front-ends of domain-specific languages. First, we examine published research related to WebDSL that address its grammar and static analysis. Second, we examine work concerning SDF3 and Statix and recent case studies using these meta languages. To conclude, we discuss related work on alternative approaches for implementing the front-end of domain-specific languages.

6.1 WebDSL

The paper of Visser (2007) introduced WebDSL as a case study of Stratego. The work lists most language constructs of the first WebDSL version, combined with SDF2 snippets of their syntax and Stratego code that demonstrates how code is generated for the WebDSL construct. Despite type checking not being mentioned for all language constructs, an extensive example plus implementation in Stratego is given for typechecking a for-loop and the declaration and resolving of a variable is WebDSL. Other work by Groenewegen and Visser present linguistically integrated extensions of WebDSL regarding access control (2008) and data validation (2013).

The work of Hemel et al. (2011) identifies static consistency checking as lacking for web frameworks that were modern at the time and they provide an analysis of how the web frameworks deal with certain consistency checks. Hemel et al. argue that domain-specific languages should be designed for consistency checking, providing a deep-dive on WebDSL as example. Additionally, they present how to perform such consistency checks with Stratego, essentially explaining an early version of the current WebDSL static analysis in Stratego.

The latest publication on WebDSL by Groenewegen et al. (2020) reflects on the WebDSL language as a whole, and provides an experience report of using WebDSL for over 10 years for increasingly ambitious applications.

6.2 Statix

Scope graphs were introduced by Neron et al. (2015) as a language-independent framework for describing name binding in programming languages. Van Antwerpen et al. (2016) build upon this work and extend the scope graph framework with generalized edge labels and introduce a constraint language with a solver that is able to express name binding and typing constraints. In a subsequent publication, Van Antwerpen et al. (2018) further extend the scope graph framework to increase the range of language constructs that can be modelled, such as parameterized types. Additionally, this work introduces Statix as a declarative language to specify type systems. In later work, the performance of Statix was boosted by Van Antwerpen and Visser (2021) through the introduction of the concurrent Statix solver, and by Zwaan et al. (2022) who introduced an incremental Statix solver.

6.2.1 Case Studies

The paper that introduced scope graphs by Neron et al. (2015) contains several illustrations on how scope graphs can model the name binding structure of programs. In the paper, the authors illustrate how to model various concepts of the Language with Modules and Records (LMR) such as let-bindings and, unsurprisingly, modules and records. The extended version (2015) contains examples on definition-before-use, Java packages and imports and C# partial classes. Van Antwerpen et al. (2016) use the LMR example again, but in addition to showing the scope graph, they list the typing constraints for certain snippets, and show an algorithm with rules that dictate how to traverse any LMR program and generate constraints.

The work that introced Statix (Antwerpen, Bach Poulsen, et al. 2018) contained case studies on simply-typed lambda calculus that shows records and structural subtyping, Featherweight Java that presents classes and nominal subtyping. Lastly, the paper contains a case study on System F that illustrates Statix' ability to deal with parametric polymorphism.

The research of Rouvoet et al. (2020) on sound scheduling of name resolution queries also contains various case studies of languages used in practice, modelled in MiniStatix: the core constraint language of Statix, with some extras implemented in Haskell. The case studies are on a subset of name resolution for Java and scala, and the whole of LMR. In their evaluation they used a combination of valid and invalid programs, similar to this thesis. In contrast to this thesis, Rouvoet et al. do not use error message expectations, simply a fail or succeed expectation but the aim of their evaluation was different than ours.

Van Antwerpen et al. (2021) implement a subset of Java in Statix to evaluate their work on real-world codebases, similar to what we show in this thesis. However, their work is not evaluated on erroneous programs, as their goal is to benchmark the parallel Statix solver in terms of run time.

In addition to published research, there are multiple Master's theses that contain Statix case studies:

Aerts (2019) In his Master's thesis, Aerts described, implemented and evaluated an approach for incrementalizing Statix based on separate compilation and dynamic dependency detection. In the evaluation, Aerts implemented a simplified version of Java in Statix that focusses on type-dependent name resolution.

Zwaan (2021) Zwaan provided a runtime that is able to handle composed Statix specifications. He validated his work by integrating Statix specifications of a small subset of SDF3 (Mini-SDF) and Stratego (Mini-STR). Additionally, his Master's thesis contains another case study where the specifications of a small toy-language with expressions, record types, functions and modules (Mod) and a subset of SQL (Mini-SQL).

Wilms (2022) In Wilms' Master's thesis, he introduced PIE DSL 2, a successor of domain specific language accompanying incremental build system PIE (Konat et al. 2019). As part of the PIE DSL 2 implementation, Wilms implemented the static semantics in Statix which modelled many interesting features such as a module system similar to Java, class inheritance and parameterized types.

