# Comparing Static Semantics Specifications for the IceDust DSL: A Case Study of Statix

*Version of September 7, 2023*

Jesse Tilro

# Comparing Static Semantics Specifications for the IceDust DSL: A Case Study of Statix

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jesse Tilro
born in Dordrecht, the Netherlands

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

© 2023 Jesse Tilro.

The source code and evaluation scripts discussed in this thesis can be obtained from:
`https://github.com/jessetilro/thesis`

# Comparing Static Semantics Specifications for the IceDust DSL: A Case Study of Statix

Author: Jesse Tilro
Student id: 4368142

**Abstract**

Reusable tools for engineering software languages can bridge the gap between formal specification and implementation, lowering the bar for engineers to design and implement programming languages. Among such tools belong NaBL2 and its successor Statix, which are meta-languages for declaratively specifying the static semantics of programming languages and generating typecheckers accordingly.

Although Statix intends to cover the domain of static semantics specification to a greater extent than NaBL2, less is known about how the meta-languages compare in terms of their practical usability.

In this thesis, we perform a case study in which we apply Statix to define the semantics of IceDust, an incremental computing DSL for modeling data with relations, and compare it to a prior NaBL2 specification.

We compare the novel and prior specification in order to determine how the meta-languages, when applied to the case of IceDust, compare in terms of high-level characteristics: expressiveness, readability, implementation effort and runtime performance. We perform four evaluations to this end: a qualitative in-depth comparison of the specifications, a measurement of specification sizes, an evaluation of correctness and a runtime performance benchmark of the resulting type checkers.

Our findings suggest that although Statix has a larger coverage of possible language definitions, in the case of IceDust it is a less expressive formalism for defining the static semantics and generates a slightly less performant type checker when compared to NaBL2. We find areas of interest for future work aiming to improve the practical usability of Statix, namely the definition of type compatibility relations, the way data in the scope graph are stored and retrieved and the integration with the compiler back-end.

Thesis Committee:

Chair: Dr. M. T. J. Spaan, Faculty EEMCS, TU Delft
Committee Member: Dr. B. P. Ahrens, Faculty EEMCS, TU Delft
University Supervisor: Ir. D. M. Groenewegen, Faculty EEMCS, TU Delft

# Preface

Six eventful years after starting the master's program in Delft, writing this preface marks a moment of great relief for me. I would like to take this opportunity to thank the people who helped me see it through to the end.

I would like to thank my supervisor, Danny Groenewegen, who supported and advised me for over three years. Whenever I strayed off course, he was there to set my priorities straight and encourage me to take another step. Also, I am thankful to the members of my committee, Benedikt Ahrens and Matthijs Spaan, for helping me improve the quality of my work and guiding me toward the defense. My thanks go out as well to everyone at the research group who took an interest in my project, provided me with technical support, or shared their experiences with me. In the same spirit, I thank David Alderliesten, Floris Doolaard, and Niels Warnars, with whom I graduated for the bachelor's program and stayed in touch ever since. They inspired me to finish the work on my thesis as they set an example and encouraged me.

In the meantime, for the past seven years, my job has been an important anchor for me. I would like to thank the original company founders Michel Fiege, Sem Goedknegt, Frank Groeneveld, Martijn Reijerse, and Bas Schoenmakers for their trust in me, and their willingness to adapt to my changing needs and availability during my studies. The working environment they created and developed over time allowed me to grow professionally without sacrificing my academic ambitions. I thank my colleagues for all the interest, feedback, and encouragement I received from them as I progressed with my thesis.

I am grateful for the love and support of my parents, Peter and Corinne, and their partners, my girlfriend, family, and friends, who helped me go through challenging times and enabled my perseverance.

Finally, I would like to pay gratitude to the late professor Eelco Visser, who welcomed me into the programming languages research group. His positivity, patience, and understanding with me during our process of defining a thesis project that fit my personal strengths and interests meant a great deal to me.

Jesse Tilro
Dordrecht, the Netherlands
September 7, 2023

# Contents

# List of Figures

# Chapter 1

# Introduction

Programming languages are fundamental tools for developing computer software. They allow developers to express their intent in a structured manner that strikes a balance between human readability on one hand, and machine processability on the other. Over the years, extensive research has been conducted to advance programming languages and their associated tools, aiming to improve productivity, code quality, and software reliability.

Programming languages can be formalized, meaning that their syntax and semantics are defined using particular definition formalisms, which creates the opportunity to prove that the language has certain properties such as type safety and functional correctness. These proofs are valuable as they ensure that these properties hold for any program written in the language and thus provide guarantees for programmers. However, the actual implementation of programming languages involves creating various tools, including compilers and editor services, to make the language practically usable. Maintaining separate formalizations and implementations can be labor-intensive, as they should be kept in alignment with each other: changes to the formal specification of a language should be correctly implemented, and vice versa.

For this reason, bridging the gap between formal specifications and language implementations is one of the focuses in research revolving around programming language theory. It aims to identify underlying concepts in language semantics, such as recurring patterns in type systems and common name binding structures, to approach them more generically. By doing so, reusable tools for language engineering can be constructed.

Reusable tools for language engineering allow for streamlining the process of designing custom domain-specific languages (DSL) for specific problem domains. Such languages equipped with the appropriate constructs and abstraction mechanisms enable programmers to reason more effectively about their programs and domains compared to general-purpose languages or frameworks. Moreover, domain-specific languages provide additional guarantees and safeties at the language level, allowing for earlier and more specific warnings to programmers during development. Reusable tools also facilitate faster iterations in language design and optimization.

This thesis specifically investigates Spoofax, a language workbench, and two of the meta-languages it features for static semantics specification: NaBL2 and Statix. Statix was designed as a successor to NaBL2. Since NaBL2 and its underlying theoretical framework have appeared to be limited to modeling simple, nominal type systems, some generalizations in the design of Statix have been made that render it applicable to a larger range of type systems with more sophisticated patterns. These generalizations are described in more detail in Section 2.4. The main motivation for the initial research on Statix has thus been to achieve a greater coverage of the domain of static semantics specification than NaBL2. However, less is known about the implications of these design differences when it comes to the practical applicability of Statix compared to NaBL2. This formed the main motivation for this the-

sis, in which we perform a case study with the meta-languages and compare their practical applicability.

Our methodology for conducting this case study was based on the framework proposed by Runeson and Höst (2009). The objective is to compare the two meta-languages in terms of high-level characteristics that include expressiveness, readability, implementation effort and runtime performance. We study the case of the IceDust language, as the studied meta DSLs are applied to express the static semantics of this object language, which involves the definition of its type system and name binding structures. Our motivation for selecting IceDust as case for static semantics specification, is that its semantics touch upon various features of the meta DSLs and therefore it has the potential to reveal insights about them throughout the research process.

As part of this case study, we aim to answer the following research questions:

RQ1. Can the static semantics of IceDust be expressed using Statix, such that the resulting typechecker performs analyses to the same level of correctness as the one based on the NaBL2 specification?

RQ2. How significantly do NaBL2 and Statix differ in expressiveness, readability and implementation effort when comparing their specifications for IceDust, and which meta-language design choices are most impactful on this?

RQ3. How do NaBL2 and Statix compare in terms of the runtime performance of their resulting type checkers for IceDust?

The differences in expressiveness between NaBL2 and Statix when applied to the static semantics of IceDust is of main interest in this thesis. This topic is therefore covered first. Chapter 3 is dedicated to describing certain parts of the specifications that comprise a non-trivial solution to a particular challenge in declaring the semantics, and comparing them in-depth. Then, primarily based on these comparisons, we evaluated the differences in expressiveness between the meta-languages and answered research question RQ2, which is described in evaluation section 4.1. In order to validate this comparison and answer research question RQ1, we evaluated the correctness of our specification which is described in evaluation section 4.2. Finally, to answer research question RQ3, we performed a performance benchmark that is described in evaluation section 4.3.

The main contributions of our research are:

1. We define a specification of the static semantics of the IceDust language project implemented in Statix, that covers the same semantics defined in the prior NaBL2 specification. This allows for the evaluations performed as part of this case study, but may also render the IceDust language applicable as object of future research.

2. We share as artifacts with this thesis the reproducible scripts used in performing our evaluations, including a benchmark pipeline for performance comparison between NaBL2 and Statix based static analyses and a test coverage estimation of SPT tests on Statix constraints. These scripts are object-language parametric, and may therefore be reused or built upon in similar studies.

3. We provide insights about four high-level language characteristics of the NaBL2 and Statix meta DSLs for static semantics specification, namely implementation effort, readability, expressiveness and performance. We present these insights categorized by three areas: type compatibilities, scope graphs and interfaces between compiler pipeline stages.

The remainder of this thesis is structured as follows. In Chapter 2 we provide background information on some of the important concepts that are involved in this thesis. Next, in Chapter 3 we describe the research that was conducted for this thesis, involving the post-facto formalization of the static semantics of the IceDust language using Statix and comparing it to the prior NaBL2 specification. We present this structured according to three areas that proved to be most significant in answering our research questions: type compatibilities, scope graphs and interfaces between compiler pipeline stages. Then, as part of Chapter 4, we describe how we evaluated our work and answered our research questions through various methods of comparing our specification with the prior specification. We discuss related work in Chapter 5. Finally, we present our conclusions in Chapter 6.

# Chapter 2

# Background

In this chapter we provide an introduction to several concepts that are of importance in this study. We elaborate upon: IceDust, a DSL that serves as the case being studied; Spoofax, the language workbench in which the studied meta-languages are integrated; NaBL2, the predecessor meta-language under comparison; and Statix, the successor meta-language under comparison.

## 2.1 IceDust

In this case study, the IceDust language was studied as case for static semantics specification. IceDust is a declarative domain-specific language allowing the specification of a data model with relations and its associated business logic (D. C. Harkes and Visser 2017) (D. C. Harkes, Groenewegen, and Visser 2016). It solves various challenges in modeling relations using dedicated language features, and enables the declaration of business logic associated with the data model independent of the computational strategy used to evaluate it.

### 2.1.1 Modeling relations

A preliminary study leading up to the development of IceDust was done by Harkes (D. Harkes and Visser 2014). This study analyses the design space of modeling data with relations. The analysis identifies and organizes approaches to modeling relations in a matrix along two orthogonal dimensions:

1. Modeling paradigms, concerning different approaches to modeling data: relational, object-oriented, object-role modeling (ORM) and graph database.

2. Relation models, concerning different datastructures as representations of a relation: edge, tuple and object.

As a result of the analysis, a novel modeling language forming the basis of IceDust is proposed that unifies and generalizes features from different approaches in the design space that contribute to desired characteristics in modeling relations. Being aligned mostly with the ORM approach and modeling relations as objects, the language aims to integrally solve challenges identified in dealing with multiplicities, navigation, arity and first-class citizenship of relations.

#### Multiplicities

Encoding *one-to-many*/*many-to-many* relations as collections of values while encoding (optional) *one-to-one*/*many-to-one* relations as (nullable) singleton values yields a discontinuity in programming style. The issues associated with this discontinuity include (1) having to

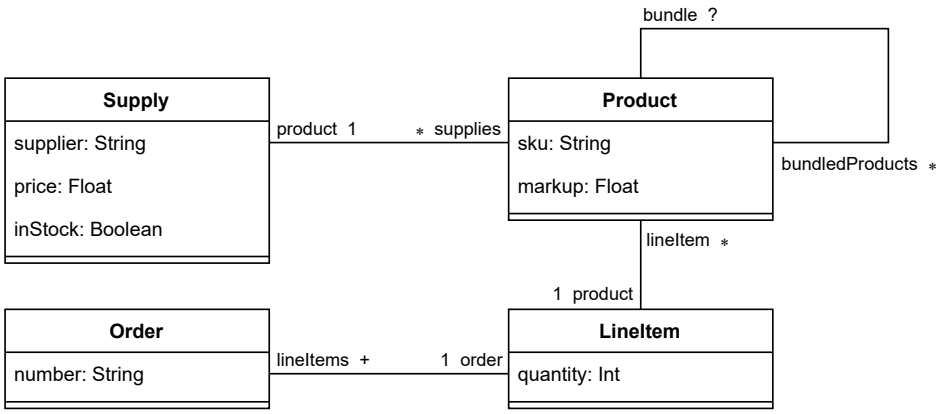



Figure 2.1: Class diagram of a running example data model. Associations have roles for both directions, and multiplicities are denoted using IceDust's multiplicity modifiers.

explicitly unwrap collection values and null-check nullable values; (2) accounting for constraints that the type system imposes on the different return types (`Collection<Type>` versus `Type`); and (3) dealing with different call semantics. These issues can be solved by incorporating multiplicities as a native feature into the modeling language and abstracting over the possible cardinalities of values, allowing the results of querying relations to be interacted with uniformly. IceDust supports multiplicity modifiers expressing that fields may have exactly one value (represented by `1`), zero or one value (represented by `?`), zero or more values (represented by `*`) or one or more values (represented by `+`).

#### Navigation

In navigating relations, it is desirable that relations can be navigated bidirectionally and that navigations in both directions can be expressed concisely. The representation of a relation as an object introducing indirection, the ORM style of navigation using named roles, and the support for shortcuts and inverses of the proposed language all contribute to this end.

#### First-class citizenship

First-class citizenship in the context of relations means according to Harkes that (1) relations can have attributes and (2) relations can be the subject in other relations. Together with named roles this is identified as a prerequisite for enabling n-ary relations.

#### Arity

By supporting n-ary relations, relationships of any arity, i.e. between an arbitrary amount of objects, can be expressed.

As an example, consider the data model depicted in figure 2.1, representing a simplified e-commerce retail system. The model contains products (Product) that can be recursively aggregated in bundles. Products may be in stock and offered for varying prices by different suppliers (Supply). Orders (Order) may consists of varying quantities of different products (LineItem). This model can be expressed in IceDust using the code depicted in figure 2.2.

### 2.1.2 Business Logic

In addition to specifying a data model with relations using aforementioned language constructs, IceDust also allows for implementation of business logic associated to the data model. Regarding the execution model, an IceDust program consists of a list of entities with fields,

```
entity Product {
  sku : String
}

relation Product.bundledProducts * <-> ? Product.bundle
relation Product.supplies * <-> 1 Supply.product
relation Product.lineItems * <-> 1 LineItem.product

entity Supply {
  supplier : String
  price : Float
  inStock : Boolean
}

entity LineItem {
  quantity : Int
}

entity Order {
  number : String
}

relation Order.lineItems + <-> 1 LineItem.order
```

Figure 2.2: IceDust specification defining the model depicted in figure 2.1.

where fields can be attributes and relations. There is a distinction between base attributes and derived attributes, the former being assigned a value through user input, and the latter yielding a value that is the result of a calculation. Similarly, relations can also be derived. Thus, derived fields enable a means of implementing business logic.

The calculation of derived attributes and relations can be declaratively specified as an expression in terms of literals and other fields. The language supports a reactive programming paradigm by automatically tracking dependencies between attributes and having changes to base attributes propagate to their dependent derived attributes. The propagations can be transitive as derived attributes may depend other derived attributes. Subexpressions may be abstracted and made reusable as parameterized functions which are declared in a separate section of and IceDust program.

To exemplify these derived attributes and relations we extend our running example in figure 2.3. We define a derived relation that relates a products with its best supply, best being defined as being in stock and having the lowest purchase price. This relation can be expressed as the composite derived relation shown in Figure 2.4. Our extension also includes derived attributes for the prices of a product, line item and order, which are defined in figure 2.5.

