Embedding Statix in Agda

Version of August 21, 2024

Alex Haršáni

Embedding Statix in Agda

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Alex Haršáni
born in Trnava, Slovakia

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

Embedding Statix in Agda

Author: Alex Haršáni
Student id: 5064104

Abstract

Static type-checking allows us to detect ill-typed programs even before running them. However, the higher complexity of type systems may cause type-checker implementation to differ from their specifications. This causes bugs and makes it hard to reason about the type of systems. To close this gap between implementation and specification, a meta-language Statix was introduced. Using Statix, we can write a specification using constraints over scope graphs and terms. Successfully solving these constraints means that the program is well-typed. However, while Statix ensures that the implementation and specification correspond to each other, it does not offer a way for its users to formally reason about the type systems' specifications. To this end, we introduce a library called STATIX-IN-AGDA. This library, written using the proof assistant Agda, includes the formalisation of scope graphs and embedding of Statix's constraints. We show how we can use our library to specify the type system of STLC-like language, prove that programs in this language are well-typed, and give type-preservation proof for a type system of a simple toy language with numbers and addition.

Thesis Committee:

Chair:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	MSc. A.S. Zwaan, Faculty EEMCS, TU Delft
Committee Member:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. S. Dumančić, Faculty EEMCS, TU Delft

Preface

This thesis marks the culmination of my time as a student at TU Delft. Studying at this university has been a great privilege, and I will forever be grateful for all the knowledge and experience I gained throughout this journey.

Firstly, I would like to thank my supervisors Jesper Cockx, Aron Zwaan and Casper Poulsen. Thank you for all the help when I was stuck, for all your useful feedback that made me improve, and for the inspiration that made me choose the field of programming languages as the main focus of my studies.

Secondly, I would like to thank my friends, who made my time at TU Delft some of the best years of my life.

Last but not least, I would like to thank my family for their support and for always believing in me.

Alex Haršáni
Delft, the Netherlands
August 21, 2024

Contents

Chapter 1

Introduction

Static type-checking gives us a way to detect erroneous programs even before running them. For example, the ill-typed program that attempts to add a number and a string (e.g., `1 + "hello"`) will be rejected by the type-checker. This may save us a lot of time that we would otherwise spend looking for type-related bugs.

The problem with type systems is that they may become much more complex than just checking whether we add two numbers. In many programming languages, we often use variables and functions of which we cannot immediately say what type they have. When adding a number and a variable(e.g., `1 + x`), we need first to know whether the type of the variable is a number. This means that type-checking depends on the name resolution. Furthermore, things become even more complex in programs with non-lexical scoping, such as importing modules or using class inheritance. This way, not only does type-checking depends on the name resolution, but the name resolution also depends on type-checking. When type-checking a variable from an imported module, we must first resolve its name. However, in order to resolve its name, we need to type-check the imported module.

The growing complexity of the type systems creates a gap between their specification and implementation, which may lead to bugs. Antwerpen et al. 2018 give an example from the conversation under the pull request related to changes in Rust's name resolution, in which one of the contributors say:[1] "I'm finding it hard to reason about the precise model proposed here, I admit. I wonder if there is a way to make the write up a bit more declarative.". To close this gap, Antwerpen et al. 2018 introduced a metalanguage called Statix. With Statix, we can specify the type system using a set of constraints based on which the sound type-checker is derived. Name resolution is done using scope graphs, in which scopes are represented by nodes, with edges between them representing the scope inclusions. The names are resolved by traversing the graph to the required declaration.

This union between implementation and specification could allow us to reason about the type system, possibly creating a way to prove various useful properties of the language itself. However, since the specification is written using Statix, we cannot reason about it formally, in the same way as we could using proof assistants such as Agda, Idris, or Coq. To make this possible, we introduce a library called STATIX-IN-AGDA, the embedding of declarative semantics of Statix-core(in the rest of the thesis, we will refer to Statix-core as Statix), the subset of Statix defined by Rouvoet et al. 2020. Using this library, we can express Statix specifications in Agda and reason about them formally. Our vision for the future is to be able to automatically translate Statix specifications to their STATIX-IN-AGDA counterparts. However, for now, this is out of this project's scope. The complete implementation can be found in the GitHub repository[2].

[1] `https://github.com/rust-lang/rfcs/pull/1560`

[2] `https://github.com/AlexHarsani/statix-in-agda`

Research Questions The main goal of this thesis is to create an embedding of declarative semantics of Statix in Agda that users can use to reason about the correctness of their specifications. For this, we define the following research questions:

1. How can we formalise scope graphs in Agda, such that they can be used with the constraints of Statix? How does the latest definition of scope graphs differ from previous definitions?
2. What are the strengths and limitations of deep and shallow embedding styles of Statix constraints? What are the trade-offs regarding usability, correctness, and complexity of the embedding?
3. How does the design of the embedding influence the ability to specify type systems and reason about their meta-theoretical properties?

Contributions To answer the research questions, this thesis makes the following contributions:

- We introduce statix-in-agda, the embedding of Statix in Agda.
- We formalise scope graphs in Agda by extending previous implementations by Bach Poulsen, Rouvoet, et al. 2017 and Casamento 2019. We analyse and identify necessary changes and extensions to be made in order to make them usable in statix-in-agda. These changes include extending the implementation with labels representing scope inclusions and changing the scopes to include only one data term(in order for every declaration to have its own scope). We also extend the scope graph library with the scope graph fragments that are used for constraint support (Chapter 2, Chapter 4).
- We compare the deep and shallow styles of embedding in the context of Statix and discuss their advantages and disadvantages with regard to usability, soundness, and complexity of the implementation. We give deep embedding of Statix constraints (Chapter 3).
- We show how statix-in-agda can be used to specify the type system of language with variables and functions. We also show how statix-in-agda can be used to prove type-preservation property for a very simple language with numbers and addition (Chapter 5).

Chapter 2

Scope Graph

The first step in embedding Statix in Agda is formalizing scope graphs. Statix uses these graphs to perform name resolution during type-checking. In this chapter, we will give preliminary information on the scope graphs, examine problems in formalizing them in Agda, and give our implementation.

2.1 Preliminaries

Scope graphs represent a language-independent way to formally represent name binding and resolution. They were first introduced by Néron et al. 2015 and then Antwerpen et al. 2018 made them a fundamental part of the metalanguage Statix. Later, Rouvoet et al. 2020 extended them with the concept of critical edges in the graph, which gave a way to carefully schedule name resolution during type-checking of programs with non-lexical scoping.

Now, let us define the scope graph and other important terms that are essential for the rest of the paper:

- *Scope graph:* Consists of scope nodes connected by labeled edges.
- *Scope:* Is a region in the program where names are resolved uniformly, represented by a node in the scope graph.
- *Data term:* Contained within the scope, represents the type of declaration.
- *Edge:* Connects two scopes/nodes, is labeled.
- *Label:* Assigned to edges based on the scope inclusion that the edge represents.