6.3 Alternative Approaches

The Spoofax language workbench provides meta-DSLs such as SDF3, Statix and Stratego for implementing all aspects of a domain-specific language. However, many alternative approaches exist.

One such approach to defining a static analysis is by using Datalog. Datalog is a subset of Prolog and similar to Statix, it is a declarative logic programming language. Where Statix builds and queries a scope graph, Datalog builds and queries a deductive database. Recent work by Pacak et al. (2020) utlize the performance of Datalog's incremental solvers by expressing algorithmic typing rules in Datalog.

Xtext (Efftinge and Völter 2006) is a language workbench that generates with a heavy focus on generated tooling, such as a compiler infrastructure and IDE support, based on the Xtext grammar language. Xtext utilizes the Eclipse Modeling Framework (Steinberg et al. 2009) and provides an API to customize behaviour of the generated tooling using dependency injection (Eysholdt and Behrens 2010). For example, this customization allows for changing the scoping and name binding rules, or adjusting the code generator.

Rascal is a metaprogramming language, developed by the CWI SWAT group (2009). A commonality of Spoofax and Rascal is that both are in a sense based on the ASF+SDF formalism (Klint 1993). A complete language implementation can be defined using Rascal, including a generated parser, static analysis and code generation. Rascal supports the generation of constraints for program analysis, but does not provide a built-in constrait solver, as they believe hand-crafted specialisations are important for making program analysis scale. In the paper titled *Rascal, 10 years later* (2019), Klint et al. mentions constraint-based type checking as an ongoing project.

The MPS platform by Jetbrains (Dmitriev 2004) is a commercial language workbench with a focus on projectional editing, using representations such as tables and graphs. The first language developed in JetBrains MPS was BaseLanguage, a dialect of Java. The other language definition DSLs of MPS build on BaseLanguage (Pech, Shatalin, and Völter 2013). Similar to Xtext, MPS provides many IDE functionality such as type checking and even code completion out-of-the-box. A recent experimental feature of JetBrains MPS named CodeRules[1] allows for type inference and type checking using logical programming with constraints, similar to Statix.

[1] https://jetbrains.github.io/mps-coderules/about

Chapter 7

Conclusion

In this thesis, we have presented a new front-end for WebDSL. WebDSL is a domain-specific language for web programming, inspired by multiple programming language paradigms. WebDSL is used to create applications such as WebLab and conf.researchr.org, which have thousands of daily users.

We have shown the conversion of the WebDSL grammar from an SDF2 specification to an SDF3 that is disambiguated without post-parse filters, and where the the definition of sorts and constructors can be reused for Statix. The grammar formalism SDF3 generates a parse table which can be executed to efficiently transform textual programs into abstract syntax trees that are used in subsequent components of the compilation chain, such as static analysis.

Next, we presented the static semantics of WebDSL modelled in Statix. Statix is a declarative constraint-based programming language using the concept of scope graphs to model program structures and types. Statix comes with a built-in constraint solver that schedules the constraints in a sounds way, builds and queries the scope graph and is able to show error messages for failing constraints.

The challenges of implementing the new front-end are documented in this thesis and we provided qualitative feedback on how to further improve the meta-DSLs SDF3 and Statix.

Lastly, the resulting modernized front-end of WebDSL was evaluated in terms of correctness and run time performance using large test suites and WebDSL applications that are used in practice.

7.1 Future work

While the modernized WebDSL front-end using SDF3 and Statix is promising, there are many possibilities for improving and extending the work shown in this thesis.

Increased Engineering Effort We argue that the correctness of the modernized WebDSL static analysis scales with the engineering effort put in to the Statix specification (see Section 5.2.1). For the development of web applications with WebDSL, catching more erroneous programs before compilation and reporting telling error messages is essential, especially since the current implementation in Stratego is capable of doing so.

Implement Revised Module System in Compiler Back-End In this thesis we described and implemented a revised module system for WebDSL in Statix (see Section 4.7) that can be described as a traditional module system where (with some exceptions) referencing a declaration made in another module requires importing that module. Additionally, it supports wildcard imports for pragmatic purposes during WebDSL development. Currently, the WebDSL compiler adheres to the module system as listed in the original WebDSL paper

(Visser 2007), where it is described as follows: *"a very simple module system has been chosen that supports distributing functionality over files, without separate compilation"*.

Connect Modernized Front-End to Existing Compiler Back-End The WebDSL static analysis implemented in Stratego is not solely used for error reporting in the IDE. It generates signatures for definitions as discussed in Section 4.9 for which code should be generated, and it creates dynamic rules with name binding and type checking information on which the code generator depends. Re-implementing the current WebDSL back-end to use the Statix analysis results from the Statix Stratego API[1] is a possibility, but this may require a significant amount of time. An alternative is to write a connecting piece of software in Stratego that takes the Statix analysis result as input and generates the correct dynamic rules and AST terms such that the current WebDSL back-end can largely be used as is.