The declarative approach to defining calculations enables a separation between the concern of specifying the semantics of the calculation and specifying its implementation strategy. An implementation strategy for a derived attribute calculation can be briefly declared using a single keyword. The supported implementation strategies of calculations are: on-demand, incremental and eventual. On-demand calculation recomputes the value of a derived attribute each time it is read, analogous to a getter method implementation in an object-oriented language. Incremental calculation recomputes the value of a derived attribute on

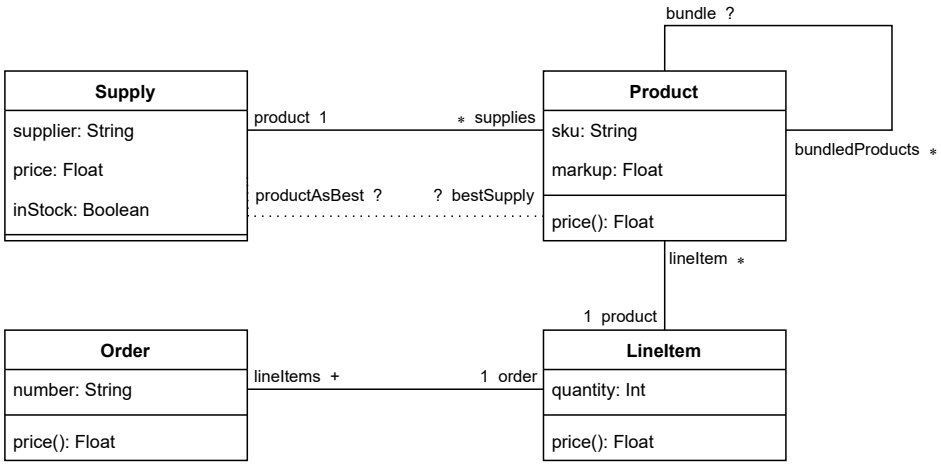



Figure 2.3: The class diagram of figure 2.1 extended with derived relations and attributes.

```
relation Product.inStocks * =
  supplies.filter(x => x.inStock)
  <-> ? Supply.productInStock

relation Product.bestSupply ? =
  inStocks.find(x => x.price == (min(inStocks.price) <+ 0.0))
  <-> ? Supply.productAsBest
```

Figure 2.4: IceDust specification of the two derived relations in the running example.

```
entity Product {
  sku : String
  markup : Float
  price : Float ? = sum(bundledProducts.price)
          <+ (bestSupply.price * (1.0 + (markup / 100.0)))
}

entity LineItem {
  quantity : Int
  price : Float ? = product.price * quantity as Float
}

entity Order {
  number : String
  price : Float ? = sum(lineItems.price)
}
```

Figure 2.5: IceDust specification of the derived attributes in the running example. For brevity, the declarations of the Supply model as well as all relations have been left out of this snippet.

each write to a dependency. Eventual calculation dirty flags dependent derived attributes on a write to a dependency, and has a separate thread eventually recompute their derived values.

Calculation strategies can thus be declaratively specified on a per attribute basis without any changes to the corresponding calculation expression, and can be soundly composed. This allows tuning calculation strategies to improve non-functional quality characteristics of the program such as performance with minimal programming effort, contrary to many other formalisms.

### 2.1.3 Static Analysis

One of the main premises of the IceDust language is to provide as many static guarantees as possible. The benefits of this include being able to warn the programmer early in the editor, preventing unnecessary runtime errors and ensuring consistency of the runtime behavior. This does introduce the necessary complexities in the static semantics of the language.

The variability in IceDust programs is modeled in a feature model revolving around the main language construct, namely the field. The features of a field are orthogonal and can therefore be independently configured. These features include for example the multiplicity constraint, the derivation type, and the calculation strategy. However, to ensure soundness of the specification, certain constraints are imposed on the combination of features.

These constraints translate into the type checking approach. Types in IceDust are represented by tuples of three lattice values. The lattices involve: the data type; the multiplicity and ordering; and the calculation strategy. Checking whether for example the actual type of an expression meets the expected type (e.g. according to the static type signature) amounts to testing whether each value in the actual type tuple is lower in the lattice than its respective value in the expected type tuple. Determining the type tuple of a particular term may amount to aggregating the values of the type tuples of its subterms in different ways. For example for a particular binary operation, the resulting data type may be the least upper bound of the data types of the operands, while the multiplicity may be derived in way that is more specific to the semantics of that operation.

Furthermore, fields can be declared to belong to an entity in different ways. As entities can be declared to extend one another, fields may be inherited and overridden. Also, a relation declaration in fact implies that the entities involved in the relation are extended with fields, those fields being the roles of the relation. This yields a number of non-lexical binding patterns. On top of this, resolving references to fields is non-trivial. An expression accessing a member of a subexpression requires the type of said subexpression to be determined before the member reference can be resolved to the corresponding field declaration. This shows a case where name resolution is type dependent.

This combination of complexities in the static semantics make IceDust an interesting case for static semantics specification using Statix, and comparison with its NaBL2 counterpart.

## 2.2 Spoofax

Spoofax is a language designer's workbench under active development by the Programming Languages research group at Delft University of Technology (Visser 2010). It features range of highly declarative meta DSLs that allow for writing definitions of various aspects of a programming language, including its concrete and abstract syntax, and static and operational semantics. The specifications for each of these aspects are automatically compiled to respective components of the compiler pipeline, as well as tools such as editor services. Within the context of this case study, the most relevant DSLs in Spoofax are the following.

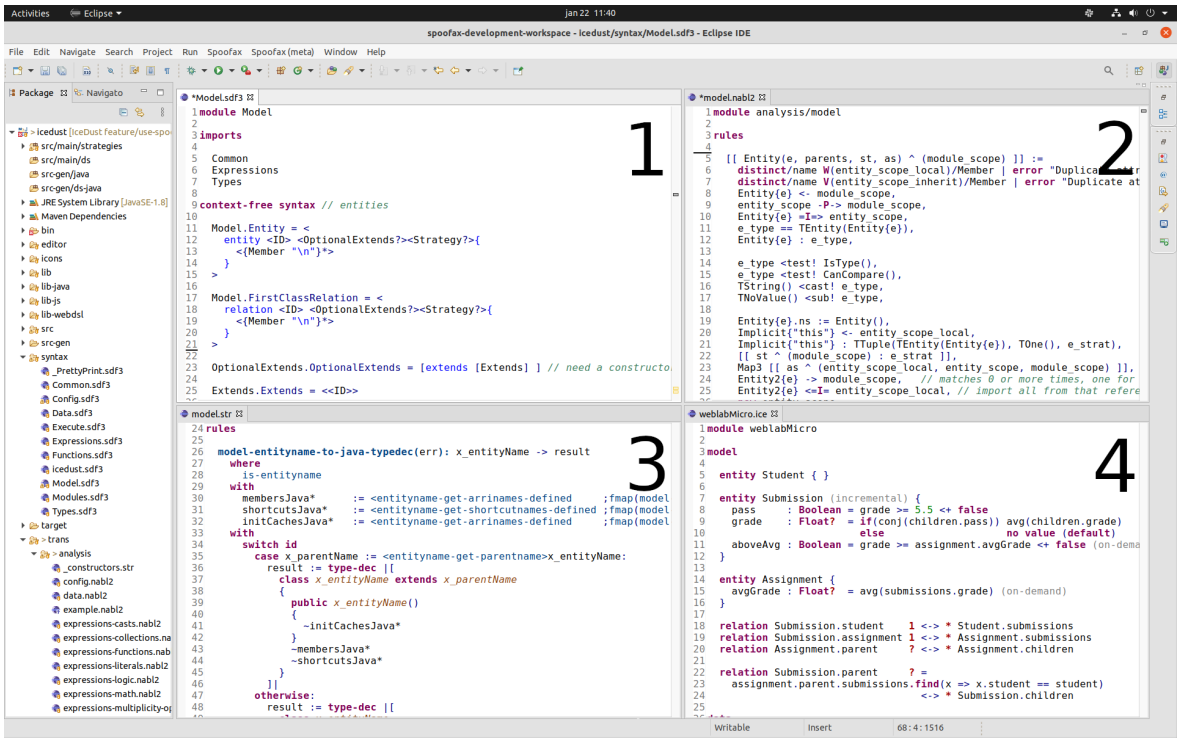



Figure 2.6: Screenshot of the Spoofax language workbench editor window. The viewports labelled with numbers involve: (1) a part of the syntax specification in SDF3 of IceDust; (2) a part of the static semantics specification in NaBL2 of IceDust; (3) a part of the dynamic semantics specification in Stratego of IceDust; (4) a part of an example IceDust program.

- SDF3, a context-free grammar formalism for specifying both the concrete and abstract syntax of a programming language in alignment. Production rules may be declared in template form such that not only a parser but also a pretty printer can be derived from them. It also supports filters for disambiguating ambiguous grammars. From a specification, a parse table is generated that is then fed to an SGLR parser generator (Souza Amorim and Visser 2020).
- NaBL2, for specifying the static semantics of a programming language in terms of name binding and type checking rules (Van Antwerpen, Néron, et al. 2016).
- Statix, also for specifying static semantics by means of user defined constraints, and intended as the successor to NaBL2 (Van Antwerpen, Bach Poulsen, et al. 2018).
- Stratego, a term rewriting language used for defining program transformations. A specification consists of rewriting rules along with strategies that declare how to traverse the AST and apply the rules. Transformations can, among others, be used to define the operational semantics of a programming language (Visser, Benaissa, and Tolmach 1998).
- SPT, the Spoofax Testing language for declaring automated test cases for a language project.

The workbench is built on top of the Eclipse IDE and the Eclipse Modeling Framework (EMF), and is implemented in Java. Due to the fact that editor services are dynamically loaded and language parametric, an object language may be developed and used side-by-side at the same time. This allows for an iterative and incremental approach to language design, where the language is developed in an exploratory manner and evolves inductively. This increases

the productivity of language engineers when compared to handwriting compiler pipeline components and editor services.

## 2.3 NaBL2

NaBL2, which is an acronym for Name Binding Language, is a meta-DSL integrated into the Spoofax Language Workbench. It enables declarative specification of the static semantics of a programming language in terms of name binding and type checking rules. NaBL2 is a successor to the NaBL/TS meta-language, and its latest implementation is not directly formalized in a publication, but rather results from a line of work in declarative static semantic analysis. Early work instroduces a first iteration of the declarative meta-language and algorithm and its integration into the Spoofax language workbench (Konat et al. 2013). This work is extended by further development of the theoretical framework upon which NaBL2 is based, introducing the theory of scope graphs (Neron et al. 2015). Later work integrates the theory of scope graphs into the meta-language to practically allow building type checkers based on them (Van Antwerpen, Néron, et al. 2016).

The NaBL2 meta-language takes a constraint-based approach to static semantics analysis by utilizing a particular constraint language internally (Van Antwerpen, Néron, et al. 2016). It compiles a specification to a language-dependent extractor that accepts an abstract syntax tree and produces a set of constraints. These constraints express the requirements that should be met in order for the program to be well-typed and well-bound. Language-independent constraint solving and unification algorithms are then used to produce a set of name and type assignments satisfying these constraints.

Theoretically, in static semantics formalization, there exists an interdependence between the disciplines of name resolution, type resolution and scope graph construction. For example, there may be cases in which name resolution is type dependent. Because of this interdependence, all constraints in NaBL2 are expressed and solved integrally using a uniform mechanism. Nevertheless, for the purpose of supporting each of these disciplines, the produced constraints are divided into three categories: scope graph constraints, resolution constraints and typing constraints. A scope graph is an abstract representation of the name binding structure of a program. It consists of scopes, declarations and references. A scope describes AST nodes that behave the same in terms of name binding; a declaration describes an identifier that introduces a name; and a reference describes an identifier that refers to a declaration. In NaBL2, internally, a scope graph is represented directly as the solution to a set of scope graph constraints. These constraints express which declarations and references belong to which scope, as well as the associations between scopes. Resolution constraints express that a particular reference must resolve to a particular declaration, or that a certain property should hold for a name collection (e.g. the declarations belonging to a particular scope should be unique). Typing constraints express the requirements for type consistency (e.g. type equality).

Resolution constraints are checked against the scope graph using a resolution calculus. The resolution calculus defines via which path through the scope graph a reference may be resolved to a declaration, the path being a sequence of traversed edges. In NaBL2, the resolution calculus is parameterized with a global set of edge labels, scope visibility order and path well-formedness predicate in the form of a regular expression. This way the resolution calculus can be tailored to the object language. However, the global scope of this parameterization is a limitation, as it disallows the definition of namespace-dependent visibility policies.

## 2.4 Statix

Statix is a constraint-based declarative meta DSL for defining the static semantics of an object language similar to NaBL2 (Van Antwerpen, Bach Poulsen, et al. 2018). It was introduced to the Spoofax language workbench as the successor to the NaBL2 meta DSL. Statix is based on the same ideas as NaBL2 but introduces a few generalizations intended to enable capturing various complexities in type systems in a more generic manner. This evolution fits some long term research goals including the standardization of the way names are treated in programming languages based on common underlying concepts, and constructing reusable tools for language design.

One of the generalizations that Statix makes compared to NaBL2 is that it allows for user defined constraints. In NaBL2, constraint generation rules are written. They can be named, and passed to higher-order rules, but are limited in their possible rule patterns and applications. In Statix, generic constraints can be defined that are named and can be referenced in the bodies of other constraints. This allows for the creation of abstract, reusable constraints within a specification.

Another generalization of Statix is that it allows for name resolution through the use of scope graph queries that are parameterized on a per constraint basis rather than through a global resolution calculus as is the case in NaBL2. A query as part of a constraint body is parameterized in a target relation (i.e. collection that declarations in a scopegraph belong to), path welformedness predicate (i.e. a regular expression in terms of scope graph edge labels), data wellformedness filter, label ordering relation (e.g. to express the shadowing policy), data comparison predicate and a result pattern. This addresses the limitation mentioned in section 2.3, allowing for the definition of name visibility policies that are namespace-dependent.

Finally, scope graph declarations in Statix are generalized to datums belonging to user defined relations, the signatures of which are allowed to include scopes in addition to other term constructors. This allows for what is dubbed the "scopes-as-types" approach to modeling type system features. The approach is well suited for modeling non-trivial type system features, in particular structural record types and parameterized types in both nominal and structural type systems. The support for this approach renders the NaBL2 feature of scope graph imports obsolete, which was therefore left out of the design of Statix.

These design choices ultimately require some differences in one's approach to model certain semantics when compared to NaBL2, because the feature sets of the DSLs differ such that there is no one-to-one correspondence for all features. Particular differences are further elaborated upon in chapter 3.

This concludes our introduction to the main topics relevant in this thesis, which include the IceDust object language and the NaBL2 and Statix meta-languages for static semantics specification. In the next chapter, we describe how these topics were involved in the work that we have done, as we explore the insights we gained over the course of migrating the static semantics of IceDust from NaBL2 to Statix.

# Chapter 3

# Design differences between NaBL2 and Statix

This thesis revolves around a case study of the Statix meta-language. It involved defining the static semantics of the IceDust DSL in Statix. A prior definition of the DSL's static semantics existed in the form of an NaBL2 specification. Therefore, our approach involved migrating the existing specification. During the migration process we were able to analyse differences between the prior and novel specification and distill a number of key findings that contributed toward answering our research questions. The findings are presented in this chapter, categorized by four topics: defining type compatibility relations, defining name binding patterns, storing declaration properties in the scope graph and defining interfaces between compiler pipeline stages.

## 3.1 Type compatibility relations

In this section we look at the way the types featured in the IceDust language are related to each other. Modeling relations between types expressively using a meta DSL requires support of appropriate metalanguage constructs. Both NaBL2 and Statix support modeling relations over sets of terms (i.e. the data structure used to represent types) using different feature sets. First, in Section 3.1.1, we explain how type compatibility relations can be defined using NaBL2 and Statix in general. Then, in Section 3.1.2, we look at examples of how we expressed particular type compatibilities of IceDust in Statix and compare it to the NaBL2 counterpart in terms of implementation effort and readability.

### 3.1.1 Defining type compatibility relations using NaBL2 and Statix

We first introduce type compatibilities and their meaning for static semantics formalization, and then explain the features of NaBL2 and Statix that can be utilized to formalize such compatibilities.

#### Type compatibility in static semantics

A static semantics specification includes a definition of type compatibility. Type compatibility in the broad sense refers to the similarity of two types to each other. Defining compatibility is important for ensuring soundness of, among others, type conversions and operations. The particular rules governing type compatibility may to some degree be specific to these conversions and operations. Compatibility is also in part inherent to the category of a type system. For example, nominal and structural type systems give a different meaning to subtyping, which plays a role in type compatibility.

In formalizing static semantics, type compatibility may be defined using the relation as mathematical construct. Consider some examples of type conversions in figure 3.1. The example shows a language supporting implicit type conversions. Line 2 shows a conversion of a value of type `Time` to type `Int`, based on the idea that a datetime value may be represented as a unix timestamp. Line 3 shows a mixed-type expression that implies a promotion of a value of type `Int` to type `Float`.

```
1 Time time = 1970-01-01 00:00:21;
2 Int timestamp = time;
3 Float answer = timestamp * 2.0;
```

Figure 3.1: Example program with type conversions.

$$
\begin{aligned}
T =& \{\texttt{Time}, \texttt{Int}, \texttt{Float}\} \\
R_c =& \{(\texttt{Time}, \texttt{Int}), (\texttt{Int}, \texttt{Float}) \\
& (\texttt{Time}, \texttt{Time}), (\texttt{Int}, \texttt{Int}), (\texttt{Float}, \texttt{Float}), \\
& (\texttt{Time}, \texttt{Float})\}
\end{aligned}
$$

Figure 3.2: Type compatibility relation over a set of primitive types.