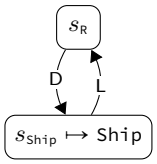
To demonstrate what the scope graph looks like, let us now go through its construction based on a simple Scala program. We start with a single scope of the program called the root scope in Figure 2.1.

Figure 2.1: Scope graph with only root scope.

First, we declare an object called Ship. Every declaration introduces its own node in a scope graph. This object was declared in the root scope, therefore it is a lexical child of the the root scope. So, we add an edge labeled L(lexical parent), going from child scope to parent scope. Also, we added an edge labeled D(declaration), going from parent scope to child scope. We get a program and a scope graph as in Figure 2.2.

Then, we add some sails and engines to the ship. Let us define methods representing the number of sails and engines on the ship. In Figure 2.3, we can see that these methods

```
1  object Ship {
2
3  }
```

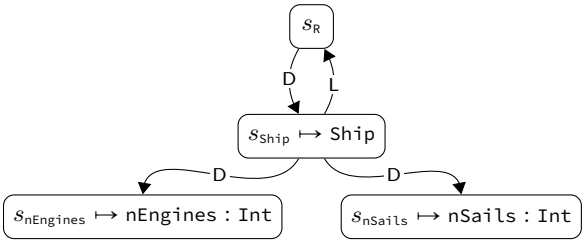


(a) Scala code (b) Scope graph

Figure 2.2: Scope graph after adding the object `Ship`.

introduced two more new scopes, with edges labeled D pointing to them from scope of the `Ship` object.

```
1  object Ship {
2      def nEngines:Int = 1;
3      def nSails:Int = 2;
4  }
```

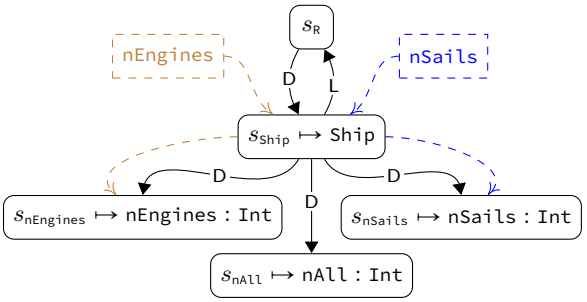


(a) Scala code (b) Scope graph

Figure 2.3: Scope graph after adding the methods for numbers of engines and number of sails.

Finally, to demonstrate how names are resolved, we introduce a method representing the total number of both engines and sails. For this method, we need to resolve the names of the previous two methods. To do this, we need to define notions of *reachability* and *visibility*. Reachability is defined with regard to a scope s, a word r, and a predicate on a scope data term D. A scope s' is reachable from scope s if predicate D holds for its data term(of scope s') through all the paths in the graph according to a word r. A word r is a regular expression that defines the sequence of labels we allow in the path. This will enable us to restrict what paths through the graph we can take. For example, we can allow the path to only take a maximum of one edge with a I label to disallow transitive imports. Conversely, visibility defines which paths are minimal according to a preorder relation R. The resolution paths can be seen in Figure 2.4.

```
1  object Ship {
2      def nEngines:Int = 1;
3      def nSails:Int = 2;
4      def nAll:Int = num_of_engines +
5                          num_of_sails;
6  }
```



(a) Scala code (b) Scope graph

Figure 2.4: Scope graph after adding the method for numbers of engines and sails, along with the dashed resolution path for variables `nSails` and `nEngines`.

2.2 Problem

Scope graphs have already been implemented in Agda by Bach Poulsen, Rouvoet, et al. 2017. They were defined as a function from scope to a pair of lists of declarations(data terms) and neighboring scopes/nodes. The scope is represented by `Fin k`, a type of finite natural numbers up to k. Casamento 2019 further generalises this implementation by replacing `Fin` with `Listable`, which allows for improved generating of scope graphs, as we do not need to know a number of scopes beforehand. The implementation based on the work of Bach Poulsen, Rouvoet, et al. 2017 can be seen in Figure 2.5.

```
1  Scope = Fin k
2
3  Graph = Scope → (List Ty × List Scope)
```

Figure 2.5: Scope graph in Agda based on the work of Bach Poulsen, Rouvoet, et al. 2017.

The main problem we have to solve with regard to scope graphs is adapting their implementation according to Statix's requirements. The first problem is to allow for easy querying and traversal in the graph, as some of the constraints, as we will see in the next chapter, require. Luckily, the previous implementations of scope graphs work quite nicely thanks to their functional representation. The second problem is that in the newer publications, such as by Rouvoet et al. 2020, the way that scope graphs are modeled slightly changed. The first difference is that the edges between scopes are labeled. This way, declarations become edges labeled to represent declarations. These labels also give more options to model more complex scope inclusions and restrict the resolution paths. Another difference is how the declarations are modeled. In the old model, the declarations are represented by special types of nodes with edges connecting them to the scope node. In the new model, declarations instead introduce their own scope. The difference between the two versions of the scope graph corresponding to the code in Figure 2.6 can be seen in Figure 2.7.

2.3 Implementation

Let us now address the problems mentioned in the previous section. The most crucial aspect we require from our embedding of scope graphs is easy traversal. In other words, we quickly want to be able to access the scope's neighbors and data. As we have seen in the previous section, the solution to this problem is to make the scope graph a function from scope to data included in its scope. Another critical design choice is how to represent scopes. As shown by Casamento 2019, `Fin` type is not a suitable option when generating scope graphs. However, as we will see later in Chapter 5, the scope graphs are not generated but given beforehand as input to the proof. Therefore, for the convenience of use, we chose `Fin` type to represent

```
class A { int x; /* ... */ }
class B extends A {
  public void m(int i, A a) {
    x = i;
    /* ... */
  }
}
```

Figure 2.6: Java program from the Figure 5 from Bach Poulsen, Rouvoet, et al. 2017, used with the permission from the author.

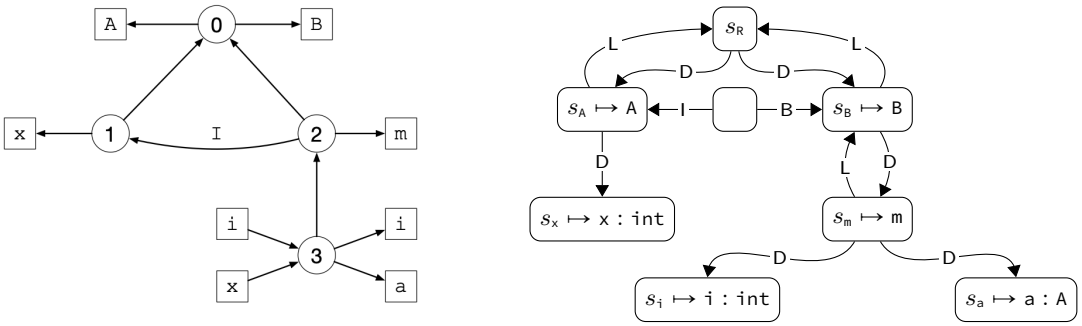


Figure 2.7: Old(left) and new(right) scope graph representation of the program in Figure 2.6. The old representation is from the Figure 5 from Bach Poulsen, Rouvoet, et al. 2017, used with the permission from the author.

