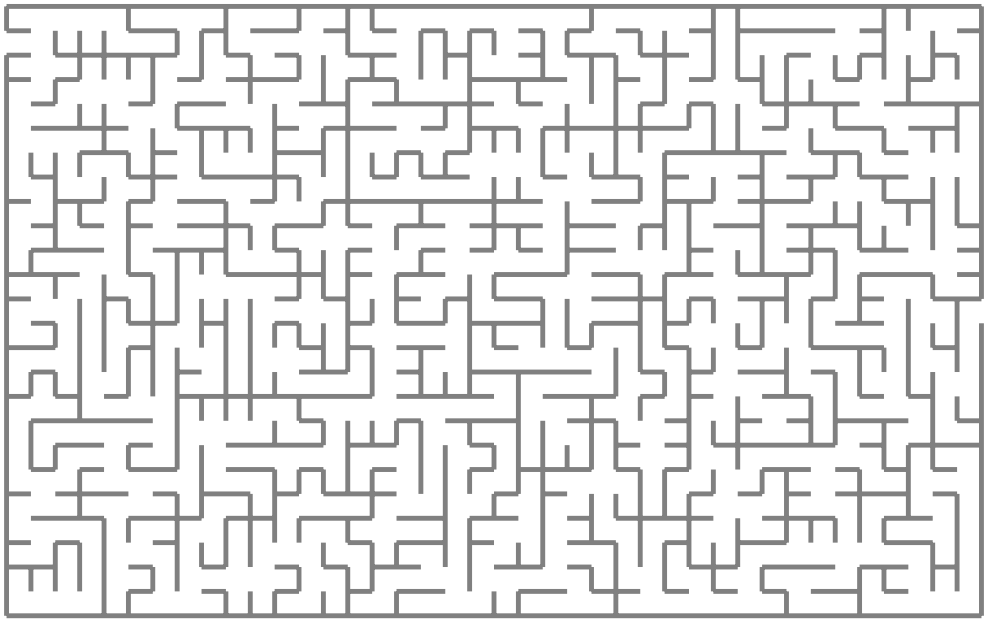
Modernizing the WebDSL front-end: A case study in SDF3 and Statix

Version of March 10, 2021



Max Machiel de Krieger

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

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Max Machiel de Krieger
born in Delft, the Netherlands

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

Cover picture: Random maze.

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

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

Abstract

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

Thesis Committee:

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

Preface

Preface here.

Max Machiel de Krieger
Delft, the Netherlands
March 10, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Broad Picture

Many different programming languages exist, with many different properties and advantages. (*TO-DO: this is crap, need something better*)

Programming Langauge Front-end Introduction

The implementation of a programmig languages can be seperated into two parts: the front-end and the back-end. The front-end is the part of the programming language with which the user interacts (the syntax, early feedback using analysis results) and the back-end is the part that makes the programming language operational (optimization, code generation). While the back-end of a programming langauge makes it work, the front-end defines how a user experiences the progrmaming language (*TO-DO: read papers about this*).

WebDSL

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

Problem Description

Because of its academic nature, many research projects added features to the language, all contributing to the success of existing WebDSL applications. The downside of having many different contributors adding new features, is that the development experience that comes from the front-end leaves much to be desired. (*TO-DO: too harsh?*) Currently, the WebDSL implementation is composed of multiple definitions in meta-DSLs supported by the Spoofax language workbench: the syntax is defined in SDF2 and the desugaring, typechecking, optimization and code generation is defined in the term transformation language Stratego. The interleaving of all the latter processes in the same Stratego environment poses a threat to the readability and maintainability of the WebDSL language.

1.1 Contributions

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

- We present an implementation of the WebDSL grammar in SDF3.

- We present an implementation of the WebDSL static analyses in Statix.
- We assess the completeness and performance of the WebDSL SDF3 and Statix implementation.
- We provide qualitative feedback about the development process with SDF3 and Statix.