The compatibility between the primitive types according to this example may be represented by the relation defined in figure 3.2. $R_c$ is defined as a binary relation over a set of types $T$, meaning $R_c \subseteq T \times T$. We define the relation to be reflexive, transitive and asymmetric, so the relation includes tuples for reflexively and transitively related types as well.

Type checking the assignment statements in the program now involves testing whether the expression type can be converted to the statically declared type. Given expression type $t_e$ and static type $t_s$, this then amounts to asserting $(t_e, t_s) \in R_c$. Type checking a binary operation (such as in line 3 of the example) with left and right operands of types $t_l$ and $t_r$ respectively then amounts to asserting $\exists t_x \in T[(t_l, t_x) \in R_c \wedge (t_r, t_x) \in R_c]$. This demonstrates that a definition of type compatibility is a prerequisite for defining the static semantics of a language.

With the practicalities involved in implementing these relations in mind, we distinguish between *static* and *dynamic* relations in this thesis. With static relations, we refer to relations that are invariant, i.e. independent of the program. For example, such a relation could express which built-in type can be cast to which other type, like in the example above. By dynamic relations, we mean relations that cannot be established until constraint time, i.e. during the analyis on an object program. For example this could be a relation capturing the inheritance relationships between user-defined classes.

#### Defining relations in NaBL2

NaBL2 features a dedicated "relation" language construct. In a separate *relations* section of a module, named relations with a term signature can be declared. Using declarative modifiers, certain properties of the relation can be defined. For example, a type compatibility relation for subtyping may be declared as follows:

```
reflexive, anti-symmetric, transitive sub : Type * Type
```

Figure 3.3: Example of a relation declaration in NaBL2.

Pairs of terms can then be added to this relation explicitly using the `term <sub! term` clause as part of a constraint generation rule, like such:

```
init ^ () :=
  TInt() <sub! TNumber().
```

Figure 3.4: Example of declaring a relation member in a constraint generation rule in NaBL2.

It can subsequently be asserted that a pair of terms belongs to this relation using a `term <sub? term` clause as part of a constraint generation rule. For example, this could be involved in testing whether the type of the body of a function declaration matches its expected return type:

```
[[ Function(id, t, exp) ^ (s) ]] :=
  [[ t ^ (s) : expected_type ]],
  [[ exp ^ (s) : body_type ]],
  body_type <sub? expected_type.
```

Figure 3.5: Example of quering a relation in a constraint generation rule in NaBL2.

The supported "positive" properties that can be declared on a relation allow inference of pairs that are not explicitly added. Let $R$ be a relation on a set of terms $T$. The positive relation properties available in NaBL2 express the following inferences:

- `reflexive` — $\forall x \in T : xRx$
- `symmetric` — $\forall x, y \in T : xRy \Leftrightarrow yRx$
- `transitive` — $\forall x, y, z \in T : xRy \wedge yRz \Rightarrow xRz$

The supported "negative" properties yield validations that are performed whenever a pair is added to the relation by a constraint clause. When an added pair causes a violation of the property, this results in an error in the program. These properties mutually exclude their positive counterpart and include:

- `irreflexive` — $\forall x, y \in T : xRy \Rightarrow x \neq y$
- `anti-symmetric` — $\forall x, y \in T : \neg(xRy \wedge yRx)$
- `anti-transitive` — $\forall x, y, z \in T : xRy \wedge yRz \Rightarrow \neg xRz$

In addition to these properties, relations feature automatically derived functions that yield the least upper bound (`lub`) and a greatest lower bound (`glb`) for a given pair of terms. For example, the resulting type of an operation may be the least upper bound of the types of the operands given the subtype relation.

```
[[ BinOp(e1, e2) ^ (s) : type ]] :=
  [[ e1 ^ (s) : e1_type ]],
  [[ e2 ^ (s) : e2_type ]],
  type is sub.lub of (e1_type, e2_type).
```

Figure 3.6: Example of applying the automatically derived least upper bound function of a relation in a constraint generation rule in NaBL2.

Apart from *relations*, NaBL2 also features a section for declaring *functions*. Functions allow for the definition of a mapping from one term to another, using generic term patterns. Functions can be used to encode further derivations of relations between terms in addition to `lub` and `glb`.

### Defining relations in Statix

In Statix, aforementioned relations and their derivations cannot be expressed using dedicated language constructs, but are rather to be defined using constraints. These constraints can be declared as binary predicates in the form of axiom rules. Pairs of terms can then be "added" to the relation by declaring a rule with the terms in the rule head, like so:

```
sub : Type * Type
sub(TInt(), TNumber()).
```

Figure 3.7: Example of declaring a member of a relation using an axiom rule in Statix.

Although there are no declarative modifiers available for constraints that ensure that certain properties of the relation hold such as in NaBL2, the positive properties `reflexive` and `symmetric` can be trivially expressed using specific rule patterns as depicted in 3.8. However, transitivity is less trivial. One might make an attempt by declaring a rule such as shown in Figure 3.8.

```
// reflexive
sub(X, X).
// symmetric
sub(X, Y) :- sub(Y, X).
// transitive
sub(X, Z) :- {Y} sub(X, Y), sub(Y, Z).
```

Figure 3.8: Example of declaring properties of a relation using constraint patterns in Statix.

Unfortunately, this does not have the intended result, because the unification algorithm of Statix will ultimately not be able to find a solution for pairs that are transitively related via more than one intermediate element using this rule. Also note that the head of this rule is equally specific as the rule for symmetry. Due to the committed choice in rule selection, this means a relation cannot be both symmetric and transitive at the same time using this approach.

The most trivial solution to this problem is by enumerating all transitively related pairs and explicitly declaring axiom rules for those as well, in addition to the directly related pairs.

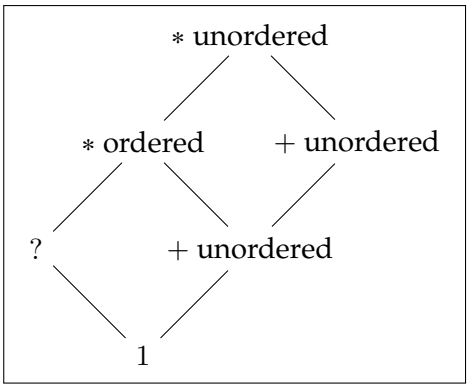



Figure 3.9: The lattice modeling the compatibilities between multiplicity and ordering combinations in IceDust.

As a drawback, this can quickly bloat the specification, and allows for mistakes (e.g. overlooking a particular pair).

As for the negative properties, there is no similar way in Statix of preventing the addition of pairs that violate such a property.

When it comes to derivations, Statix does not offer an abstraction for finding the least upper bound and greatest lower bound of a relation similar to NaBL2. These functions are to be explicitly defined as constraints, like other derivations. In Statix, the equivalent to NaBL2 functions would be constraints in the form of functional rules.

In the next section, we will discuss how these differences impacted our approach to modeling the type compatibilities of the IceDust language.

### 3.1.2 Modeling the type compatibilities of IceDust

Taking into account the the ways in which NaBL2 and Statix allow for defining binary relations over sets of types, in this section we give some examples of how we defined certain type compatibilities of the IceDust language in Statix and compare these with the respective solution in NaBL2.

#### Static type compatibility relations of IceDust

We look at an example of how one of the type lattices of IceDust can be declared in NaBL2 and Statix. Consider the lattice depicted in figure 3.9. A lattice is a partially ordered set in which each pair of elements has a unique least upper bound. The relation on such a set is a partial order, which can be described as transitive and anti-symmetric. In this case, the partial order is "non-strict", meaning it is reflexive. We show the implementation of this partial order — i.e. the declaration of the pairs of terms that belong to the relation as well as the least upper bound function — using NaBL2 and Statix in figures 3.10 and 3.11 respectively. The examples bring to light a notable difference in expressiveness between the metalanguages. In NaBL2, only the directly related pairs of terms need to be declared, whereas in Statix more enumeration is required. This includes the transitively related pairs and the least upper bound of every possible combination of terms. However, the symmetry rule of the `multLub` constraint allows restricting the enumeration to all combinations instead of permutations, while the reflexive rule allows exluding all identity cases. Note that the symmetric rule can be used here because there exists a least upper bound for all possible combinations of multiplicities. Otherwise, the constraint might fail which would yield an infinite loop due to the recursion.

```
signature
  relations
    reflexive, anti-symmetric, transitive mulOrd : Mul * Mul
rules
  init ^ () :=
    TOne()                <mulOrd! TZeroOrOne(),
    TOne()                <mulOrd! TOneOrMoreOrdered(),
    TZeroOrOne()          <mulOrd! TZeroOrMoreOrdered(),
    TOneOrMoreOrdered()   <mulOrd! TZeroOrMoreOrdered(),
    TOneOrMoreOrdered()   <mulOrd! TOneOrMore(),
    TZeroOrMoreOrdered()  <mulOrd! TZeroOrMore(),
    TOneOrMore()          <mulOrd! TZeroOrMore().
```

Figure 3.10: Implementation of the multiplicity and ordering lattice of IceDust in NaBL2

```
rules
  mult : MULT * MULT

  mult(ONE(), ZERO_ONE()).
  mult(ONE(), ONE_MORE_ORD()).
  mult(ZERO_ONE(), ZERO_MORE_ORD()).
  mult(ONE_MORE_ORD(), ZERO_MORE_ORD()).
  mult(ONE_MORE_ORD(), ONE_MORE()).
  mult(ZERO_MORE_ORD(), ZERO_MORE()).
  mult(ONE_MORE(), ZERO_MORE()).

  // transitive
  mult(ONE(), ZERO_MORE_ORD()).
  mult(ONE(), ONE_MORE()).
  mult(ONE(), ZERO_MORE()).
  mult(ZERO_ONE(), ZERO_MORE()).
  mult(ONE_MORE_ORD(), ZERO_MORE()).

  // reflexive
  mult(M, M).
```

(a) Declaration of the multiplicity relation.

```
rules
  multLub : MULT * MULT -> MULT

  multLub(X, X) = X.
  multLub(X, Y) = multLub(Y, X).

  multLub(ZERO_MORE(), _) = ZERO_MORE().
  multLub(ZERO_MORE_ORD(), ONE_MORE()) =
    ZERO_MORE().
  multLub(ZERO_MORE_ORD(), ZERO_ONE()) =
    ZERO_MORE_ORD().
  multLub(ZERO_MORE_ORD(), ONE_MORE_ORD()) =
    ZERO_MORE_ORD().
  multLub(ZERO_MORE_ORD(), ONE()) =
    ZERO_MORE_ORD().
  multLub(ONE_MORE(), ZERO_ONE()) =
    ONE_MORE().
  multLub(ONE_MORE(), ONE_MORE_ORD()) =
    ONE_MORE().
  multLub(ONE_MORE(), ONE()) =
    ONE_MORE().
  multLub(ONE(), ZERO_ONE()) =
    ZERO_ONE().
  multLub(ONE(), ONE_MORE_ORD()) =
    ONE_MORE_ORD().
```

(b) Declaration of the least upper bound (lub) function on the multiplicity relation

Figure 3.11: Implementation of the multiplicity and ordering lattice of IceDust in Statix

### Dynamic type compatibility relations of IceDust

Some relations apply to terms that represent declarations in the program such as custom types (e.g. classes), and are therefore established at constraint time. In NaBL2, dynamic relations can also be modelled equivalently to static relations using the *relations* feature, as pairs can be added to a relation using a $term$ `"<"`$relation - id$`"!"` $term$ clause that may be included in any constraint generation rule. In Statix, however, the mutable data structures during constraint time are limited to the AST node properties and the scope graph. Because declarations in the program are already represented in the scope graph for name binding purposes, it makes sense to reuse this structure for modeling relations such as type compatibilities.

```
signature
  sorts
    TYPE

  constructors
    INT : TYPE
    STRING : TYPE
    FLOAT : TYPE
    BOOLEAN : TYPE
    DATETIME : TYPE
    NOVALUE : TYPE
    ENTITY : scope -> TYPE
```

Figure 3.12: The type signatures of IceDust in Statix.

In IceDust, entity declarations introduce new types. These were modeled in Statix using the "scopes-as-types" approach, as displayed in figure 3.12. The scope encapsulated in the type term represents the lexical scope of the entity declaration, containing its member declarations. When resolving the type of references to the entity elsewhere in the program, a reference to the entity scope is obtained and subsequently allows resolution of references to members of the entity. This removes the need for import edges, as applied in the NaBL2 implementation.

When entities are declared to inherit from one another, this is modeled by an edge (labelled `INHERIT`) in the scope graph. Although these edges, like the rest of the scope graph, are primarily used for name binding purposes, they can also be considered to represent the subtype compatiblity relation between these types.

For example, consider the example program with inheriting entities in figure 3.13a. The corresponding scope graph in figure 3.13b shows the edges between the scopes corresponding to the entity declarations.

Adding pairs to the relation is implicitly done by declaring scope graph edges. Testing the relation and computing derivations such as the least upper bound can now be encoded using scope graph queries.

The implementation for testing subtype compatibility between entity types is depicted in figure 3.14. The `substypeEntity` predicate includes a query that establishes whether there is a path in the scope graph via `INHERIT` labelled edges from the scope of the subtype to the scope of the supertype.

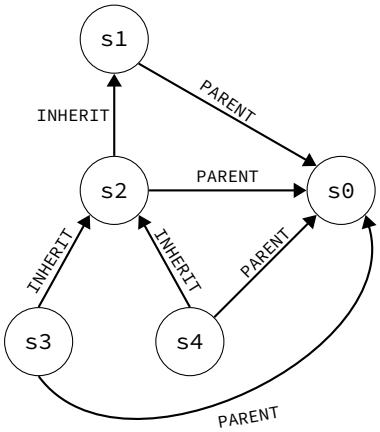
The implementation for the least upper bound function on the subtype compatibility relation between entity types is a bit more involved. The Statix implementation is shown in figure 3.15. This approach works because of the restrictions imposed on inheritance relation-

```
module example s0
model
  entity Ancestor s1 {}

  entity Parent s2 extends Ancestor {}

  entity SubOne s3 extends Parent {}
  entity SubTwo s4 extends Parent {}
```

(a) IceDust example with inheritance




(b) Corresponding scope graph

Figure 3.13: Example of entity declarations composing an inheritance hierarchy in IceDust, and the corresponding scope graph.

```
rules
  subtype : TYPE * TYPE

  subtype(T, T).
  subtype(NOVALUE(), _).
  subtype(T1@ENTITY(_), T2@ENTITY(_)) :- subtypeEntity(T1, T2).

  subtypeEntity: TYPE * TYPE
  subtypeEntity(ENTITY(x), ENTITY(x)).
  subtypeEntity(ENTITY(s_entity), ENTITY(s_super)) :- {results}
    query ()
      filter INHERIT+ and true
      min and true
      in s_entity |-> results,
    includesScope(results, s_super).

  includesScope : list((path * scope)) * scope
  includesScope([(_, x) | _], x).
  includesScope([_ | xs], x) :- includesScope(xs, x).
```

Figure 3.14: Implementation of the subtype compatibility relation including (dynamic) entity types in Statix.

```
lubtype : TYPE * TYPE -> TYPE
lubtype(X, X) = X.
lubtype(NOVALUE(), INT()) = INT().
lubtype(NOVALUE(), FLOAT()) = FLOAT().
lubtype(NOVALUE(), STRING()) = STRING().
lubtype(NOVALUE(), BOOLEAN()) = BOOLEAN().
lubtype(NOVALUE(), DATETIME()) = DATETIME().
lubtype(NOVALUE(), ENTITY(X)) = ENTITY(X).
lubtype(INT(), NOVALUE()) = INT().
lubtype(FLOAT(), NOVALUE()) = FLOAT().
lubtype(STRING(), NOVALUE()) = STRING().
lubtype(BOOLEAN(), NOVALUE()) = BOOLEAN().
lubtype(DATETIME(), NOVALUE()) = DATETIME().
lubtype(ENTITY(X), NOVALUE()) = ENTITY(X).
lubtype(ENTITY(X), ENTITY(X)) = ENTITY(X).
lubtype(T1@ENTITY(s1), T2@ENTITY(s2)) = ENTITY(s3) :-
  {scopes1 scopes2 shared_scopes}
  scopes1 == superScopes(s1),
  scopes2 == superScopes(s2),
  shared_scopes == intersection(scopes1, scopes2),
  [s3 | _] == shared_scopes.
superScopes: scope -> list((path * scope))
superScopes(s) = results :-
  query ()
    filter INHERIT+ and true
    min and true
    in s |-> results.
intersection: list((path * scope)) * list((path * scope)) -> list(scope)
intersection([(_, x) | xtail], [(_, x) | ytail]) =
  [x | intersection(xtail, ytail)].
intersection([(_, x)], [(_, x) | tail]) = [x].
intersection([(_, x) | tail], [(_, x)]) = [x].
intersection([(_, x)], [(_, x)]) = [x].
intersection([(_, x) | tail], ys) =
  concat(intersection([(_, x)], ys), intersection(tail, ys)).
intersection([(_, x)], [_ | tail]) =
  intersection([(_, x)], tail).
intersection([(_, x) | tail], []) = [].
intersection([(_, x)], []) = [].
intersection([], [(_, x) | tail]) = [].
intersection([], [(_, x)]) = [].
intersection([(_, x)], [(_, y)]) = [].
intersection(_, []) = [].
intersection([], _) = [].
intersection([], []) = [].
concat: list(scope) * list(scope) -> list(scope)
concat([x | xs], ls) = [x | concat(xs, ls)].
concat(ls, []) = ls.
concat([], ls) = ls.
```

Figure 3.15: Implementation of the least upper bound function on the subtype relation including (dynamic) entity types in Statix.

ships that may be declared, ensuring that inheritance hierarchies form directed rooted trees (in-trees) in the scope graph. If the subtype relations between entities were allowed to be less strict, such as the lattices of built-in types, then this approach would no longer be correct. Given two nodes in the tree (i.e. entity types), the algorithm (1) finds the ordered list of ancestors of both nodes; (2) finds the intersection between these lists; and (3) takes the first element from the intersection. This yields the lowest common ancestor, in this case equivalent to the least upper bound of the subtype relation. From the extensiveness of the Statix implementation it becomes apparent that it is non-trivial to translate this algorithm from the imperative paradigm to the constraint logic paradigm of Statix. Encoding this relation in Statix therefore requires more effort than encoding it in NaBL2.