scopes. We also need to facilitate the changes in the model of the scope graphs discussed in the previous section. Namely, we need to add labels to the scope's outgoing edges, as well as allow the scope to include only one data term. This way, we get the representation in Figure 2.8. Using this representation, we can define a scope graph from Figure 2.4b in Agda, as can be seen in Figure 2.9.

```
ScopeData : Nat → Set → Set
ScopeData numberOfScopes Term = (List (Label × (Fin numberOfScopes))) × Term

ScopeGraph : Nat → Set → Set
ScopeGraph numberOfScopes Term = (Fin numberOfScopes) → ScopeData numberOfScopes Term
```

Figure 2.8: Scope graph in Agda.

```
scope-graph-ship : ScopeGraph 5 Type
scope-graph-ship zero = (d , suc zero) :: [] , noType
scope-graph-ship (suc zero) =
    (l , zero) ::
    (d , suc (suc zero)) ::
    (d , suc (suc (suc zero))) ::
    ((d , suc (suc (suc (suc zero)))) :: [] , Ship
scope-graph-ship (suc (suc zero)) = [] , num
scope-graph-ship (suc (suc (suc zero))) = [] , num
scope-graph-ship (suc (suc (suc (suc zero)))) = [] , num
```

Figure 2.9: Agda representation of scope graph from Figure 2.4b.

Chapter 3

Constraints

In this chapter, we extend our library with constraints, the building blocks of Statix for specifying type systems. First, We will describe the types of constraints in Statix, how they can be used to specify a type system, and how Statix type-checks the program by solving the constraints. Then, we will compare different styles of embedding for the constraints, discussing their strengths and weaknesses. Finally, we will show our implementation of constraints.

3.1 Preliminaries

In Statix, we have three groups of constraints. There are basic constraints, scope graph constraints, and user-defined constraints. The first is a group of constraints, possibly independent from the scope graph, that are defined over data terms based on a target language. These are, for example, *term equality* constraints or *separating conjunction* constraints. The second group is constraints over the scope graph, such as scope graph *queries* or *node assertion* constraints. The last group, the user-defined constraints, is defined by users of Statix and is defined in terms of the previous two groups of constraints. Now, let us explore the constraint groups in more detail.

Basic Constraints

- Emp : Holds trivially.
- $False$: Does not hold.
- $Conj\ C_1\ C_2$: Holds, if and only if both C_1 and C_2 hold.
- $Eq\ t_1\ t_2$: Holds, if and only if t_1 and t_2 are equal.
- $Exists\ t\ C$: Holds, if there exists a term t, that makes C hold.
- $Forall\ ts\ C$: Holds, if and only if all terms in ts make C hold.
- $Single\ t\ ts$: Holds, if and only if terms ts are a singleton set of only t.

Scope Graph Constraints

- $Node\ s\ t$: Holds, if and only if scope s is in the scope graph and contains data term t.
- $Edge\ e$: Holds, if and only if edge e is in the scope graph.
- $Data\ s\ t$: Holds, if and only if scope s contains data term t. Similar to $Node$, but as we will see in the next chapter, it is supported by empty fragment of scope graph, unlike $Node$ constraint.

- $Query\ s\ r\ D\ C$: Finds all paths according to regular word r from scope s to any scope with data term according to predicate D, and holds if applying these paths to constraint C holds.

- $Min\ p\ p'\ R$: Holds, if and only if set of paths p' only contains minimal paths from p, according to definition in Equation 3.1 and some preorder relation R.

$$min(A, R) = \{p \in A | \forall q \in A. Rqp \Rightarrow Rpq\} \tag{3.1}$$

User-defined Constraints

To show an example of a user-defined constraint, let us create a Statix specification. We define a simple language with grammar from Figure 3.1. We also define a set of typing rules from Figure 3.2.

⟨*num*⟩ := N

⟨*bool*⟩ := true | false

⟨*expr*⟩ ::= num | bool
| ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ > ⟨*expr*⟩
| '`if`' ⟨*expr*⟩ '`then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩

⟨*type*⟩ ::= number | boolean

Figure 3.1: Grammar of the simple language with numbers, booleans, addition, comparisons, and conditional statements.

$$\frac{}{\Gamma \vdash n : \text{Int}}\ \text{(T-Num)} \qquad \frac{}{\Gamma \vdash \text{true} : \text{Bool}}\ \text{(T-Bool-True)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ \text{(T-Bool-False)} \qquad \frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}\ \text{(T-Add)}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 > e_2 : \text{Bool}}\ \text{(T-Gt)} \qquad \frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}\ \text{(T-If)}$$

Figure 3.2: Typing rules for a simple language with numbers, booleans, addition, comparisons, and conditional statements.

Now that we have defined our language, we can write a Statix specification of its type system. We define a *user-defined constraint* `typeOfExpression` with cases for each type of the expression. First, we start with trivial cases for numbers and booleans. The type of `num` is `number`, and the type of `bool` is `boolean`. We define these cases of user-defined constraint in terms of basic constraint *term equality*. Next, we add cases for our binary expressions. In both of them, we expect the operands to be numbers. Additionally, we expect addition to be of type number, and greater-than to be of type boolean. For these cases, we use a combination of basic constraints of *term equality* and a *separating conjunction*. Finally, we create a similar case for if-condition. The user-defined constraint `typeOfExpression` can be seen in Figure 3.3.

```
typeOfExpression(s, num, T)     = T ≡ number
typeOfExpression(s, bool, T)    = T ≡ boolean
typeOfExpression(s, e1 + e2, T) = typeOfExpression(s, e1, number) *
                                  typeOfExpression(s, e2, number) *
                                  T ≡ number
typeOfExpression(s, e1 > e2, T) = typeOfExpression(s, e1, number) *
                                  typeOfExpression(s, e2, number) *
                                  T ≡ boolean
typeOfExpression(s, if cond then e1 else e2, T) =
                                  typeOfExpression(s, cond, boolean) *
                                  typeOfExpression(s, e1, T) *
                                  typeOfExpression(s, e2, T)
```

Figure 3.3: User-defined constraint for the type system of the programming language from Figure 3.1. As we can see, there is a close resemblance to typing rules from Figure 3.2.

3.1.1 Type-checking with Statix

Based on the constraints we defined, Statix can perform the type-checking. To do this, it creates a single constraint that corresponds to the conjunction of all of our constraints. This constraint C, in addition to an empty scope graph ϵ, forms a starting state $\langle \epsilon, C \rangle$. From the starting state, a constraint from C is picked non-deterministically and solved according to operational semantics [Rouvoet et al. 2020; Antwerpen et al. 2018]. During constraint solving, the scope graph is built until the state eventually reaches $\langle G, \varnothing \rangle$, which means the constraints were solved and the program successfully type-checks. If the state reaches $\langle G, false \rangle$, the constraints are rejected, and the program does not type-check. If the state reaches a point where it cannot be further reduced, it is stuck.