1.2 Outline

The rest of this thesis is structured as follows. In chapter 2 we describe WebDSL, its features and its current implementation. Next, chapter 3 and 4 go in detail about the new implementation of the WebDSL front-end in SDF3 and Statix respectively. The result of this implementation is evaluated in chapter 5 and compared with related work in chapter 6. Finally, chapter 7 concludes this thesis.

Chapter 2

WebDSL

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

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

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

2.1 User Interfaces

- Syntax
- Data binding and action code
- Template overriding
- Template overloading
- Dynamically scoped redefines

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

3.1 Introduction to SDF3

- Syntax
- Disambiguation

3.2 WebDSL Grammar

3.3 Migration from SDF2 to SDF3

- Preparation for migration:
  - Remove alternations
  - Remove restrictions
  - Remove parameterized sorts and parameterized imports
- Duplicate constructors, normalize constructors with Stratego desugaring
- Priority chains are not transferred
- Comments removed
- Implement templates for pretty-printing
- Merge syntax definition of previously mixed HQL

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

- Syntax
- Usage in rules
- Difference from SDF3

3.4.2 Statix Signature Generator

- Conversion from SDF3 constructors to Statix
- Optional sorts
- Injections
- Disambiguation (ref to separate section)

3.4.3 Changing the SDF3 WebDSL Specification

- Lots of new sorts due to explicit optional sorts
- Desugar new optional sorts
- Add more constructors for the removal of injections

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 go in detail about the implementation of the WebDSL static semantics in Statix, according to the defined semantics in chapter *TO-DO*. First, we introduce the meta-DSL Statix. Next, we describe the implementation of a simple type system in Statix. Lastly, we zoom in on the challenges faced while implementing non-trivial WebDSL features in Statix.

4.1 Introduction to Statix

- Syntax
- Scope graphs
- Relations and queries
- Boolean logic

4.2 Simple Type Systems

4.2.1 Constant Expressions

- Typing rules
- Type compatibility

4.2.2 Variables

- Declaration and resolving
- Prevent duplicates
- Declare before use
- Shadowing

4.2.3 Entities and properties

- Declaration and resolving
- Prevent duplicates
- Parameters

4.2.4 Functions

- Declaration and resolving
- Prevent duplicates
- Parameters

4.2.5 Pages and Templates

- Declaration and resolving

4.3 Inheritance

Linking the Scopes

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

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

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

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

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

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

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

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

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

Overwriting Functions

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

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

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

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

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

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

```
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.
```

Entity Type Compatibility

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

With the addition of entity inheritance, we need to expand this definition. To this end, we added the rules as shown in listing *TO-DO*. Given two entity scopes, the `inherits(s_sub, s_super)` predicate returns true when the query has one result. The query in the `inherits` rule requests

all paths from scope `s_sub` to scope `s_super` consisting of only `INHERIT` edges. Such a path exists if and only if the entity belonging to scope `s_sub` inherits the entity belonging to `s_super`.

```
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().
```

4.4 Entity Extension

4.5 Function and Template Overloading

4.6 Module system

4.7 TO-DOs in Statix spec

Statix spec TO-DOs that most likely are interesting:

- HQL
- Access control reference checking
- Inclusing `built-in.app` in analysis

4.8 Possibly: Compose HQL and WebDSL's Type System using Aron's Thesis

4.9 Reflection on Statix

Chapter 5

Evaluation

5.1 Correctness

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

5.2 Validation

- How elegant in the Statix implementation?

5.3 Performance

- Explain metrics and methods
- Results
- Discuss results

5.4 Evaluating Statix

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

Chapter 6

Related work

- Papers about WebDSL
- Papers about modern Spoofax
- Papers about the definition of web programming languages
- Papers about modernizing (web) programming languages

Chapter 7

Conclusion

7.1 Future work

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

Bibliography

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.

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

Acronyms

AST abstract syntax tree

DSL domain-specific language

Appendix A

A