## 3.2 Name binding patterns

In addition to simple lexical scoping, where the visibility of names declared in a program is solely determined by their position in the source text, many programming languages support more complicated patterns of name binding where declarations of a scope may selectively be made available elsewhere in the program. In Section 3.2.1 we explain how NaBL2 and Statix support definition of such name binding patterns in general. Then, in Section 3.2.2 we describe how we applied this for the purpose of defining particular name binding cases of IceDust in Statix, and compare it to the prior NaBL2 solution.

### 3.2.1 Defining name binding patterns with imports using NaBL2 and Statix

Statix generalizes NaBL2 in various aspects. One of the consequences is that the scope graph formalism that is being employed by these meta-languages also differs.

Originally, the scope graph framework as built upon by NaBL2 included imports (Neron et al. 2015). Using imports, non-lexical binding patterns can be expressed. By declaring an import edge from a declaration $a$ to a particular scope $S_a$, the declarations visible in that scope can be exported via the declaration. In a scope $S_x$ elsewhere in the scope graph, the exported declarations can then be imported by adding an import reference $a$ in scope $S_x$. Given that declaration $a$ is reachable from scope $S_x$, this implies a that all declarations reachable from $S_a$ are now also reachable from $S_x$. Due to imports, the definition of reachability thus has a recursive nature.

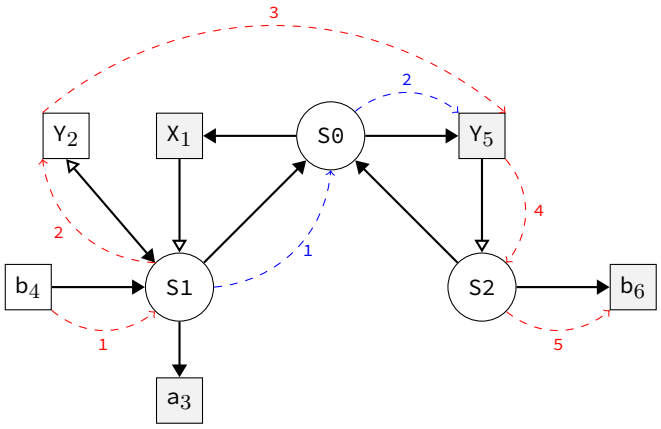
Consider for example an object language featuring modules that can import from one another. An example program in such a language along with its scope graph using imports is depicted in figure 3.16. The scope graph is constructed according to the accompanying NaBL2 specification. The way the scope graph is constructed can be explained by the steps outlined in the figure.

As explained in Section 2.4, the adapted scope graph framework that Statix is built upon no longer features dedicated import primitives, because imports can be encoded using the scopes-as-types approach. To demonstrate this, the scope graph using the Statix framework for the exact same example program as before is depicted in figure 3.17. Again, the scope graph construction can be explained by an imperative analogy as outlined in the figure.

The NaBL2 and Statix approaches to modeling the name binding in this example program shows some notable differences.

```
1  S0
2  module X1 { S1
3    import Y2
4    member a3 = b4
5  }
6  module Y5 { S2
7    member b6 = 21
8  }
```

(a) Example program with imports.




(b) Corresponding scope graph. The red edges compose the reachability path between reference $b_4$ and declaration $b_6$. The blue edges compose the reachability path between import reference $Y_2$ and import declaration $Y_5$, which is assumed by the red reachability path.

```
3  signature
4    // ...
5    name resolution
6      // ...
7      well-formedness
8        P* I?
9  rules
10   [[ Program(modules) ^ (s0) ]] :=
11     Map1 [[ modules ^ (s0) ]].
12   [[ Module(id, statements) ^ (s0) ]] :=
13     new s_mod,
14     s_mod -P-> s0,
15     Modules{id} <- s0,
16     Modules{id} =I=> s_mod,
17     Map1 [[ statements ^ (s_mod) ]].
18
19   [[ Import(id) ^ (s_mod) ]] :=
20     Modules{id} <=I= s_mod.
21   [[ Member(id, value) ^ (s_mod) ]] :=
22     Members{id} <- s_mod,
23     [[ value ^ (s_mod) ]].
24
25   [[ Literal(_) ^ (_) ]] := true.
26   [[ Reference(id) ^ (s_mod) ]] :=
27     Members{id} -> s_mod,
28     Members{id} |-> _.
```

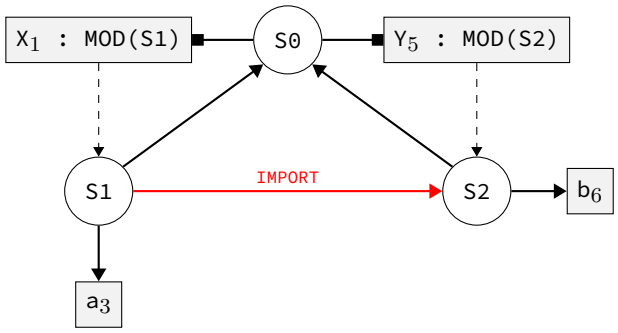
(c) NaBL2 specification (spec) governing the construction of the above scope graph. For brevity, the signatures (i.e. sorts, constructors and scope graph edge labels and label order) have been left out of this snippet, except for the global well-formedness constraint such that it may be compared to its Statix counterpart encoded in query constraints.

1. The constraint on spec line 10 is applied to the root AST node (`Program`), and is passed the initial scope `s0`.
2. In accordance with the constraint on spec line 12, the module declarations in the program on lines 2 and 6 result in the creation of scopes `s1` and `s2` respectively (spec line 13). These scopes are associated with scope `s0` via a parent edge (spec line 14). The respective declarations $X_1$ and $Y_5$ are added to the scope graph (spec line 15) and are associated with scopes `s1` and `s2` via an import edge (spec line 16).
3. The member declarations on lines 4 and 7 result in declarations $a_3$ and $b_6$ as per the constraint on spec line 21.
4. The import statement in the program on line 3 will result in the addition of import reference $Y_2$ to the scope graph according to the constraint on spec line 19.
5. Finally, the reference $b_4$ on program line 4 is then added to the scope graph as per the constraint on spec line 26, and a resolution constraint is added yielding the resolution of reference $b_4$ to declaration $Y_5$.

Figure 3.16: Example of an NaBL2 scope graph including imports. The scope graph represents the name binding for a program expressed in an example language featuring modules.

```
1  S0
2  module X1 { S1
3    import Y2
4    member a3 = b4
5  }
6  module Y5 { S2
7    member b6 = 21
8  }
```

(a) Example program with imports.




(b) Corresponding scope graph utilizing the scope graph framework of Statix.

```
2  rules
3    programOk : Program
4    programOk(Program(modules)) :- {s0}
5      new s0,
6      modulesOk(s0, modules).
7
8    modulesOk maps moduleOk(*, list(*))
9    moduleOk : scope * Module
10   moduleOk(s0, Module(id, statements)) :-
11     {s_mod}
12     new s_mod,
13     s_mod -PARENT-> s0,
14     !modules[id, MOD(s_mod)] in s0,
15     statementsOk(s_mod, statements).
16
17   statementsOk maps statementOk(*, list(*))
18   statementOk : scope * Statement
19   statementOk(s, Import(id)) :- {s_mod}
20     query modules
21       filter PARENT* and { x' :- x' == id }
22       min $ < PARENT and true
23       in s |-> [(_, (_, MOD(s_mod)))],
24     s -IMPORT-> s_mod.
25   statementOk(s, Member(id, value)) :-
26     !members[id] in s,
27     valueOk(s, value).
28
29   valueOk : scope * Value
30   valueOk(_, Literal(_)).
31   valueOk(s, Reference(id)) :-
32     query members
33       filter IMPORT* and { x' :- x' == id }
34       min $ < IMPORT and true
35       in s |-> [(_, _)].
```

(c) Statix specification (spec) governing the construction of the above scope graph. For brevity, the signatures (i.e. sorts, constructors, scope graph edge labels and relations) have been left out of this snippet.

1. The initial constraint on spec line 4 creates scope `S0` and passes it to the `modulesOk` constraint.
2. The declarations of the modules on program lines 2 and 6 will then result in the creation of the $X_1$ and $Y_5$ scope graph datums respectively, along with their associated scopes `S1` and `S2`, as per spec line 10 (`moduleOk`).
3. The member declarations on program lines 4 and 7 yield the scope graph datums $a_3$ and $b_6$ respectively as per specification line 25 (`statementOk`).
4. The import statement on program line 3 will, given identifier $Y_2$, query the scope graph datum $Y_5$ as per specification line 19 (`statementOk`). An `IMPORT`-labelled edge is then added to the scope graph from `S1` to `S2`, the latter being referenced from the type in the queried scope graph datum.
5. Finally, the reference $b_4$ on program line 4 can then be resolved to declaration $b_6$ via the `IMPORT`-labelled edge as per specification line 31 (`valueOk`).

Figure 3.17: Example of a Statix scope graph encoding imports using the scopes-as-types approach.

On one hand, the Statix scope graph features fewer elements, as it misses specialized import references and declarations. On the other hand, the rules section of the Statix specification consists of more non-empty lines (30) than the one in the NaBL2 specification (17). In part, this may be explained by the fact that used-defined Statix constraints require a separate type signature to be declared, which enables certain static guarantees. Another reason for this difference lies in the resolution constraints. In NaBL2, the resolution algorithm is parameterized globally by a few lines in the *signature* section. All introduced resolution constraints in the *rules* section then require just a single line, as the global configuration applies to all of them. In Statix, the algorithm is configured on a per-query basis. This requires inclusion of more lengthy query constraints for each name resolution. This way, Statix circumvents the effort of having to tailor the global parameterization to be applicable to all name resolutions involved in the specification at the same time, at the cost of having to spend more effort on defining each resolution.

As a general takeaway we observe that Statix makes a trade-off in the area of scope graph manipulation when compared to NaBL2. Statix has the benefit that it simplifies the construction of the scope graph, which is restricted to fewer types of elements. It also offers more fine grained control over how the scope graph is queried, as queries can be customized on a per constraint basis. However, this does have the drawback that for trivial resolutions, Statix requires more boilerplate query constraint definitions compared to NaBL2, which may be considered less readable. In the next subsection, we will look at further implications that these differences have on modeling particular IceDust language features.

In the next section, we look at how these ways of defining imports are applied in the formalization of IceDust.

### 3.2.2 Record extension patterns of IceDust

The entity primitive of the IceDust language can be described using a common idea in type theory referred to as a record. A record can be regarded as a collection of fields, which associate a name with a value. The types of such records can be modelled respectively using record types, which are a collection of associations between a name and a type.

Declarations of a record type in a program need not be final. Type systems may support patterns of so called record extension, where some types are derived by composing other record types in a particular way. This may involve for example merging the fields of two record types, and having fields of one record shadow the fields of another. Record extension can be modelled using scope graphs in a way that retains the original composition.

Record extension is one of the more complex patterns involved in the type system of the IceDust language. It is involved in two IceDust features: inheritance and relations. We analyze these features and compare their implementations in NaBL2 and Statix.

#### Entity inheritance in IceDust

The IceDust language features inheritance between entities. This means that an entity can be declared as a subtype of another entity, and thus inherit the fields, including both attributes and relations, of the other entity. This is modelled using record extension, where the fields of the subtype are merged with the fields of the supertype, and the fields of the subtype shadow the fields of the supertype. An example of a declaration of an entity inheriting from another entity is displayed in Figure 3.19.

The corresponding scope graph as per the NaBL2 specification is depicted in Figure 3.20. Due to the globally parameterized resolution calculus, in combination with the recursive nature of resolving import references, a specialized scope graph construction is required to define the inheritance. For each entity, three scopes are constructed. A root scope, a scope for local declarations, and a scope importing declarations of the parent entity. In the pa-

```
modelOk(s, Entity(id, optional_parent, _, members)) :-
  { s_model s_resolve rtype }
  // ... (abbreviated)
  extendScopes(resolveEntityExtensions(s, id), s_model).

modelOk(s, Relation(e1, r1, m1_optional, m2_optional, e2, r2)) :-
  { s_e1 s_e2 s_r1 s_r2 m1 m2 id1 id2 s_p1 s_p2 }
  // ... (abbreviated)
  new s_r1,
  s_r1 -EXTEND-> s_e1,
  s_r1 -P-> s,
  new s_r2,
  s_r2 -EXTEND-> s_e2,
  s_r2 -P-> s,
  !entity_extension[e1, s_r1] in s,
  !entity_extension[e2, s_r2] in s.

extendScopes maps extendScope(list(*), *)
extendScope : (path * (string * scope)) * scope
extendScope((_, (_, s)), s). // extension is not reflexive
extendScope((_, (_, s')), s) :- s -EXTEND-> s'.

resolveEntityExtensions : scope * ID -> list((path * (string * scope)))
resolveEntityExtensions(s, id) = ps :-
  query entity_extension
    filter PARENT* and { x' :- x' == (id, _) }
      min and true
      in s |-> ps.
```

Figure 3.18: The Statix definition of constraints related to entity extension. The `extendScope` constraint introduces edges in the scope graph that ensure that the scopes of an entity and its extensions form a clique. The `resolveEntityExtensions` constraint queries all extension scopes of an entity via scopes encapsulted in declarations. The `modelOk` constraints for entity and relation declarations that rely on these constraints are listed for context.

rameterization of the resolution calculus, two distinct edge labels are defined with different priorities. The edge between the root scope and the local scope has a higher priority label than the edge between the root scope and the parent scope. This means that fields declared within the lexical scope of the entity declaration shadow any inherited fields.

The counterpart scope graph as per the Statix specification is depicted in Figure 3.21. What stands out is that this scope graph is more concise, as it only features one scope per entity, and direct edges between those scopes to express inheritance relationships. The reason that Statix allows for this more concise scope graph is that it allows for more flexibility both constructing scope graph, i.e. using the scopes-as-types approach, as well as querying it, i.e. using per query resolution parameters. This eases defining a suitable scope graph construction, reasoning about it and verifying its correctness, which appears to be a clear benefit of Statix.

```
module example s0
model
  entity Number1 s1, s2, s3 {}
  entity Integer2 extends Number3 s4, s5, s6 {}
```

Figure 3.19: IceDust example with inheritance.