3.2 Problem

Embedding a domain-specific language, such as Statix, requires us to choose the depth of the embedding. In the *deep embedding*, we represent the embedded language's terms by their syntax. The semantics of the embedded language are specified in some evaluation function that traverses the abstract syntax tree. On the other hand, in the *shallow embedding*, the terms are represented directly by their semantics, taking more significant advantage of the host's language [Gibbons and Wu 2014]. The choice of depth is not limited to only one of these approaches, but they can also be combined for various parts of the language.

3.2.1 Shallow Embedding

In the first approach, we use the shallow embedding to represent constraints. In Figure 3.4, we can see that a constraint is a function from the scope graph to `Set`. This means that the constraint itself is defined by its satisfiability(`Set`). The definition of some of the individual constraints works quite naturally in Agda, as they nicely resemble available definitions that are part of the host language's library. `EmpC` and `FalseC` to unit and empty type, respectively, `EqC` to identity type, `ExistsC` to sigma type and `*C` to a pair type.

```
1 Constraint : Nat → Set → Set₁
2 Constraint numberOfScopes Term = ScopeGraph numberOfScopes Term → Set
```

Figure 3.4: Shallow embedding of Statix's constraints.

While this approach works well in practice, it unfortunately allows for creating new constraints that could compromise the soundness of statix-in-agda. Users could, for example, create constraints that potentially break the support property(constraint support will be discussed in detail in the next chapter), resulting in successful type-checking of ill-typed programs. That is why we would only like the user to use the predefined set of constraints or their composition.

3.2.2 Deep Embedding

To mitigate the issue of the previous approach, we can define the syntax of constraints separately from their semantics. We do this by defining an Agda data type to represent the syntax of the constraints, with constructors for each of the specific constraints. Satisfiability is defined as a separate function. In Figure 3.5, we can see that the function `sat` closely resembles the definition of constraints themselves in the previous approach.

This approach has the advantage in that it restricts users to only using the predefined set of constraints or combining them to introduce their own. On the other hand, deep embedding has the disadvantage that it introduces an additional layer of complexity on top of Agda, which makes it harder to work with.

```
1  data Constraint {numberOfScopes : Nat} {Term : Set} (g : ScopeGraph numberOfScopes Term)
2      : Set₁ where
3      ...
4
5  sat : {numberOfScopes : Nat} {Term : Set} (g : ScopeGraph numberOfScopes Term) →
6      Constraint g → Set
7  sat = ...
```

Figure 3.5: Deep embedding of Statix's constraints.

3.3 Implementation

In statix-in-agda, we choose the deep embedding for the constraints. As we see in the previous section, the reason for not choosing a shallow embedding is that it allows users of statix-in-agda to freely create new constraints, which could compromise soundness. Instead, we opt for defining constraints as constructors of a constraint data type. Additionally, we need to define a function `sat` that defines the satisfiability for each of the constraints. For user-defined constraints, we define them as a function that returns basic or scope graph constraints. This way, the user can only create new user-defined constraints that are a combination of pre-defined constraints. The outline of the deep embedding, as well as the example of user-defined constraint, can be seen in Figure 3.6. Now, let us take a look at specific constraints.

3.3.1 Basic Constraints

For `EmpC` and `FalseC`, the satisfiability is defined using Unit($\top$) and Empty($\bot$) type, respectively. The `EqC` constraint is satisfied if we can prove that the supplied terms are equal. Similarly, `SingleC` is satisfied if the supplied list of terms is a singleton list containing only the other supplied term. The cases of the `sat` function for these constraints can be seen in Figure 3.7.

Next, we have the constraint `*C`, also called the separating conjunction. In order for this constraint to hold, both of the conjugated constraints must hold, as can be seen in Figure 3.8.

```
data Constraint {numberOfScopes : Nat} {Term : Set}
  (g : ScopeGraph numberOfScopes Term) : Set₁ where
    -- true
    EmpC : Constraint g
    -- separating conjunction
    _*C_ : (c1 : Constraint g) → (c2 : Constraint g) → Constraint g
    ...

UserDefinedC : {numberOfScopes : Nat} {Term : Set}
  (g : ScopeGraph numberOfScopes Term) → Constraint g
UserDefinedC g = EmpC *C EmpC

sat : {numberOfScopes : Nat} {Term : Set}
    (g : ScopeGraph numberOfScopes Term) →
    Constraint g → Set
sat g c = ...
```

Figure 3.6: Deep embedding of Statix's constraints allows a user to only create constraint such as `UserDefinedC`, that are a combination of pre-defined constraints.

```
-- EmpC
sat g EmpC = ⊤
-- FalseC
sat g FalseC = ⊥
-- EqC
sat g (EqC t1 t2) = (t1 ≡ t2)
-- SingleC
sat g (SingleC t ts) = ((t ∷ []) ≡ ts)
```

Figure 3.7: Satisfiability of basic constraints.

```
-- *C
sat g (c1 *C c2) = (sat g c1) × (sat g c2)
```

Figure 3.8: Satisfiability of the constraint conjunction.

The last two basic constraints, `ExistsC` and `ForallC`, as their names suggest, resemble logical quantifiers. For the former, we use the sigma type(or dependent pair) that is used in Agda to represent the existential quantifier. For the proof of $ExistsC\ c$ with some constraint c, this means that we have to give a term for which the constraint c is satisfied. On the other hand, the `ForallC` works a bit differently. We do not quantify on all possible terms, but just on the list of terms we pass as an argument. For example, in $ForallC\ ts\ c$, with ts being a list of terms and c being a constraint, we assert that the constraint c must be satisfied with each term in ts. One notable difference between our embedding and Statix is that we use a list of terms instead of a set in `ForallC` and `SingleC` constraints for simplicity of use. This is not a problem since the ordering of terms does not change how the constraints works. Cases of `sat` function for `ExistsC` and `ForallC` can be seen in Figure 3.9.

3.3.2 Scope Graph Constraints

The next group of constraints are those that make assertions about the scope graph itself. Firstly, we have in Figure 3.10 , the more trivial constraints, `NodeC`, `EdgeC` and `DataC`. $NodeC\ s\ t$, asserts that the scope graph contains scope s, and that the term t is contained within the node of scope s. $EdgeC\ e$, asserts the presence of edge e in the scope graph. The constraint $DataC\ s\ t$ asserts that a term t is in the scope s.

```
-- ExistsC
sat g (ExistsC {Term} cf) = Σ Term λ t → (sat g (cf t))
-- ForallC
sat g (ForallC [] cf) = sat g EmpC
sat g (ForallC (t :: ts) cf) = sat g (cf t) × sat g (ForallC ts cf)
```

Figure 3.9: Satisfiability of exists and forall constraints. In forall, we recursively use similar technique to conjunction constraint from Figure 3.8.

```
-- NodeC
sat g (NodeC s t) = decl (g s) ≡ t
-- EdgeC
sat g (EdgeC e@(s₁ , l , s₂)) = (l , s₂) ∈ edges (g s₁)
-- DataC
sat g (DataC s t) = decl (g s) ≡ t
```

Figure 3.10: Satisfiability of basic scope graph constraints. In `NodeC` and `DataC`, we use function `decl`, which projects the data term from scope s. In the `EdgeC` constraint, we check whether the scope s_2 is a neighbor of the scope s_1 through some edge with a label l(this is some arbitrary label, not to be confused with label L that stands for lexical parent).