TO-DO: Write about interaction between analysis and transformation rules: https://researchr.org/publi

Evaluate the Incremental Statix Solver The recently published incremental Statix solver by Zwaan et al. (2022) is promising in terms of speeding up the run time of executing Statix specifications. Although their work uses the WebDSL specification that we presented in this thesis, it would be interesting to further evaluate the run time on other (larger) applications such as WebLab and conf.researchr.org, and the impact of specific WebDSL language constructs on the incrementality.

[1] https://www.spoofax.dev/references/statix/stratego-api/

Bibliography

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

Antwerpen, Hendrik van, Casper Bach Poulsen, 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`.

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 and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers". In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark* (*Virtual Conference*). Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ISBN: 978-3-95977-190-0. DOI: `10.4230/LIPIcs.ECOOP.2021.1`. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2021.1`.

Backus, John W. et al. (1963). "Revised report on the algorithmic language ALGOL 60". In: *Comput. J.* 5.4, pp. 349–367. DOI: `10.1093/comjnl/5.4.349`. URL: `https://doi.org/10.1093/comjnl/5.4.349`.

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`.

Dmitriev, Serguei (2004). "Language Oriented Programming: The Next Programming Paradigm". In.

Efftinge, Sven and Markus Völter (2006). "oAW xText: A framework for textual DSLs". In: *Workshop on Modeling Symposium at Eclipse Summit*.

Eysholdt, M. and H. Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, pp. 307–309.

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.

Groenewegen, Danny M. and Eelco Visser (2008). "Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns". In: *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*. Ed. by Daniel Schwabe, Francisco Curbera, and Paul Dantzig. IEEE, pp. 175–188. isbn: 978-0-7695-3261-5. doi: 10.1109/ICWE.2008.15. url: http://dx.doi.org/10.1109/ICWE.2008.15.

— (2013). "Integration of data validation and user interface concerns in a DSL for web applications". In: *Software and Systems Modeling* 12.1, pp. 35–52. doi: 10.1007/s10270-010-0173-9. url: http://dx.doi.org/10.1007/s10270-010-0173-9.

Hemel, Zef et al. (2011). "Static consistency checking of web applications with WebDSL". In: *Journal of Symbolic Computation* 46.2. Automated Specification and Verification of Web Systems, pp. 150–182. issn: 0747-7171. doi: https://doi.org/10.1016/j.jsc.2010.08.006. url: http://www.sciencedirect.com/science/article/pii/S0747717110001367.

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.

Klint, Paul (1993). "A Meta-Environment for Generating Programming Environments". In: *ACM Transactions on Software Engineering Methodology* 2.2, pp. 176–201. doi: 10.1145/151257.151260. url: http://doi.acm.org/10.1145/151257.151260.

Klint, Paul, Tijs van der Storm, and Jurgen J. Vinju (2009). "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, pp. 168–177. isbn: 978-0-7695-3793-1. doi: 10.1109/SCAM.2009.28. url: http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28.

— (2019). "Rascal, 10 Years Later". In: *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*. IEEE, p. 139. isbn: 978-1-7281-4937-0. doi: 10.1109/SCAM.2019.00023. url: https://doi.org/10.1109/SCAM.2019.00023.

Konat, Gabriël et al. (2019). "Precise, Efficient, and Expressive Incremental Build Scripts with PIE". In: *Second Workshop on Incremental Computing (IC 2019)*.

Neron, Pierre, Andrew Tolmach, 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.

Neron, Pierre, Andrew P. Tolmach, et al. (Jan. 2015). *A Theory of Name Resolution with extended Coverage and Proofs*. Technical Report TUD-SERG-2015-001. Extended version of ESOP 2015 paper "A Theory of Name Resolution". Software Engineering Research Group. Delft University of Technology.

Pacak, Andre, Sebastian Erdweg, and Tamas Szabo (2020). "A systematic approach to deriving incremental type checkers". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA. doi: 10.1145/3428195. url: https://doi.org/10.1145/3428195.

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., Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)". In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.26. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.26.

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.

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 (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.

Wilms, Ivo (2022). "Extending the Domain Specific Language for the Pipelines for Interactive Environments build system". MA thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:567a7faf-1460-4348-8344-4746a18fb0b1.

Zwaan, Aron (2021). "Composable Type System Specification using Heterogeneous Scope Graphs". MA thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:68b7291c-0f81-4a70-89bb-37624f8615bd.

Zwaan, Aron, Hendrik van Antwerpen, and Eelco Visser (2022). "Incremental Type-Checking for Free: Using Scope Graphs to Derive Incremental Type-Checkers". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2. DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3563303.

Acronyms

AST abstract syntax tree

DSL domain-specific language

Appendix A

A