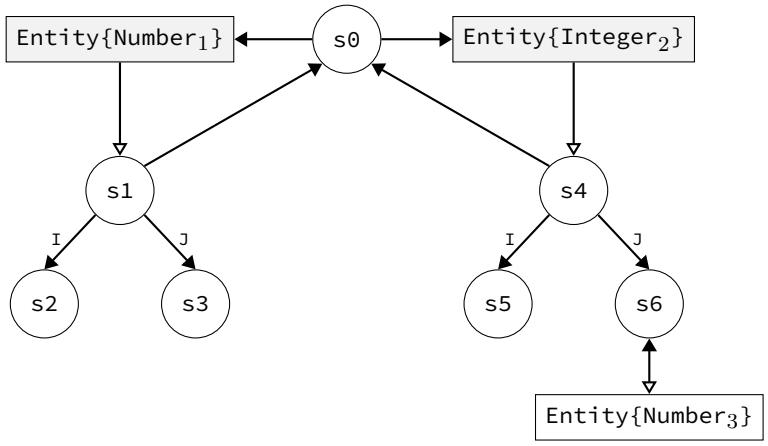



Figure 3.20: Corresponding NaBL2 scope graph. The edge label `I` represents an import, while label `J` represents a lower priority import.

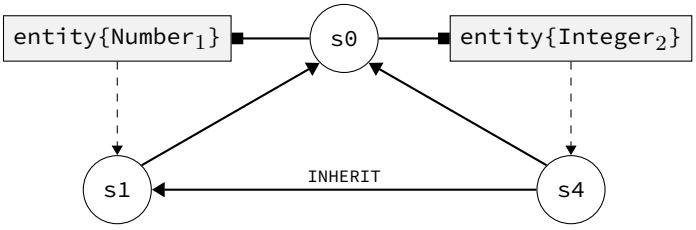



Figure 3.21: Corresponding Statix scope graph.

### Relations in IceDust

The IceDust language features relations between entities as first class citizens, which means they can be declared at module level using a dedicated language primitive, and can be a subject in other relations. In a relation declaration, role names are defined for navigating the relation in both directions between the involved entities. This effectively means that the entities involved in the relation are extended with fields having the according role names. This allows reference to the relation from other fields of the entity such as derived attributes and other relations. The fields of an entity are thus extended by a relation.

An example of a relation declaration in IceDust is depicted in Figure 3.22. The declaration implies that the `Summary` entity is extended with a field `numbers`, while the `Number` entity is extended with a field `summary`. Using these fields, the relation can be navigated by referencing them in the declarations of other fields.

The corresponding scope graph resulting from the NaBL2 specification is depicted in Figure 3.23. A relation introduces two scopes, each representing an extension of one of the resptive entity. Fields that entities are extended with will shadow inherited fields. Therefore extension scopes are referenced using an import reference in the local scope of the entity.

```
module example s0
model
  entity Summary1 s1, s2, s3 {}
  entity Number2 s4, s5, s6 {}

  relation Summary3.numbers4 s7 * <-> 1 Number5.summary6 s8
```

Figure 3.22: IceDust example with a relation.

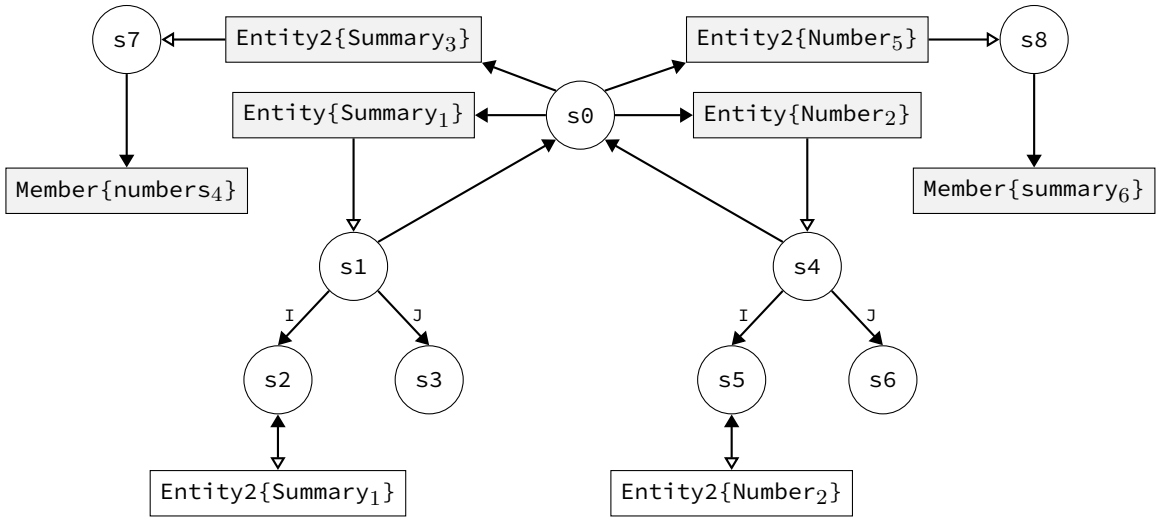



Figure 3.23: Corresponding NaBL2 scope graph. The edge label `I` represents an import, while label `J` represents a lower priority import.

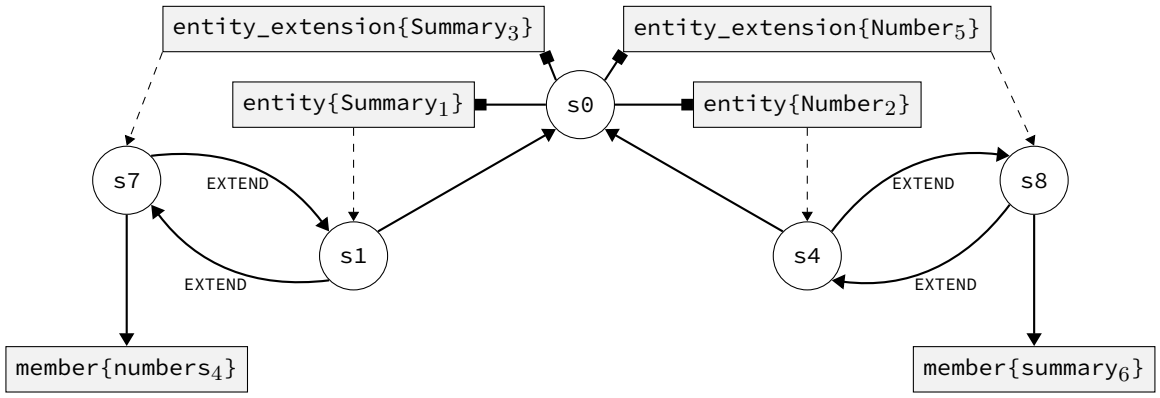



Figure 3.24: Corresponding Statix scope graph.

The extension scopes are accessible via an import declaration in the root scope. Using this construction, any field from an extension is reachable from the local scope of the entity.

The equivalent scope graph resulting from the Statix specification is shown in Figure 3.24. This scope graph similarly introduces two scopes for a relation, one for each involved entity, and ensures all fields from extensions are reachable from the entity scope. However, as an alternative to import references, using custom constraints that find all extending scopes for an entity via scopes-as-types, `EXTEND`-labelled edges are created to make sure that the

```
model
  entity Student{
  }
  entity Course{
    code : String
  }
  relation Enrollment{
    grade : Float

    Student.enrollment -> student
    Course.enrollment  -> course

    student.course <-> course.student
  }
data
  :Student {
    course = <{ grade = 8.0 }> { code = "IN1337" }
  }
```

Figure 3.25: Example of an IceDust program with data section that contains a declaration of a value for the shortcut role of an entity instance.

entity scope together with the extending scopes form a clique. The constraints governing this construction approach are listed in Figure 3.18. Then, the path wellformedness predicate of query constraints are defined such that at most one `EXTEND` edge may be traversed, preventing cycles.

One might argue that in the case of Statix, related scopes are closer together due to the cliques rather than imports, again making it slightly easier to reason about the scope graph, especially as they scale in size. However, the construction of the scope graph is less trivial, as exemplified by the custom constraints that were applied for this case, so slightly more implementation effort and ingenuity will be required in that area.

## 3.3 Declaration properties

In NaBL2, it is possible to assign arbitrary properties (key/value pairs) to a scope graph occurrence. In the formalization of IceDust, this was utilized to store additional contextual information about scope graph declarations that is not included in the declaration term itself. This information would then be used to guide certain constraint generation rules.

An example of this can be found when declaring the value for a shortcut role of an entity instance within the data section of an IceDust program. In Figure 3.25 we see an example of such a program. The program features a first class relation, that like described in the previous subsections, extends the involved entities with additional fields, namely the shortcuts defined on the relation. As per the example, entity `Student` will be extended with field `course` resolving to type `Course`, and vice versa. However, when now declaring an instance of one of the involved entities, in this case the `Student` entity, a value may be supplied for this shortcut role field. This value may not only consist of an instance of the related entity, but also an instance of the intermediate relation. As the name of the shortcut role field is defined to simply resolve to the entity, it is required to somehow also record in addition via which relation this role is defined, in order for the declared relation instance to be bound and type checked.

```
rules
[[ Shortcut(role1, shortcut1, role2, shortcut2) ^
  (entity_scope_local, entity_scope, module_scope) ]] :=
  // recording the type of the relation in which
  // these shortcut members were declared
  // as properties into the member declarations
  Member{shortcut1}.reltype := this_type,
  Member{shortcut2}.reltype := this_type,
  // ...
  true.

[[ MemberValue(member, vs) ^
  (entity_instance_scope, module_scope) ]] :=
  // reading the relation type from the property
  // of the resolved member declaration
  // and storing it under a declaration in
  // the implicit namespace
  Member{member} |-> member_def,
  member_def.reltype := relation_object_type,
  Implicit{"membervalue"}.reltype := relation_object_type,
  // ...
  true.

[[ RelationInstanceNoType(name, ms) ^
  (member_value_scope, module_scope) : relation_object_type ]] :=
  // reading the relation type from the declaration
  // in the implicit namespace and declaring it
  // as type of "this" within the scope of
  // the relation instance, such that members of
  // the relation may be resolved
  Implicit{"membervalue"} |-> member_value_def,
  member_value_def : member_value_def_type,
  member_value_def.reltype := relation_object_type,
  Implicit{"this"} : relation_object_type,
  // ...
  true.
```

Figure 3.26: Summary of the constraints involved to determine the type of a relation instance when supplied as value to a shortcut role, utilizing the NaBL2 support for assigning arbitrary properties to scope graph declarations. Remaining constraints that are part of the rules have been ommitted for brevity.

In NaBL2, this was solved by simply storing the relation type as a property on the shortcut role member declaration that the entity is extended with, as summarized in Figure 3.26. It shows that passing this additional contextual information through to the dependent constraints can be expressed rather concisely this way.

In Statix, there is no similar support for assigning additional arbitrary properties to a scope graph datum like in NaBL2.

The most trivial solution would be to include all the derived information in the datum itself. This would involve extending the constructor signature of the term used as datum with

the necessary arguments. However, this may quickly bloat the signature specifications, and disallow interaction with the datum in a sparse manner (e.g. in case some of the properties are optional).

A more involved solution would be to encode these properties - i.e. a pair of a key and a value - as scope graph datums themselves. In order to compose properties applying to the same declaration, they can be assigned to the same scope. This scope can then be referenced from the declaration datum using the scopes-as-types approach. This requires extending constructor signatures for terms that may have properties with only one additional argument.

For IceDust, we implemented this approach. The constraints providing an abstraction for this approach are displayed in figure 3.27. Because the set of properties for a declaration in some cases needed to be extended in different constraint rules than where the declaration was introduced, the property scopes need to be extensible just like the extension pattern for entities presented in Section 3.2.2. This can be seen in the `extendProps`, `resolvePropExtensions` and `extendPropScope` constraint definitions. Also, in order to deal with properties that are optional, to prevent constraint failures when reading the value of these properties, we need to wrap their values in a list. This creates additional complexity as can be seen from the `getPropOptionalType`, `getPropOptionalTypeFromList` and `getPropList` constraints.

It is interesting to note that although Statix allows for storing arbitrary properties associated with scope graph datums like this, it does require boilerplate definition of constraints manipulating the scope graph accordingly in non-trivial ways. Additionally, it requires extending the sorts and constraint rules for each new sort of term that may be stored as property value, rendering this solution to be highly language dependent and requiring it to be actively maintained throughout evolution of the specification. This clearly negatively impacts the conciseness of the Statix specification compared to the NaBL2 specification for IceDust.

```
initProps : scope * scope -> PROPS
initProps(s_context, s_props) = PROPS(s_props) :-
  extendPropScopes(resolvePropExtensions(s_context, s_props), s_props).

extendProps : scope * scope * scope
extendProps(s_context, s_props, s_extension) :-
  s_extension -EXTEND-> s_props,
  !prop_extension[s_props, s_extension] in s_context.

resolvePropExtensions : scope * scope -> list((path * (scope * scope)))
resolvePropExtensions(s, s_props) = ps :-
  query prop_extension
    filter PARENT* and { x' :- x' == (s_props, _) }
      min and true
      in s |-> ps.
extendPropScopes maps extendPropScope(list(*), *)
extendPropScope : (path * (scope * scope)) * scope
extendPropScope((_, (_, s)), s).
extendPropScope((_, (_, s')), s) :- s -EXTEND-> s'.

setProp: scope * string * string
setProp(s, key, value) :-
  !prop[key, PROPVAL_STRING(value)] in s.

setPropType: scope * string * TYPE
setPropType(s, key, type) :-
  !prop[key, PROPVAL_TYPE(type)] in s.

setPropOptionalType: scope * string * TYPE
setPropOptionalType(s, key, NOTYPE()).
setPropOptionalType(s, key, type) :- setPropType(s, key, type).

getProp: PROPS * string -> PROPVAL
getProp(PROPS(s), key) = propval :-
  query prop
    filter EXTEND? and { x' :- x' == key }
      min and true
      in s |-> [(_, (_, (propval)))].

getPropOptionalType : PROPS * string -> TYPE
getPropOptionalType(props, key) = type :- { proplist }
  proplist == getPropList(props, key),
  PROPVAL_TYPE(type) == getPropOptionalTypeFromList(proplist).

getPropOptionalTypeFromList : list((path * (string * (PROPVAL)))) -> PROPVAL
getPropOptionalTypeFromList([]) = PROPVAL_TYPE(NOTYPE()).
getPropOptionalTypeFromList([(_, (_, propval)) | _]) = propval.

getPropList : PROPS * STRING -> list((path * (string * (PROPVAL))))
getPropList(PROPS(s), key) = proplist :-
  query prop
    filter EXTEND? and { x' :- x' == key }
      min and true
      in s |-> proplist.
```

Figure 3.27: Statix abstractions for storing generic properties as scope graph datums. This snippet is abbreviated, as actually more getter and setter constraint are featured in the Statix specification for IceDust for values of different sorts

## 3.4 Integration with the compiler back-end

Some outcomes of the static analysis step in the front-end of the compiler pipeline may provide valuable information for further guiding the steps in the back-end of the compiler pipeline, such as optimization and code generation.

IceDust was modelled such that code generation is not syntax-directed but scope-graph directed. This is rather unconventional and was done with the idea in mind to decrease the coupling between the SDF3 grammar and the Stratego code, such that grammar changes do not necessarily require code-generation rule changes. However, as a trade-off, this increases the coupling between the static analysis and the code generation.

This particular design decision in the original IceDust implementation posed a number of interesting challenges when attempting to migrate also the back-end of the existing implementation, in the form of Stratego code, to the new Statix based implementation. These challenges proved to be too big to overcome within the scope of this research, and migrating this part of the IceDust implemenation was therefore left out of the scope. However, in exploring these challenges, we were able to learn more about the effort involved in developing Statix specifications, as its interfaces with other pipeline stages do influence this process. In this section, we therefore discuss the challenges we encountered and the possible solutions we explored.

### 3.4.1 Constructor sharing

Sharing constructors between Statix and Stratego that were not generated from the SDF3 grammar has to be done through duplication. Whether all constructors should be part of the grammar (even though they do not have a syntactic representation, as they never appear as a term in the AST) can be a point of discussion.

### 3.4.2 Querying the scope graph

Accessing the static analysis results such as the constructed scope graph during code generation requires using the a Stratego API to the static analysis.

Due to the original decision to make the code generation specification of IceDust scope graph directed, the back-end code is highly coupled with the Stratego API to NaBL2. Because, at the time this case study was conducted, the Stratego API to Statix was quite different and relatively immature when compared to the Stratego API to NaBL2, migrating this was not practically feasible yet.

One main reason for this lies in performing resolutions on the scope graph during code generation to obtain relevant data. In the NaBL2 API, resolutions can be performed. Given an AST term representing a reference identifier, the occurrence that was added to the scope graph during analysis corresponding to this identifier can be constructed. This is because the the AST term not only holds a string value, but also an associated term index. This term index represents the position in the AST at which the term originally occurred, allowing for traceability. Both the NaBL2 constraint generation rule and the Stratego transformation rule have access to this same term index, allowing for a one-to-one correspondence. A resolution can then be performed via the API to find the matching declaration.