The constraint $MinC\ p\ p'\ R$ asserts that the list of paths p' contains only the shortest paths from p, according to the decidable relation R. Figure 3.11 shows the satisfiability of `MinC`, as well as the Agda implementation of the `min` property mentioned in equation 3.1.

```
-- MinC
sat g (MinC paths paths' R? isPreorder) = (min R? paths paths ≡ paths')

...

min : {numberOfScopes : Nat} {Term : Set} {g : ScopeGraph numberOfScopes Term}
  {R : (Rel (Path g) Agda.Primitive.lzero)} →
  Decidable R → (A A' : List (Path g)) → List (Path g)
min R? [] A' = []
min R? (p :: A) A' =
    if isMin R? A' p then p :: min R? A A' else min R? A A'

isMin : {numberOfScopes : Nat} {Term : Set} {g : ScopeGraph numberOfScopes Term}
  {R : (Rel (Path g) Agda.Primitive.lzero)} →
  Decidable R → (A : List (Path g)) → Path g → Bool
isMin R? [] p = true
isMin R? (q :: A) p = if does (R? q p)
    then if does (R? p q) then isMin R? A p else false
    else isMin R? A p
```

Figure 3.11: In function $isMin$, we iterate through the paths in the list to check, whether the path p is smaller than or equal to all other paths, according to decidable relation R.

The very last constraint is the `QueryC`. There are two ways to implement this constraint in Agda. We can either algorithmically calculate all paths in the graph to desired scopes, or we could use a declarative way by already supplying a list of paths and proving that certain properties that make the path valid hold for them. In this thesis, we chose the latter way for two reasons. For one, it makes proving the satisfiability of the `QueryC` constraint easier. Here, one might ask how we know which paths to supply. This brings us to the second reason. In our vision for the future Statix-in-Agda, users will be able to input paths computed by Statix

automatically by using a translator to Agda. For now, though, paths have to be computed externally and then manually added.

Let us now define the following properties that need to be true for each of the supplied paths:

- Well-formed: the path should exist in the scope graph.
- Non-cyclic: the path should not consist of cycles.
- Valid start and end: the path should start in a specified scope and end in any scope that contains a term for which the specified predicate holds.
- Allowed: the path should be allowed according to the predicate specified by the user.

Additionally, no other paths adhering to the previously mentioned properties should be missing from the list of the supplied paths. Based on these properties, we define a record called `ValidQuery` as in Figure 3.12.

```
record ValidQuery {numberOfScopes : Nat} {Term : Set}
  (g : ScopeGraph numberOfScopes Term) (s : Fin numberOfScopes)
  (r : (Path g) → Set) (D : Term → Set) : Set where
    constructor query-proof
    field
        paths           : List (Path g)
        well-formed     : ∀ {path} → path ∈ paths → validPath g path
        non-cyclic      : ∀ {path} → path ∈ paths → noCycle path
        allowed         : ∀ {path} → path ∈ paths → r path
        valid-start     : ∀ {path} → path ∈ paths → firstScope path ≡ s
        valid-end       : ∀ {path} → path ∈ paths → validEnd g D path
        no-path-missing : ∀ {path} →
                          noCycle path →
                          r path →
                          validPath g path →
                          firstScope path ≡ s →
                          validEnd g D path →
                          path ∈ paths

```

Figure 3.12: Record containing properties that make the query valid. In `well-formed`, we prove that all the paths in `paths` are in the graph. In `non-cyclic`, we prove that all the paths in `paths` are non-cyclic. In `allowed`, we prove that all the paths in `paths` are allowed with respect to user-supplied property `r`. In `valid-start` and `valid-end`, we prove that all the paths in `paths` start in scope `s` and end in scopes for which the predicate `D` holds. Finally, in `no-path-missing`, we prove that `paths` includes all possible paths for which the previous properties hold.

Using the `ValidQuery`, we can define a `sat` case for `QueryC`. There is an additional argument to `QueryC` in the form of another constraint, to which we apply the list of paths from `ValidQuery`. This can be seen in Figure 3.13.

```
-- QueryC
sat g (QueryC s r D cf) =
    Σ (ValidQuery g s r D) (λ s → (sat g (cf (ValidQuery.paths s))))
```

Figure 3.13: Satisfiability of the query constraint.

3.3.3 User-defined Constraints

User-defined constraints give users a way to compose basic and scope graph constraints to create their type system specifications. Essentially, every constraint of this type resembles a typing rule. In Chapter 5, we will see an example of such a user-defined constraint that will be used to specify type-system for a simple language with variables and functions.

Chapter 4

Constraint Support

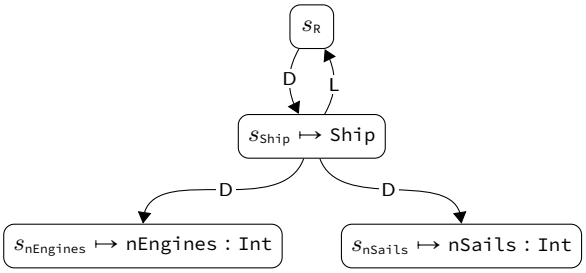
In this chapter, we show the final piece in the STATIX-IN-AGDA library, the constraint support. We will see what the support is and why it is essential for sound type-checking. Then, we will discuss the challenges in embedding constraint support and show the implementation.

4.1 Preliminaries

Each of the constraints is supported by a pair $< n, e >$ of possibly empty sets of nodes n and edges e. In this thesis, we call this pair a graph fragment. The purpose of these fragments is to ensure the soundness of type-checking. The fragment of the top-level user-defined constraints needs to contain all the nodes and edges of the scope graph. This property enforces that we cannot just use any scope graph to force the program to type-check. In Figure 4.1, we can see an ill-typed program and an unsupported scope graph. This scope graph includes the "junk" scope for variable `nSails` that is undefined in the program. With this scope graph, we could successfully solve the constraints that type-check the program.

```
1  object Ship {
2      def nEngines:Int = nSails;
3  }
```

(a) Scala code



(b) Scope graph

Figure 4.1: On the left side of the figure, we can see an ill-typed program. For an unsupported scope graph, such as the one on the right of the figure, we could force the ill-typed program type-check.