In Statix there is no similar notion of an *occurrence* that can be constructed and identified by a unique term index. This means that, only given an AST node, there is no way to directly obtain its corresponding reference, scope or declaration. Furthermore, the Statix API does not allow to perform queries such as in the Statix specification. Given a scope, only the associated data and edges can be retrieved. This means resolving references to declarations would require implementation of a custom resolution algorithm using Stratego transformation rules, traversing edges one-by-one in a correct sequence, which is non-trivial and would

defeat the purpose of Statix implementing this behaviour already. Finally, the Stratego API to Statix does not provide direct access to the root scope of the scope graph, to bootstrap any querying on the static analysis results. A workaround was devised during the migration process of IceDust, where the root scope was assigned as a term attribute to the root AST node. However, the absence of a dedicated API endpoint to retrieve the root scope can be regarded a limitation [1].

### 3.4.3 Accessing code generation directives

In Section 3.2 we discussed the ability of NaBL2 of storing arbitrary key/value pairs with scope graph declarations. We presented an abstraction that we defined in our Statix specification to achieve a similar way of storing these properties.

In addition to storing additional contextual information during static analysis, these properties were also utilized in the NaBL2 specification of IceDust for storing information that is aimed to guide code generation and as such is depended upon by the operational semantics. During transformation, these properties can then be accessed via the Stratego API to NaBL2.

We modeled these code generation directives with our earlier introduced abstraction for storing properties for scope graph datums in Statix as well. In theory, these may be retrieved through the Stratego API to Statix if one were to migrate the compiler back-end to integrate with the Statix specification.

As the amount of declarations requiring these properties is significant, as well as the amount of properties per declaration, this has the drawback of bloating the scope graph with a large amount of scopes and datums serving the purpose of storing these code generation directives.

In conclusion, in this chapter we elaborated upon how NaBL2 and Statix differ in the way they allow definition of type compatibility relations, scope graph manipulation and integration with other stages of the compiler pipeline. We explained the impact of these differences on the implementation effort and readability of the resulting specifications for IceDust. In order to be able to draw conlusions from these observations and answer our research questions, we describe the way we evaluated our work in the next chapter.

[1] https://github.com/metaborg/nabl/issues/84

# Chapter 4

# Evaluation

In this chapter we present our method and results for evaluating and comparing the expressiveness, correctness and runtime performance of the Statix and NaBL2 specifications for IceDust.

## 4.1 Expressiveness

We performed evaluations to compare Statix to NaBL2 in terms of its expressiveness, given the case of IceDust. The approach to this consisted of both a qualitative and a quantitative evaluation method.

### 4.1.1 Comparing solutions to challenging semantics specification cases

We analysed and compared certain parts of the specifications that revolved around particular use cases in static semantics definition. The details and results of these analyses are described in Chapter 3 of this thesis.

In Section 3.1 we elaborate upon differences in the way NaBL2 and Statix facilitate the modeling of binary relations for the purpose of defining type compatibilities. We find that due to the omission of a dedicated language construct for modeling relations from Statix, more effort is required to define and test relations for type compatibilities using more generic language constructs. Different approaches are required for modeling static and dynamic relations. The results are less concise, and require complex querying of the scope graph.

Subsequently, in Section 3.2, we look at how the construction and querying of scope graphs differs between NaBL2 and Statix. We show that Statix, due to the "scopes-as-types" approach it supports, allows for working with scope graphs that are easier to reason about since they can express the same meaning while consisting of less elements. This does come with the drawback that boilerplate query definitions are required in trivial cases of name resolution.

In addition, in Section 3.3, we describe how we defined additional constraints for storing arbitrary properties as datum in the scope graph, and how this introduced additional complexity to the specification.

Finally, in Section 3.4 we look at the constraints that migration from NaBL2 to Statix imposes on the interfaces between static analysis and the other compiler pipeline stages. Due to significant differences between the Stratego API for NaBL2 and the Statix counterpart, in combination with the scope graph directed approach to code generation of IceDust, integrating the Statix specification with the compiler backend appears non-trivial.

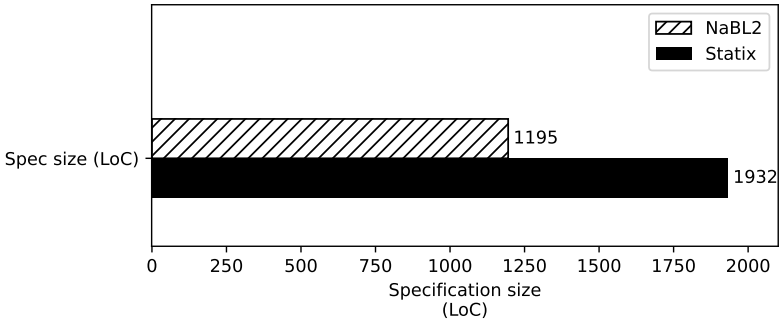



Figure 4.1: Comparison between the sizes in LoC of the NaBL2 and Statix specifications.

### 4.1.2 Comparing specification sizes

The sizes of the specifications were measured using a custom profile for the cloc tool that excludes empty and comment lines. Figure 4.1 plots the results of the measurements. From these results it appears that the Statix specification is less concise than the NaBL2 specification. As far as defining the static semantics of IceDust is concerned, Statix therefore appears to be a less expressive definition formalism than NaBL2.

## 4.2 Correctness

In order to allow for a comparison between the specifications as well as answering our research question RQ1, the correctness of the implemented Statix specification must be evaluated. This correctness should be relative to the prior specification, implying a functional equivalence between the specifications. This means that the Statix specification is correct, if the static analysis resuls it produces are the same as those produced from the NaBL2 specification, for any case. We approached evaluating this equivalence using a number of methods which will be described in this section. First we describe the method we applied to ensure the completeness of the Statix specification. Having established the completeness of the specification, the remainder of our evaluation is then solely dedicated to verifying the correctness of the resulting analyses. We describe the different software testing techniques that we applied to this end.

### 4.2.1 Completeness verification

During the migration process from the prior to the new specification, we exhaustively visited all the constraints listed in the prior specification while ensuring that there is a counterpart in the new specification. This contributes to establishing the completeness of the Statix specification.

The way this comparison method was applied, is exemplified in Figure 4.2. In this relatively trivial example, we compare the definition of the constraint rule that is applied to function declarations in an IceDust program between the Statix (left-hand side) and the NaBL2 (right-hand side) specification. To ease the comparison, for the purpose of this example we reordered the constraints, provided them with comments and vertically aligned them. As can be observed from the figure, at an abstract level all the required constraints that constitute this rule are present in both specifications. In turn, this method has been applied to the definitions of dependency constraints featured in the rule (such as `declareFunction` and `parametersOk`). In general, the application of the method was driven by all constraint rules in the NaBL2 specification, to ensure that for each of them there is a complete counterpart in the Statix specification.

```
functionOk(s, Function(f, ps, t, m, e)) :-
  { function_s props_s paramtypes type mult
    richtype e_type e_mult e_strat }
  // 1: introduce declaration with properties
  // to the scope graph
  new props_s,
  setPropRichtypes(props_s, "paramtypes",
    paramtypes),
  declareFunction(s, f, richtype,
    PROPS(props_s)),

  // 2: introduce new scope to the scope graph
  new function_s,
  function_s -PARENT-> s,

  // 3: resolve parameter types
  type == typeOfAnyType(s, t),
  mult == multOfMultList(m),
  richtype == RICHTYPE(type, mult, e_strat),

  // 4: resolve parameter types
  paramtypes == parametersOk(function_s, ps),

  // 5: resolve expression type
  RICHTYPE(e_type, e_mult, e_strat) ==
    richTypeOfExp(function_s, e),

  // 6: assert expression type is compatible
  // with static type signature
  subtype(e_type, type) |
    error $[type mismatch: ...]@e,
  // 7: assert expression multiplicity is
  // compatible with static multiplicity
  // signature
  mult(e_mult, mult) |
    error $[multiplicity mismatch: ...]@e.
```

(a) statix

```
[[ Function(f, ps, t, m, e) ^ (module_scope) ]] :=
  // 1: introduce declaration with properties
  // to the scope graph
  Function{f} <- module_scope,
  Function{f} : TTuple(f_type, f_mult, e_strat),
  Function{f}.paramtuples := ps_tuples,
  Function{f}.params := ps,
  Function{f}.expr := e,
  Function{f}.ns := Function(),


  // 2: introduce new scope to the scope graph
  new f_scope,
  f_scope -P-> module_scope,

  // 3: resolve static type signatures
  [[ t ^ (module_scope) : f_type ]],
  [[ m ^ (module_scope) : f_mult ]],


  // 4: resolve parameter types
  Map1T [[ps ^ (f_scope) : ps_tuples]],

  // 5: resolve expression type
  [[ e ^ (f_scope) :
    TTuple(e_type, e_mult, e_strat) ]],

  // 6: assert expression type is compatible
  // with static type signature
  e_type  <sub? f_type |
    error $[Type error: ...]@e,
  // 7: assert expression multiplicity is
  // compatible with static multiplicity
  // signature
  e_mult <mulOrd? f_mult |
    error $[Multiplicity error: ...]@e.
```

(b) nabl2

Figure 4.2: Example of how the completeness of the Statix specification was verified through exhaustive comparison between corresponding constraint rules.

### 4.2.2 Automated testing

As mentioned in Section 2.2, the Spoofax language workbench features the SPT meta-language for declaring automated test cases to verify the correctness of a language implementation. For the original NaBL2-based implementation of the IceDust language, a collection of SPT test suites was developed, including but not limited to test cases that verify the static analyses resulting from the NaBL2 specification.

In order to evaluate whether our Statix specification results in the same analyses as the NaBL2 specification, we migrated and applied 88 of these SPT test cases in the test suite for the prior specification to the Statix specification and ensured they passed. A screenshot of the results of the test execution is displayed in Figure 4.3. As can be seen in the screenshot on the left-hand side, a test case consists of a fixture with corresponding assertions. The fixture is expressed in the form of textual fragment that represents a program written in the language under test. The assertions are expressed in the form of specific expectations related to an analysis or transformation that is performed on the program as per the language definition (e.g. parsing, static analysis, code generation).

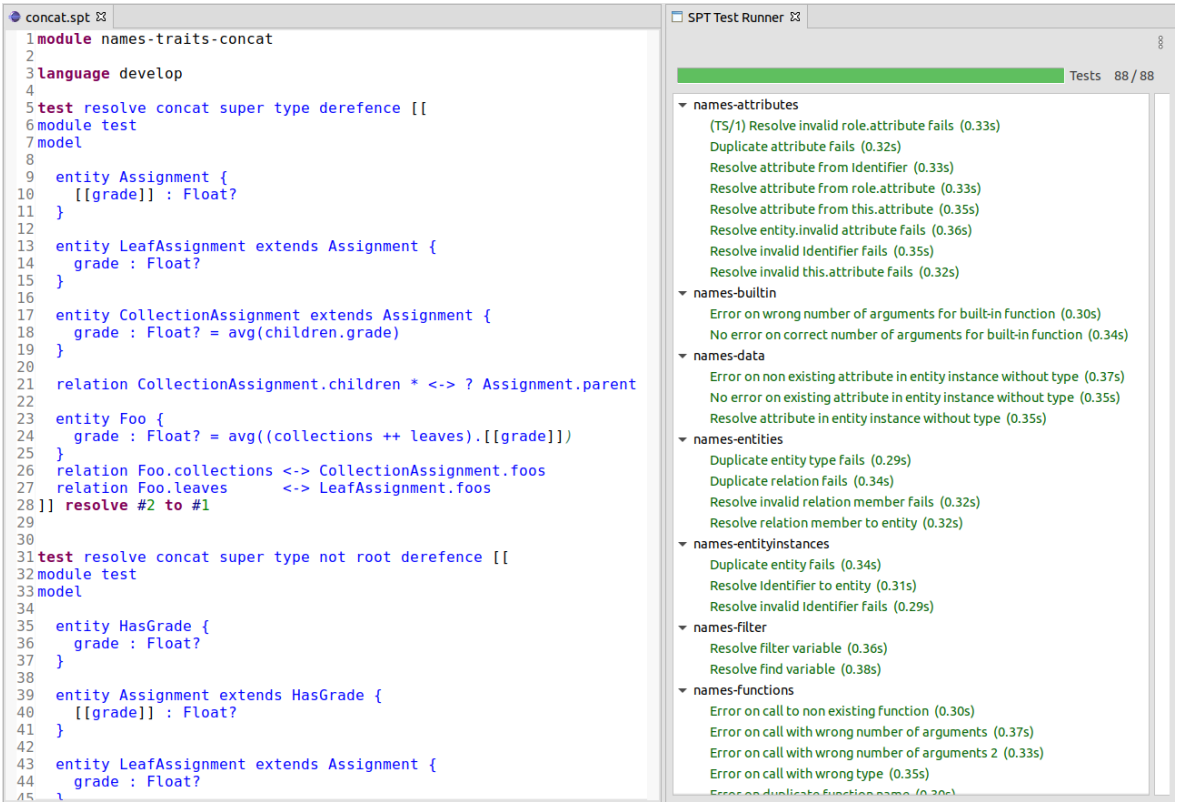



Figure 4.3: Screenshot of the results from executing the test suite on the Statix specification displayed in the SPT Test Runner of the Spoofax IDE.

The selection of the 88 test cases does not comprise the entire prior test suite, but covers those cases that have expectations dedicated specifically to testing the static analysis, in particular the more complex cases of name resolution. It was not feasible to migrate other static analysis related test cases due to the dependence of their expectations on the Stratego API for NaBL2 and the limitations involved in migrating this as described in Section 3.4.

### Test coverage analysis

We aimed to quantitatively evaluate how our selected test cases actually cover the constraints in our Statix specification, in order to measure the degree to which the constraints are covered, as well as to identify undertested constraints in order to guide our further testing efforts. However, no prior work has been done on measuring such a notion of test coverage with the SPT framework. In order to make this analysis feasible within the scope of our research, we devised a method to estimate the test coverage. The procedure that we scripted for this purpose can be outlined in the following three steps:

1. Extract for each test case in the SPT test suite all the term constructors that are featured in its fixture.
2. Extract for each rule in our Statix specification the term constructors featured in the pattern of the head of the rule.
3. Cross-reference the term constructors extracted from the test cases and from the Statix rules to infer for each rule in the specification in which test cases it will be applied.

**Extracting test cases** Using the core Java API of the SPT testing framework, in combination with the programmatic API of Spoofax, we were able to extract the textual program frag-

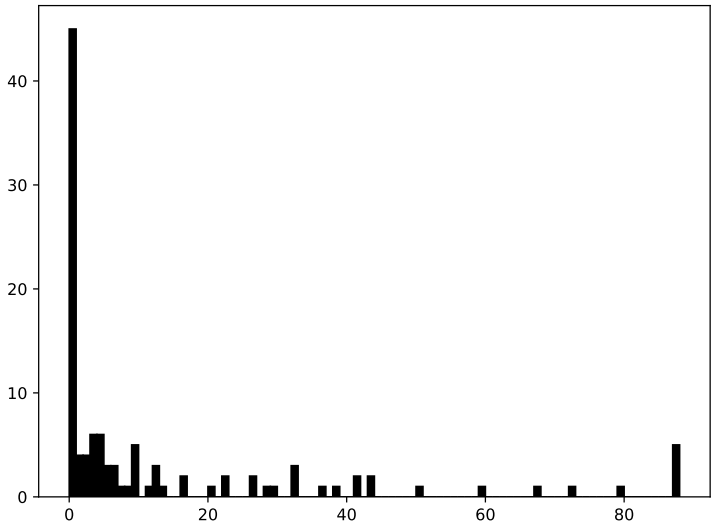



Figure 4.4: Distribution of the constraint rules (y-axis) over the the amount of tests covering them (x-axis).

ments that made up the fixtures of the test cases in our test suite. These fragments where then parsed and transformed using the same strategy that is applied when preparing the abstract syntax tree for the Statix analysis, which includes desugaring and explicating injections. Then, from these final pre-analysis abstract syntax trees, we were able to infer which term constructors may be encountered when traversing the AST during analysis.

**Extracting constraint rules** We parsed the constraint rules from our Statix specification and selected those that have a pattern in the rule head in terms of at least one AST term constructor. This allows us to infer, given a term constructor occurring in the AST, which rules in our specification may be applied to it.

**Cross-referencing test cases with constraint rules** Based on the extracted term constructors from the test cases and constraint rules, we infered for each rule in our specification by which test cases it was covered, and calculated various statistics as a result.

**Results** Ultimately, our three stage analysis outlined above resulted in the distribution of the constraint rules over the amount of tests cases they are covered by as plotted in Figure 4.4. In total 60.4% of our Statix constraint rules are covered by at least one test case. The remaining 39.6% percent of constraints not covered by any test case were inspected in detail and are summarized in Table 4.5. This summary primarily indicates that our test suite does not exhaustively cover all of the primitive data types, built-in functions and logical and mathematical operators featured in the language. Since many of these instances are analysed in the same or a similar way, the absence of tests covering these particular instances does not appear to be a siginificant shortcoming of the test suite, and we conclude that most of the significant constraints are covered. Nonetheless, we used this information for further guiding our manual testing efforts and spent extra attention to verifying the correctness of the constraint rules not covered by the test suite.

**Discussion** The inference of which rules apply to which test case is based on the assumption that if a term occurs in the program, any rule matching that term will actually be applied during analysis. In general this should be the case, as most rules are defined to apply to any

| Constraints with amount of uncovered rules | | | |
|---|---|---|---|
| Constraints | # Rules | | Scope of the rule application |
| relationInstanceOk | 2 | 100% | Instances of relations declared in the data section of a program. |
| typeOfValue | 1 | 33% | Declaration of the value for a shortcut of an entity including both a relation instance and an entity instance in the data section of a program. |
| typeOfEntityInstanceValue | 1 | 50% | Entity instance references within the data section of a program. |
| richTypeOfExp | 24 | 59% | Various unary and binary operators as well as built-in functions. |
| multOfMultOrd | 2 | 66% | Multiplicity declarations: OneOrMore-Ordered and ZeroOrMoreOrdered. |
| zeroness | 1 | 33% | The literal integer "0". |
| stratOfStrat | 4 | 80% | Calculation strategy declarations: OnDemand, OnDemandIncremental, Eventual, OnDemandEventual. |
| typeOfAnyType | 1 | 50% | Declaration of a static type signature for an entity type that may be used in derivation attributes and function parameters. |
| typeOfPrimitiveType | 2 | 40% | Static type signature declarations for: Boolean and Datetime. |
| richTypeOfLitVal | 4 | 57% | Literal values: True, False, Datetime and NoValue. |

Figure 4.5: Summary of the rules not covered by the test suite for our Statix specification, grouped by constraint.

instance of the term in the AST, and are coupled such that they are transitively applied in order to traverse the entire AST in a topdown manner. However, there may be small exceptions where a rule is only applied if a term appears within a particular context. These exceptions may lead us to overestimate coverage. As the exceptions are limited to just a few cases, we chose to tolerate this small overestimation.

In addition, the set of constraint rules being included in the coverage analysis only includes constraint rules that have a rule head pattern in terms of constructors that may occur in the AST. This assumes that all other constraint rules are transitively applied. As the definition of the constraint rules is primarily syntax-directed, and constraint rules with patterns expressed in terms of non-AST terms (e.g. types) are generally abstractions utilized in rules with patterns expressed in terms of AST terms, we expect this assumption to hold.

Finally, whenever there are multiple rules matching on the same term constructor defined for the same constraint, this created an ambiguity. We had to manually assess which rule would be applied as it was not feasible to incorporate the actual pattern matching in this procedure. In the cross-referencing script we incorporated an interactive prompt for manual selection of the applicable rules that was displayed each time such an ambiguity was encountered. In total 120 cases were manually assessed. This mostly involved the same few constraints for each test case.

### 4.2.3 Manual testing

In addition to ensuring our specification passes all the selected automated tests, we also performed manual acceptance testing of our specification. By testing on different levels and using different approaches, we ensured that the analysis resulting from our specification both validates correct programs and invalidates erroneous programs.

#### Confidence testing

Throughout the evolution of our specification we performed high-level smoke tests to build confidence in our solution. We did this by running the static analysis resulting from our specification on a set of example IceDust programs describing real world data modeling cases that jointly have a large coverage of the constructs and combinations thereof that are supported by the language. These example programs include but are not limited to the programs comprising the benchmark dataset listed in Table 4.6. We utilize the prior implementation of IceDust to ensure the validity of these programs to allow them to be used as ground truth.

The final implementation of our Statix specification results in static analyses that pass on each of these programs. This tells us that for positive cases, our Statix specification results in correct recognition of valid programs. The static analysis outcomes were further verified by utilizing the editor services generated from the specification. This includes navigating through references to see whether they resolve to their expected declaration, hovering over elements in the program to validate inferred type information, and generating and inspecting the resulting scope graphs to ensure their correctness.

However, this does not tell us anything about whether our specification also correctly invalidates programs with any static errors. To evaluate this, we performed manual mutation testing at a more granular level by introducing any kind of defect into a program to see if subsequently an error is correctly raised, which is described in the next few sections.

#### Whitebox testing approach

We indexed all 130 constraints in our specification that have an error level message supplied with them. These messages indicate that the constraint is expected to fail in case of a particular defect being present in the program under analysis. We manually verified that the corresponding constraints fail when introducing a corresponding defect into a program. In addition, we utilized the outcome of our test coverage analysis to test the specific constraints that were not covered by our test suite. This included exhaustive testing of all operators and built-in functions to see if they fail when applied to sub expressions with incompatible data type, multiplicity bound or calculation strategy.

#### Blackbox testing approach

IceDust can be considered a configuration language for a feature model revolving around the construct of a “field” (D. C. Harkes and Visser 2017). The features that may be selected for a field do not support full orthogonality. In addition, when fields are composed, which can be done by introducing references to fields into the definition of other fields and performing operations on them, there is a definition for what it means for these compositions to be sound. Guided by the feature model, and the restrictions imposed on the variability it supports, we tested for a large variety of invalid feature selections and compositions. For example, for the multiplicity bounds and calculation strategies, lattices are defined describing exactly which compositions for these feature are valid and which are not.

### 4.2.4 Incompleteness in the prior specification

During the migration process we noted that some parts of the original NaBL2 specification were incomplete due to the omission or commenting out of certain rules or constraints.

A notable example was the "trait" language construct, which is an element that may be declared in an IceDust program at module level alongside entities and relations. It appears to be intended as an abstraction mechanism for code reuse, to allow certain behaviors to be mixed into multiple entities. However, this appeared to be an experimental feature as both the syntax definition as well as the NaBL2 constraints were commented out.

In addition, within the constraint generation rules for particular kinds of field declarations, some constraints were commented out for asserting the compatibility between the declared and actual expression data type, multiplicity bound and/or calculation strategy. There appears to be no clear indication as to why these constraints were left out.

Because we define our notion of correctness to be relative to the prior specification, we also left these constraints out of our Statix specification and disregarded them for the purpose of our evaluation.

## 4.3 Runtime performance

In order to answer research question RQ3, we assess if there are any significant gains or losses in the runtime performance of the typechecker implementation that is generated by the Statix specification described in chapter 3 compared to its NaBL2 counterpart. We do this by measuring the duration of the analyses of these typecheckers on various example IceDust programs.

### 4.3.1 Method

We perform benchmarks to gain insights in the runtime performances of the typecheckers under review. We run macrobenchmarks assessing the complete analysis time on a number of real-world example programs. This analysis time is representative for the response times experienced by the end-user when interacting with the generated editor for the object language. During the benchmark we run the full analyses based on the Statix and NaBL2 specifications. To this end, the `spoofax-analysis-benchmark`[1] tool was implemented. This tool is based on the Java Microbenchmark Harness (JMH). It utilizes the Spoofax programmatic API and was made language parametric so it can be used to invoke the compiler pipelines of both the IceDust implementations. The actual measurements isolate the analysis stage of these pipelines. The benchmark performs 5 warmup iterations followed by 20 measurement iterations for each example program. The result is then computed as the arithmetic mean of the 20 measurements.

The example programs that form the benchmark datasets are identical for the benchmarks of both specifications in order to enable a fair comparison. In total 3 programs were developed or included for this purpose, the statistics of which are summarized in Table 4.6.

The benchmarks were executed on a desktop PC running Ubuntu 20.04.5 LTS with the following specifications: Intel® Core™ i7-2700K CPU @ 3.50GHz; 16GiB DIMM DDR3 Synchronous 1333 MHz; and a 256GB SSD (SATA 600).

### 4.3.2 Results

The results collected from the benchmark are summarized in Figure 4.7. The results show that for all benchmarks, the Statix specification results in a longer analysis duration than the NaBL2 specification, the increase of runtime ranging from 21 to 36 percent. Altogether,

[1] https://github.com/metaborg/spoofax-analysis-benchmark

| Benchmark Dataset | |
|---|---|
| IceDust program | Size (LoC) |
| icedust model | 175 |
| accounting | 293 |
| weblab | 435 |

Figure 4.6: Listing of the specifications written in IceDust that made up the dataset used for the benchmark.

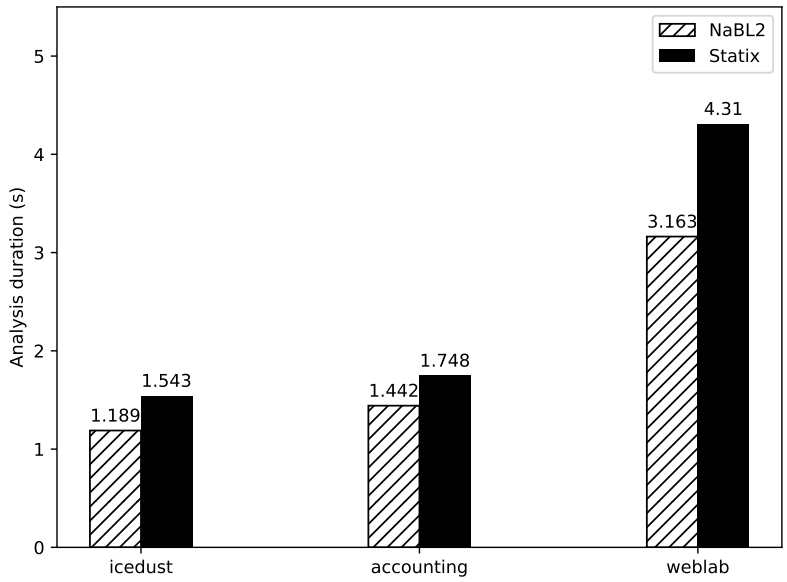



Figure 4.7: Comparison between the analysis durations of the NaBL2 and Statix specifications applied to different example IceDust programs.

we note that the Statix specification results in a slightly slower typechecker than the NaBL2 specification.

### 4.3.3 Discussion

We performed a benchmark of the full analysis procedures for the Statix and NaBL2 specifications. The entrypoint of this procedure is an invocation of the Stratego runtime from the Java API, which is thus at a relatively high level of abstraction. This makes it difficult to pinpoint the exact cause of the difference in runtime performance between the two implementations. For example, it is not clear whether the increase in runtime can be attributed to differences in the constraint solvers or to overhead in the analysis procedure at higher levels of abstraction.

In addition to any differences between the implementation of the NaBL2 and Statix runtime environments, there are also differences between the way we implemented the Statix specification of IceDust and the way the prior specification was implemented. These difference may also have affected the benchmark results. For example, with Statix we have had to introduce a number of additional complexities at the specification level in order to account for the lack of dedicated meta-language constructs. At specification level, these complexities may be less optimized than at the meta-language level. The complexities include for example the testing of certain type compatibilities by querying the scope graph as discussed in Section 3.1.2, and the maintainance and querying of all declaration properties as datums in the scope graph as discussed in Section 3.3. These particular solutions may cause a considerable increase in the amount of scope graph queries being performed during analysis. In

theory, if alternative solutions to capturing these complexities in a Statix specification are found, this could allow for further optimization of the runtime performance at specification level. However, this is challenging due to the highly declarative nature of Statix.

Also, we note that the benchmark results are based on a small dataset. Nonetheless, the benchmark dataset does include a number of real-world example programs. We therefore expect these results to be representative for actual use cases of IceDust.

In summary, in this chapter we discussed the way we evaluated our work by comparing the expressiveness, correctness and runtime performance of our Statix specification and the prior NaBL2 specification for IceDust. The evaluation of expressiveness showed that overall, Statix appears to be a less expressive definition formalism for the static semantics of IceDust than NaBL2, allowing us to answer RQ2. We also evaluated the correctness of our specification relative to the prior specification, validating our findings and showing that is possible to express the semantics of IceDust with Statix, which answers RQ1. Finally, our benchmark showed that the analysis resulting from our specification is slightly slower than the one from the prior specification, which provides an answer to RQ3. The next chapter discusses work related to this thesis, describing alternative approaches to abstractly and declaratively implementing typecheckers.

# Chapter 5

# Related work

## 5.1 Abstractions for type checker implementation

In this study, the Spoofax language workbench and its meta-DSLs were employed to formalize the static semantics of a programming language and generate language processor implementations accordingly. Alternative solutions enabling the declarative and iterative development of aspects of domain-specific languages have been researched. We look at related work studying approaches to abstracting over the implementation of type checkers, as well as - if applicable - the software language workbenches they are incorporated in.

**Ott** Ott is a metalanguage allowing to express the semantics of object languages in a concise way (Sewell et al. 2010). It mechanizes the work involved in semantics definition, by compiling specifications to artifacts such as code for various proof assistant back-ends and a LaTeXtypeset variant of the specification. Ott takes a more integral approach to specifying semantics, by also including operational semantics, whereas Statix is dedicated to describing static semantics. Nonetheless, the metalanguages are similar in their ability to define type systems and name binding rules. Similarly to our analysis, the research on Ott also includes case studies that test the ability of the metalanguage to post-facto formalize existing object languages. In addition, one of the main goals driving the research on Ott is to make definitions concise and easy to read and edit by minimizing syntactic overhead, with the intent of increasing semanticist productivity. The degree to which the resulting metalanguage was indeed "intuitively clear, concise, and easy to read and edit" was assessed by means of an informal, empirical analysis based on hands-on experiences with the language, similar to ours. However, as Ott lacks a clear predecessor, the analysis does not include any comparisons with alternatives.

**Needle and Knot** KNOT is a metalanguage that allows for concisely expressing the name binding structure of a programming language, automatically generating a corresponding mechanized proof (Keuchel, Weirich, and Schrijvers 2016). Using the NEEDLE tool, a KNOT specification can be compiled to code for the Coq proof assistant including necessary boilerplate code. The research aims to reduce boilerplate code a semanticist has to write in order to mechanize a proof. It therefore includes an evaluation to assess the expressivity of the metalanguage by measuring the sizes of KNOT specifications in lines of code (LoC) for the name binding of various object languages. These sizes are compared to the sizes of equivalent specifications constructed using different approaches, including both an unassisted approach as well as alternative abstractions for mechanizing metatheory from the POPLmark benchmark. The results show that KNOT allows for some substantial savings in specification size.

**Turnstile** Turnstile is a metalanguage similar to Statix with regards to its goal of bridging the gap between semantics specification and implementation (Chang, Knauth, and Greenman 2017). It offers a similar abstraction mechanism for specifying the static semantics of an object language and is able to generate a type checker implementation from such a specification. It uses a macro-based approach, by reusing the binding on the underlying macro system. Turnstile+ is a later iteration of Turnstile that introduced the support necessary to model dependently typed languages (Chang, Ballantyne, et al. 2020). Turnstile+ is intended to be backwards compatible with Turnstile specifications. This evolution is comparable to the introduction of Statix as successor to NaBL2, where Statix aims to retain the same coverage and completeness as NaBL2.

**Xtext and Xsemantics** The Xtext language workbench offers a framework for developing domain-specific languages and associated tooling (Efftinge and Völter 2006). With a rich set of features, including a concise grammar notation, automated editor generation, and integrated validation and code generation, Xtext provides a comprehensive solution for creating languages and their accompanying development environments. Furthermore, Xtext is extensible as the behaviour of generated components can be customized through an API.

Xsemantics is a DSL for implementing type systems that is part of the XText language workbench (Bettini 2016). Similar to Statix, its syntax aims to resemble a formalization of the type system. The DSL also transforms abstract syntax trees to a Java-based type checker implementation based on a set of rules. Their approach distinguishes between the features of "scoping" and "validation", which are similar to the abilities of Statix to deal with name binding and type checking respectively. Instead of using scope graphs for name resolution as in the case of Statix, Xsemantics allows capturing relationships between references and declarations during grammar definition. Based on this, the parsed AST will actually become a graph that already captures the binding structure. During static analysis, name resolution will then be further restricted based on scopes, an abstract concept equivalent in its meaning to scopes in Statix. Similar to Statix in Spoofax, because of its integration with XText, Xsemantics features extensive Eclipse IDE-based tooling, both for the purpose of implementing specifications themselves as well as transforming such a specification into an editor for the modeled object language featuring a type checking editor service. The DSL also has features dedicated to "help the developer to implement type systems easily". The study assesses the applicability of the DSL by demonstrating its use for prototyping a number of example object languages (including Featherweight Java and $\lambda$-calculus).