In Statix, the constraints are supported by the following graph fragments:

- Emp, $False$, $Eq\ t_1\ t_2$, $Single\ t\ ts$, $Data\ s\ t$, $Min\ p\ p'\ R$: Supported by an empty scope graph fragment.
- $Conj\ C_1\ C_2$: Supported by disjoint union of supports for $C1$ and $C2$
- $Exists\ t\ C$: Supported by the fragment supporting C.
- $Forall\ ts\ C$: Supported by the disjoint union of fragments supporting C for each of the ts.

- $Node\ s\ t$: Supported by a graph fragment containing scope s.
- $Edge\ e$: Supported by a graph fragment containing edge e.
- $Query\ s\ r\ D\ C$: Supported by the fragment supporting C.

4.2 Problem

In our implementation, we have to address a few problems related to constraint support. The main problem is how to model the support in relation to satisfiability. As we can see in Rouvoet et al. 2020, the satisfiability of constraint is defined with regard to the support.

Another challenge that comes with the fragments is their invariant of disjointedness. When fragments are merged(for example, in conjunction or for constraint), the scopes within the merged fragments must only be in either of the fragments.

Finally, we need a way to prove that the scope graph fragment of the top-level fragment is the same as the scope graph.

4.3 Implementation

While it seems like using the same functional representation for scope graph fragments, as we did for scope graphs, is the obvious next step, we have to choose a different model. Unlike scope graphs, we do not need to traverse fragments. On the other hand, we would like them to be easily created and merged together. For this, a list-like representation works much better. Figure 4.2 shows a scope graph fragment represented as a record type.

```
1 record ScopeGraphFragment {numberOfScopes : Nat} {Term : Set}
2     (g : ScopeGraph numberOfScopes Term) : Set where
3     constructor <_,_>
4     field
5         fragmentNodes : List (Fin numberOfScopes)
6         fragmentEdges : List (Edge g)
```

Figure 4.2: Scope graph fragment in Agda.

During constraint solving, more precisely in conjunction and forall constraints, the fragments supporting the sub-constraints are merged together. There is, however, a property that needs to hold during merging, which is the disjointedness of scopes in two fragments. In simpler terms, if scope "a" is in fragment "1", it cannot be also in fragment "2". This does not hold for edges, as they can be in both fragments[1]. The disjointedness property can be seen in Figure 4.3.

Now that we have defined scope graph fragments and the disjointedness property, we can incorporate it into the definition of satisfiability. Based on the definition in Rouvoet et al. 2020, a constraint is only satisfied with respect to the scope graph support. Therefore, we change the `sat` function to return a dependent pair type of the satisfiability property and the scope graph fragment that supports the constraint. In Figure 4.4, we can see the new definition of the `sat` function, as well as the cases for `EmpC` and `*C` constraints.

Finally, in order for the whole scope graph to support the top-level constraint, we need a way to check whether the top-level fragment matches the scope graph. As we have seen before, scope graphs are functions from scope to scope data, while fragments are list of scopes and edges. In order to be able to compare these, we define a helper function called

[1]While this is not true in Rouvoet et al. 2020, it holds for Java implementation of Statix: `https://github.com/metaborg/nabl/tree/master`.

```
data DisjointGraphFragments {numberOfScopes : Nat} {Term : Set}
    {g : ScopeGraph numberOfScopes Term}
    (gf1 : ScopeGraphFragment g) : ScopeGraphFragment g → Set where
        disjointEmpty : ∀ {edges} → DisjointGraphFragments gf1 < [] , edges >
        disjointNonEmpty : {nodes : List (Fin numberOfScopes)} {edges : List (Edge g)}
            {scope : Fin numberOfScopes} →
            (scope ∉ (ScopeGraphFragment.fragmentNodes gf1)) →
            DisjointGraphFragments gf1 < nodes , edges > →
            DisjointGraphFragments gf1 < scope ∷ nodes , edges >
```

Figure 4.3: Disjointedness property of fragments.

```
sat : {numberOfScopes : Nat} {Term : Set}
    (g : ScopeGraph numberOfScopes Term) →
    Constraint g →
    (Σ Set λ s → (s → ScopeGraphFragment g))
-- EmpC
sat g EmpC = ⊤ , λ _ → empGf
-- *C
sat g (c1 *C c2) = Σ (proj₁ c1-sat × proj₁ c2-sat)
                     (λ (c1-proof , c2-proof) →
                         DisjointGraphFragments (proj₂ c1-sat c1-proof)
                                                (proj₂ c2-sat c2-proof)),
                 λ ((c1-proof , c2-proof) , disjoint) →
                     mergeFragments (proj₂ c1-sat c1-proof)
                                    (proj₂ c2-sat c2-proof)
  where
    c1-sat = sat g c1
    c2-sat = sat g c2

satisfies = proj₁
fragment = proj₂
```

Figure 4.4: Function `sat` with the scope graph support. We also create aliases for projection functions to make projecting satisfiability and fragments more readable.

`functionToFragment`, which converts the functional representation of scope graphs to fragment representation. This way, we check whether the scope graph is the same as the top-level fragment. One of the downside of using list instead of set for graph fragments, is that we have to use whether list permutations are the same, as the order does not matter. The `validTopLevelGraphFragment` property is shown in Figure 4.5.

```
validTopLevelGraphFragment : {numberOfScopes : Nat} {Term : Set}
  {g : ScopeGraph numberOfScopes Term} →
  (c : Constraint g) → (c-proof : proj₁ (sat g c)) → Set
validTopLevelGraphFragment {g = g} c c-proof =
    ((ScopeGraphFragment.fragmentNodes (proj₂ (sat g c) c-proof)) ↔
     (ScopeGraphFragment.fragmentNodes (functionToFragment g))) ×
     (ScopeGraphFragment.fragmentEdges (proj₂ (sat g c) c-proof) ↔
     (ScopeGraphFragment.fragmentEdges (functionToFragment g)))
```

Figure 4.5: Property that top-level fragment matches the scope graph. As the ordering of scopes and edges in the fragment does not matter, we check whether they are permutations(↔) of each other.

Chapter 5

Examples

In the following chapter, we show two examples of STATIX-IN-AGDA's use. In the example in Section 5.1, we implement a type system for a simple language with variables and functions and show how to type-check programs. In the second example in Section 5.2, we prove type preservation for a simple toy type system without variables and functions.

5.1 Implementing simple type system

Let us extend the language from Figure 3.1 with functions, function applications, variables, and let bindings(for conciseness, we also remove if-statements and greater-than operator). Let us define the grammar of this language in Figure 5.1 and typing rules in Figure 5.2.