**JastAdd (attribute grammars)** Syntax-directed translation is an approach to implementing a compiler in which the translation from source to target language is completely driven by the parser. This can be formalized using a syntax-directed definition. Attribute grammars are a formalism that allow to construct such a definition. They extend the production rules of a context-free grammar of a language with additional rules that describe how particular attributes (i.e. name/value pairs) should be derived for their corresponding symbols. In the syntax-directed translation process these rules may then be used to calculate for each node in the abstract syntax tree its corresponding attributes. The result is referred to as an "annotated" or "decorated" syntax tree. The attributes can capture context-sensitive / semantic information in addition to the syntactical structure. A specialization of the attribute grammar formalism is the reference attribute grammar, which allows the values of attributes to be references to other attributes.

JastAdd is a Java-based tool for implementing compilers (Hedin and Magnusson 2003). It supports mixing both an imperative and an declarative programming paradigm based on attribute grammars, which may be selected on a per subproblem basis. In comparison to the single paradigm that Statix is committed to, the ability that JastAdd offers to escape between

paradigms has a clear benefit. For example in Figure 3.15 we presented a case in which it was challenging to express certain logic in the declarative paradigm that Statix supports, for which an imperative approach may have been a better fit.

The declarative paradigm that JastAdd supports is based on reference attribute grammars. It utilizes an object-oriented abstract syntax tree, in which nodes are instances of classes that may hold references to other nodes. This allows also capturing the name binding structure of a program within the AST, by creating references between nodes that represent identifier use sites and nodes that represent the corresponding declarations. This is an alternative to the approach of Statix, where static semantic information is captured in a scope graph which is separate from the AST.

The concept of the decorated attribute grammar is a specialization of the attribute grammar that combines attribute evaluation with strategic programming (Kats, Sloane, and Visser 2009). This formalism allows for the definition of attributes and their dependencies, enabling the specification of type constraints and semantic rules. Attributes are associated with nodes in the abstract syntax tree, and their values are computed and propagated according to attribute evaluation rules. The strategic programming aspect of decorated attribute grammars enables the specification of evaluation strategies for attribute dependencies, providing fine-grained control over the order and conditions of attribute computation. This allows type checkers with complex language specific analyses to be implemented based on decorated attribute grammars.

**MPS (projectional editing)** Instead of editing a core definition directly, projectional editing systems hold a definition in a model and allow editing the definition through projections of the model. This allows for editing environments that are different from textual editors, for example in the sense that they can be more graphical. MPS is an example of a workbench that allows the definition of software languages through projectional editing (JetBrains n.d.). It offers an experience close to traditional text editing as the projection still resembles a textual form. However, due to it nonetheless presenting a projectional editor rather than a textual one, it allows for the inclusion of some more visual representations such as decision tables and graphs. For defining the static semantics of a language, MPS allows defining type checking and inference rules using a constraint logic programming paradigm similar to what Statix offers. These rules impose constraints that are solved by the internal engine. As a generalization of this, the more recent CodeRules feature of MPS offers abstraction mechanisms that enable developers to express more complex type rules and constraints. This allows users to specify language-specific constraints, such as type compatibility and inference, among others, more precisely.

**Rascal (meta-programming)** The RASCAL domain-specific language allows for the implementation of full compiler pipelines including analysis and transformation of programs by means of meta-programming capabilities (Klint, Storm, and Vinju 2009). Like Spoofax, syntactic features of RASCAL are based on SDF. By leveraging RASCAL's expressive syntax and library of built-in functions, language engineers can define static typing rules for a given object language. RASCAL's flexible type system allows for the specification of complex type constraints, including type compatibility, type inference, and type coercion. Moreover, RASCAL's pattern matching capabilities enable the extraction of relevant information from source code, facilitating the identification and resolution of typing errors.

**Datalog and Formulog (satisfiability modulo theory)** Pacak et al. propose a novel approach for deriving type checkers based Datalog, a logic programming language (Pacak, Erdweg, and Szabó 2020). It comprises a DSL that supports writing inference rules for expressing static semantics which are then translated to Datalog rules. By leveraging the ex-

pressive power of Datalog and its ability to reason over relations and constraints, the authors demonstrate how this can be used to automatically derive efficient type checkers. Datalog maintains a datastructure consisting of facts, rules and queries that is referred to as a deductive database. In the application of Datalog for modeling static semantics, this database serves a similar function as the scope graph in the case of Statix. The authors performed several case studies to assess the expressivity of the DSL, similar to our study.

Bembenek, Greenberg, and Chong propose a similar Datalog-based approach by introducing FormuLog, a declarative language for defining type systems (Bembenek, Greenberg, and Chong 2022). FormuLog allows type checking constraints to be expressed in a more concise and intuitive manner compared to directly writing Datalog rules. It aims to provide a safe and abstract interface to the underlying SMT solver. By further abstracting the type system definition formalism this way, a user-friendly and expressive approach to implementing type checkers can be imagined. Case studies with FormuLog having similar objectives as our study demonstrate that the DSL allows for natural encoding of complex static semantics and results in satisfactory performant type checkers.

These solutions benefit from the performance and scalability of Datalog, which is well-established and optimized for solving constraint satisfaction problems.

# Chapter 6

# Conclusion

In the process of this case study, we aimed to assess and compare a number of high-level characteristics of the NaBL2 and Statix meta-languages when applied to the static semantics of the IceDust DSL. This resulted in a number of key insights.

Firstly, when investigating the encoding of relations on user-defined types along with their least-upper-bound and greatest-lower-bound functions in Statix and NaBL2, it appeared that encoding such relations requires more effort in Statix when compared to NaBL2. This is due to the fact that Statix lacks dedicated language primitives for the purpose of working with relations, contrary to NaBL2. Therefore, in our Statix specification we devised methods and abstractions to be able to define these same semantics, at the cost of introducing additional complexity to the specification.

In addition, when analysing the resulting scope graphs, we found that the flexibility of Statix in terms of querying and using scopes as types allows for the construction of more concise scope graphs that are easier to query, reason about and verify. However, we noted that this increased flexibility of Statix to a degree comes at the cost of increased implementation effort and potentially less readable specifications. This is due to the fact that more boilerplate query constraint definitions are required for trivial cases of name resolution, and at the same time more complex solutions may be required for non-trivial cases, when compared to NaBL2. In addition, since Statix lacks the ability to assign arbitrary properties to scope graph declarations, we had to define additional constraints for storing properties as datum in the scope graph. This introduced overhead in the size of the specification.

Furthermore, we encountered several challenges when investigating the requirements for also migrating the remainder of the IceDust compiler pipeline to integrate with the Statix based static semantics definition. We found that the Stratego API for Statix is not yet as mature as the API for NaBL2, which makes integration of static analysis results in the code generation phase not practically feasible yet, at least as far as more implementations with a high coupling between compiler front-end and back-end such as IceDust are concerned. This includes accessing name binding information in the scope graph, as well as arbitrary code generation directives recorded during static analysis.

In line with these findings, our evaluation of specification size showed that the Statix specification is significantly larger in terms of lines of code than the NaBL2 specification. Aditionally, our evaluation of runtime performance of the analyses showed that our Statix specification of IceDust results in a slower analysis than the prior NaBL2 specification.

With these insights we aim to answer our original research questions.

RQ1. Can the static semantics of IceDust be expressed using Statix, such that the resulting typechecker performs analyses to the same level of correctness as the one based on the NaBL2 specification?

We evaluated the completeness and correctness of our Statix specification relative to the

NaBL2 specification using a variety of automated and manual testing methods as elaborated upon in Section 4.2. This led us to conclude that Statix allows for the expression of the same semantics of the IceDust language as NaBL2 does.

RQ2. How significantly do NaBL2 and Statix differ in expressiveness, readability and implementation effort when comparing their specifications for IceDust, and which meta-language design choices are most impactful on this?

The designs of NaBL2 and Statix differ in the way they allow definition of type compatibility relations, scope graph manipulation and integration with other stages of the compiler pipeline. These differences appear to be most influential of the implementation effort and readability of the resulting specifications.

Looking at specific challenges in modeling the semantics of IceDust as summarized in Section 4.1.1, Statix tends to require the implementation of more complex solutions to overcome these challenges in the form of abstractions at specification level. This results in parts of the specification that tend to be more verbose and less readable, due to enumeration of cases and repetition of constraints for different sorts.

In line with these observations we also note that on a general level, the size of the Statix specification in terms of lines of code is significantly larger than that of the NaBL2 specification, as evaluated in Section 4.1.2.

This altogether suggests that Statix is a less expressive formalism for declaring the static semantics of the IceDust language.

RQ3. How do NaBL2 and Statix compare in terms of the runtime performance of their resulting type checkers for IceDust?

In our evaluation in Section 4.3 we showed that when disregarding incremental analysis, and looking at the total analysis times, Statix appeared to generate a slightly less performant type checker for IceDust than NaBL2.

In conclusion, in this thesis we have investigated some of the key trade-offs between Statix and NaBL2 in terms of meta-language characteristics. The advanced facilities of Statix for manipulation of scope graphs may demand more implementation effort and potentially results in less readable code when defining boilerplate for trivial name resolutions or complex constraints for type compatibility relations. It appeared to generate a slightly less performant typechecker for IceDust in terms of runtime than NaBL2. These findings contribute to a better understanding of the influence of advancements in meta-language design on their practical applicability. This may prove useful in guiding future evolution of abstractions for type checker implementation that aim to improve the productivity of language engineers.

## 6.1 Future work

Based on the insights gained from thesis, we recommend a number of directions for future research.

First of all, as elaborated upon in Chapter 3, Statix offers less constructs dedicated to specific aspects involved in the domain of static semantics specification than NaBL2. This required us to implement certain abstractions at the specification level in order for these aspects to be defined. In our evaluation in Chapter 4 we found that these abstractions introduce complexities into the specification that negatively impact implementation effort, readability specification size and the runtime performance of the resulting typechecker. In future work,

additional or altered meta-language features can be explored that support defining these aspects, in particular type compatibilities, more expressively and efficiently.

Additionally, in Section 3.4 we discussed our exploration of integrating our Statix specification with the existing back-end of the IceDust compiler. We noted that, due to the scope graph-directed approach to code generation, the Stratego code is highly coupled with the Stratego API for NaBL2. It appeared to be non-trivial to directly translate this integration to the Stratego API for Statix in its current state. Future work could investigate if the code generation implementation of IceDust can be migrated to work with the Stratego API for Statix, and if necessary further develop this API. This may be beneficial for practical adoption of Statix for language projects that require a complex, static analysis result driven code generation phase such a IceDust.

Furthermore, more work can be done to measure the general gain or loss in performance of Statix-based typecheckers relative to NaBL2. This could involve taking into account the impact of incremental analysis when making edits to a program under analysis, and profiling to identify specific parts of the analysis that can be further optimized.

Finally, as part of our evaluation of the correctness of our work in Section 4.2, we presented a method for approximating degree to which the constraints in our Statix specification were covered by our SPT test suite, as well as determining which constraints were covered and which were not. In future work, an exact method for measuring such a notion of test coverage for SPT test suites may be developed. To this end, the method may involve instrumentation of some of the internals of the Statix runtime environment such that information can be collected about which constraints were solved during the execution of a test case, while tracing these constraints back to lines in the specification. Such a solution would circumvent the discussed limitations of the approximation method we employed. This may prove beneficial in verification of Spoofax language projects, as test coverage reports may efficiently guide testing efforts of language engineers as well as build confidence in the correctness of the implementation.

# Bibliography


Bembenek, Aaron, Michael Greenberg, and Stephen Chong (2022). Formulog: Datalog+ SMT+ FP. In:

Bettini, Lorenzo (Aug. 2016). Implementing type systems for the IDE with Xsemantics. In: *Journal of Logical and Algebraic Methods in Programming* 85.5, pp. 655–680. ISSN: 2352-2208. DOI: 10.1016/J.JLAMP.2015.11.005.

Chang, Stephen, Michael Ballantyne, et al. (Jan. 2020). Dependent type systems as macros. In: *Proceedings of the ACM on Programming Languages* 4.POPL, pp. 1–29. ISSN: 24751421. DOI: 10.1145/3371071. URL: https://dl.acm.org/doi/10.1145/3371071.

Chang, Stephen, Alex Knauth, and Ben Greenman (Jan. 2017). Type systems as macros. In: *ACM SIGPLAN Notices* 52.1, pp. 694–705. ISSN: 15232867. DOI: 10.1145/3009837.3009886. URL: https://dl.acm.org/doi/10.1145/3009837.3009886.

Efftinge, Sven and Markus Völter (2006). oAW xText: A framework for textual DSLs. In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 118.

Harkes, Daco and Eelco Visser (2014). Unifying and generalizing relations in role-based data modeling and navigation. In: *International Conference on Software Language Engineering*. Springer, pp. 241–260.

Harkes, Daco C., Danny M. Groenewegen, and Eelco Visser (July 2016). IceDust: Incremental and eventual computation of derived values in persistent object graphs. In: vol. 56. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, pp. 111–1126. ISBN: 9783959770149. DOI: 10.4230/LIPIcs.ECOOP.2016.11. URL: https://drops.dagstuhl.de/opus/volltexte/2016/6105/.

Harkes, Daco C. and Eelco Visser (June 2017). IceDust 2: Derived bidirectional relations and calculation strategy composition. In: vol. 74. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, pp. 141–1429. ISBN: 9783959770354. DOI: 10.4230/LIPIcs.ECOOP.2017.14. URL: https://drops.dagstuhl.de/opus/volltexte/2017/7251/.

Hedin, Görel and Eva Magnusson (Apr. 2003). JastAdd—an aspect-oriented compiler construction system. In: *Science of Computer Programming* 47.1, pp. 37–58. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(02)00109-0.

JetBrains (n.d.). *Meta programming system*. URL: https://www.jetbrains.com/mps/.

Kats, Lennart CL, Anthony M Sloane, and Eelco Visser (2009). Decorated attribute grammars: Attribute evaluation meets strategic programming. In: *Compiler Construction: 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 18*. Springer, pp. 142–157.

Keuchel, Steven, Stephanie Weirich, and Tom Schrijvers (2016). Needle & Knot: Binder Boilerplate Tied Up. In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 419–445. ISBN: 978-3-662-49498-1.

Klint, Paul, Tijs van der Storm, and Jurgen Vinju (2009). RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 168–177. DOI: 10.1109/SCAM.2009.28.

Konat, Gabriël et al. (2013). Declarative name binding and scope rules. In: vol. 7745 LNCS, pp. 311–331. ISBN: 9783642360886. DOI: 10.1007/978-3-642-36089-3_18. URL: doi.org/10.1007/978-3-642-36089-3_18.

Neron, Pierre et al. (2015). A Theory of Name Resolution. In: *Springer* 9032, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9. URL: https://link.springer.com/chapter/10.1007/978-3-662-46669-8_9.

Pacak, André, Sebastian Erdweg, and Tamás Szabó (2020). A systematic approach to deriving incremental type checkers. In: *Proc. ACM Program. Lang.* 4.OOPSLA, pp. 121–127.

Runeson, Per and Martin Höst (2009). Guidelines for conducting and reporting case study research in software engineering. In: *Empirical software engineering* 14, pp. 131–164.

Sewell, Peter et al. (Jan. 2010). Ott: Effective tool support for the working semanticist. In: *Journal of Functional Programming* 20.1, pp. 71–122. ISSN: 09567968. DOI: 10.1017/S0956796809990293.

Souza Amorim, Luís 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 18*. Springer, pp. 1–23.

Van Antwerpen, Hendrik, 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.tudelft.idm.oclc.org/10.1145/3276484.

Van Antwerpen, Hendrik, Pierre Néron, et al. (Jan. 2016). A constraint language for static semantic analysis based on scope graphs. In: Association for Computing Machinery, Inc, pp. 49–60. ISBN: 9781450340977. DOI: 10.1145/2847538.2847543.

Visser, Eelco (2010). The Spoofax Language Workbench Rules for Declarative Specification of Languages and IDEs. In: *dl.acm.org*, pp. 444–463. DOI: 10.1145/1869459.1869497. URL: http://www.se.ewi.tudelft.nl/.

Visser, Eelco, Zine-el-Abidine Benaissa, and Andrew Tolmach (1998). Building program optimizers with rewriting strategies. In: *ACM Sigplan Notices* 34.1, pp. 13–26.