$\langle num \rangle$:= N

$\langle bool \rangle$:= true | false

$\langle expr \rangle$ ::= num | bool
| $\langle expr \rangle$ + $\langle expr \rangle$
| 'fun (' $\langle ident \rangle$ ' : ' $\langle type \rangle$ ') = ' $\langle expr \rangle$
| 'var' $\langle ident \rangle$
| 'app' $\langle expr \rangle$ $\langle expr \rangle$
| 'let' $\langle ident \rangle$ '=' $\langle expr \rangle$ 'in' $\langle expr \rangle$

$\langle type \rangle$ ::= number | boolean | $\langle type \rangle$ -> $\langle type \rangle$

Figure 5.1: Grammar of the language with numbers, booleans, addition, functions, variables, function application, and let-binding.

$$\frac{}{\Gamma \vdash n : \text{number}}\ \text{(T-Num)} \qquad \frac{}{\Gamma \vdash \text{true} : \text{boolean}}\ \text{(T-Bool-True)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{boolean}}\ \text{(T-Bool-False)} \qquad \frac{\Gamma \vdash e_1 : \text{number} \qquad \Gamma \vdash e_2 : \text{number}}{\Gamma \vdash e_1 + e_2 : \text{number}}\ \text{(T-Add)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}(x : \tau_1) = e : \tau_1 \to \tau_2}\ \text{(T-Fun)} \qquad \frac{}{\Gamma \vdash \text{var}\ x : \Gamma(x)}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{app}\ e_1\ e_2 : \tau_2}\ \text{(T-App)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let}\ x = e_1\ \text{in}\ e_2 : \tau_2}\ \text{(T-Let)}$$

Figure 5.2: Typing rules of the language with numbers, booleans, addition, functions, variables, function application, and let-binding.

5.1.1 Specification

Now that we have the necessary foundation, we can start building our statix-in-agda specification. Let us create a user-defined constraint `typeOfExpression`.

Root. First, we use the `NodeC` constraint to define the root scope of our program. This is the top-level scope that is a lexical parent for all of the scopes in the rest of our program.

```
typeOfExpression' : {numberOfScopes : Nat} → (g : ScopeGraph numberOfScopes NodeTerm) →
    (Fin numberOfScopes) → Expr → Type → Constraint g
typeOfExpression' = {! !}

typeOfExpression : {numberOfScopes : Nat} → (g : ScopeGraph numberOfScopes NodeTerm) →
    (Fin numberOfScopes) → Expr → Type → Constraint g
typeOfExpression g s e t = NodeC s empNode *C typeOfExpression' g s e t
```

Literals and addition. For the first three types of expressions, user constraints in statix-in-agda are rather simple. For literals, we just check if their types are as expected using the `EqC` constraint. In addition, we check whether expressions on both sides of the operator are numbers and finally check if the whole expression is a number.

```
typeOfExpression' g s (numLit x) t = EqC num t
typeOfExpression' g s (boolLit x) t = EqC bool t
typeOfExpression' g s (e1 +' e2) t = EqC num t *C
    (typeOfExpression' g s e1 num *C typeOfExpression' g s e2 num)
```

Functions, their application and let binding. For the rest of the user constraints, it gets slightly more complicated. The *function* constraint comes down to two main aspects. Building the necessary structure of graph fragments and verifying whether the function and its body are of the correct type. The former means that we have to use `NodeC` constraints to build function scope and argument scope and connect them with edges using `EdgeC` constraints. For *function applications*, we just verify whether the first expression is of a function type and whether the second expression has the correct type as per the function argument. *Let* binding constraints work very similarly to functions, with only one exception. While in *functions*, we only type-check the body, here we also have to type-check the expressions that are bound to the variable.

```
typeOfExpression' g s (fun< x of t1 >body body) t = ExistsC λ t2 → ExistsC λ sf →
    (NodeC sf empNode) *C
    (EdgeC (s , d , sf) *C (EqC t (t1 to t2) *C
    ((ExistsC λ sx → EdgeC (sf , d , sx) *C NodeC sx (var x |' t1)) *C
    (EdgeC (sf , l , s) *C typeOfExpression' g sf body t2))))
typeOfExpression' g s (fun e1 app e2) t2 = ExistsC λ t1 →
    typeOfExpression' g s e1 (t1 to t2) *C
    typeOfExpression' g s e2 t1
typeOfExpression' g s (lett x be e1 inn e2) t2 = (ExistsC λ t1 → ExistsC λ sb →
    (NodeC sb empNode) *C
    (EdgeC (s , d , sb) *C (typeOfExpression' g s e1 t1 *C
    (ExistsC (λ sx → EdgeC (sb , d , sx) *C NodeC sx (var x |' t1)) *C (EdgeC (sb , l , s) *C
    typeOfExpression' g sb e2 t2)))))
```

Variables. For type-checking variables, our user constraint needs to use the `QueryC` constraint to retrieve the path to the queried variable. Since we only want one path, we also use the `SingleC` constraint to assert that the query yields only one path.

```
typeOfExpression' g s (var x) t = QueryC s (λ _ → ⊤) (λ d → d ≡ (var x | t)) λ paths →
    ExistsC λ path → SingleC path paths
```

5.1.2 Type-checking of programs

The type-checking proof for a given program consists of two steps. In the first step, we show that the user-defined constraints are satisfied. If the constraints are satisfied, we proceed to the second step, where we show that the top-level graph fragment is valid with regard to the scope graph. To encapsulate these two steps, we define a property `topLevelConstraintTypeCheck`, as in Figure 5.3. If the user wants to write a type-checking proof, they have to prove this property. The examples of such proofs were left out of this paper for conciseness, as they grow quite large, but they can be found in the project repository.

```
1 topLevelConstraintTypeCheck : {numberOfScopes : Nat} {Term : Set} →
2     (g : ScopeGraph numberOfScopes Term) → (Constraint g) → Set
3 topLevelConstraintTypeCheck g c =
4     Σ (satisfies (sat g c)) λ c-proof → validTopLevelGraphFragment c c-proof
```

Figure 5.3: Property that encapsulates properties of satisfiability and valid top-level fragment.

5.2 Proving type preservation

Type preservation is a meta-theoretical property that states evaluating an expression does not change its type. More formally, let e_1 and e_2 be expressions, τ a type, Γ a context, and $\rightarrow$ an evaluation relation. If $\Gamma \vdash e_1 : \tau$ and $e_1 \rightarrow e_2$, then it holds that $\Gamma \vdash e_2 : \tau$.

Now, let us define a STATIX-IN-AGDA specification for a very simple language with only number literals and addition as in Figure 5.4. Additionally, we need to define small-step operational semantics. In Figure 5.5, we define a function `eval`. We can see that this function uses a function called `view` on the expression. This is called a *view* pattern, and it is used to change the structure of pattern matching. This pattern reshapes the data type by either further splitting it into more constructors or, conversely, merging into fewer constructors. Here, we split the addition into three cases based on which side of the expression needs to be evaluated. This makes it easier to prove type preservation, as we can later on in this chapter.

```
1 typeOfExpression' : {numberOfScopes : Nat} → (g : ScopeGraph numberOfScopes NodeTerm) →
2     (Fin numberOfScopes) → Expr → Type → Constraint g
3 typeOfExpression' g s (numLit x) t = EqC num t
4 typeOfExpression' g s (e1 +' e2) t = EqC num t *C
5     (typeOfExpression' g s e1 num *C typeOfExpression' g s e2 num)
```

Figure 5.4: Type system with only number literals and addition.

```
1 eval : Expr → Expr
2 eval e with view e
3 ... | numView {n} = e
4 ... | addLitView {n1} {n2} = (numLit (n1 + n2))
5 ... | addOneView {n1} {e2} = (numLit n1) +' eval e2
6 ... | addView {e1} {e2} = ((eval e1) +' e2)
```

Figure 5.5: Evaluation function.

Proof

After defining the type system, we give the type of our proof for type preservation as in Figure 5.6. Something worth mentioning here is that evaluating the expression could also change the scope graph. In this proof, however, the constraints for both the expression and its evaluated counterpart are checked under the same scope graph in order to make the proof easier. While this breaks the full-support property(4.5) of the scope graph, this is not an issue for the validity of this proof, as there are no variables in this language, and therefore the scope graph is always empty.

```
1 type-preservation-proof : (g : ScopeGraph numberOfScopes NodeTerm) →
2     (s : Fin numberOfScopes) →
3     (e : Expr) (t : Type) →
4     satisfies (sat g (typeOfExpression' g s e t)) →
5     satisfies (sat g (typeOfExpression' g s (eval e) t))
```

Figure 5.6: Type of the type preservation proof.

We proceed with this proof by splitting on the cases for each of the expressions. With the literals and the addition of two literals, the proof is very trivial. Both of these expressions are reduced to number literal; therefore, their type is obviously preserved.

For the cases in which either side of the addition is not yet reduced to literal, the proof becomes more complex. Surprisingly, the main obstacle comes in the form of proving the disjointedness of the graph fragments. Let us now explore the case of adding two non-literal expressions.

As we can see in Figure 5.7, proving that the constraint holds is rather easy. On line 4, we just use the type preservation proof for the left side of the addition(`e1-type-preservation`) and use the hypothesis for the right side of the addition(`e2-num`).

```
1 type-preservation-proof g s (e1 +' e2) num ((+-num , (e1-num , e2-num) , d1) , d2)
2     with view (e1 +' e2)
3 <other cases excluded for conciseness>
4 ... | addView = (refl , (e1-type-preservation , e2-num) ,
5         {! !} ) ,
6         disjointEmpty
7     where
8         e1-type-preservation = type-preservation-proof g s e1 num e1-num
```

Figure 5.7: Incomplete type preservation proof for addition of two expressions.

However, since the user-defined constraint for addition reduces to separating conjunction constraint, we also have to prove that graph fragments of the constraints that type-checks evaluated e_1 and non-evaluated e_2, respectively, are also disjoint. In Figure 5.7, proof for this property is currently left out. We know from our hypothesis that graph fragments for constraints type-checking of non-evaluated e_1 and e_2 are disjoint. How can we use this in our proof?

In our very simple language, we can see that evaluating expressions does not introduce any new variables. This means that the scope graph fragment supporting the constraint that type-checks the evaluated expression will always be smaller or equal to the scope graph fragment supporting the constraint that type-checks the non-evaluated expression. While this is true for this language, it does not have to be true for any language. For example, a language with first-order functions may introduce new variables during evaluation.

Another fact we can derive, this time for all possible graph fragments, is that if two fragments gf_1 and gf_2 are disjoint, then for some fragment gf_1' that is a sub-fragment of gf_1(nodes

in the fragment gf_1' are a subset of nodes in fragment gf_1), gf_1' and gf_2 are also disjoint.

Using the two facts from previous paragraphs, we define lemmas `no-nodes-added` and `smaller-fragment-still-disjoint`. However, since lemma `no-nodes-added` mutually depends on our `type-preservation-proof`, we have to define them in mutually recursive way, as we can see on Figure 5.8.

```
mutual
    type-preservation-proof : (g : ScopeGraph numberOfScopes NodeTerm) →
        (s : Fin numberOfScopes) →
        (e : Expr) (t : Type) →
        satisfies (sat g (typeOfExpression' g s e t)) →
        satisfies (sat g (typeOfExpression' g s (eval e) t))
    type-preservation-proof g s (numLit x) num hypothesis = hypothesis
    type-preservation-proof g s (e1 +' e2) num ((+-num , (e1-num , e2-num) , d1) , d2)
        with view (e1 +' e2)
    ... | addLitView = refl
    ... | addOneView = (refl , (refl , e2-type-preservation) , disjointEmpty) ,
            disjointEmpty
        where
            e2-type-preservation = type-preservation-proof g s e2 num e2-num
    ... | addView = (refl , (e1-type-preservation , e2-num) ,
            smaller-fragment-still-disjoint (no-nodes-added e1) d1) , disjointEmpty
        where
            e1-type-preservation = type-preservation-proof g s e1 num e1-num

    no-nodes-added : {s : Fin numberOfScopes} → (e : Expr) →
        {e-num : satisfies (sat g (typeOfExpression' g s e num))} →
        SubFragment
            (fragment (sat g (typeOfExpression' g s (eval e) num))
                (type-preservation-proof g s e num e-num))
            (fragment (sat g (typeOfExpression' g s e num))
                e-num)
    no-nodes-added (numLit x) = subFragmentEmp
    no-nodes-added (e1 +' e2) with view (e1 +' e2)
    ... | addLitView = subFragmentEmp
    ... | addOneView = no-nodes-added e2
    ... | addView = subfragment-merge-preservation (no-nodes-added e1)
```

Figure 5.8: Complete type preservation proof.

5.3 Discussion

In the previous two sections, we demonstrated how we can use STATIX-IN-AGDA to define type systems and how to prove type preservation. To answer the last research question, we can see that with our design of the embedding, typing rules translate well to STATIX-IN-AGDA specification, as evident when comparing code in Subsection 5.1.1 to typing rules on Figure 5.2. The current downside to our approach is that the type-checking proofs have to be written manually in Agda. As we will see later in the next chapter, the way to improve this could be by introducing higher-level combinators that would make writing proofs easier or by proof automation.

We have also shown how we can prove type preservation for a simple type system with numbers and addition, specified using STATIX-IN-AGDA. While our example demonstrates the foundation of how we can prove this meta-theoretical property, it does not demonstrate the

full power of scope graphs and Statix. For this reason, for future work, we would like to extend the language with variables and functions. More on this in the next chapter.

Chapter 6

Future work

In this chapter, we outline possible improvements and future research directions that could build upon the work of this thesis. We explore how the user experience can be improved, discuss the translation from Statix specifications to their STATIX-IN-AGDA counterparts, and suggest possible new features. We also discuss how the proof from the previous chapter can be extended in the future to be applicable to more useful and complex languages.

6.1 Type preservation for a language extended with functions and variables

The language from the proof in Section 5.2 is very simple and, unfortunately, useless for practical applications. Additionally, proving type preservation for such a simple language without variables does not demonstrate the full capabilities of Statix. Therefore, arguably, one of the most important future directions is extending the language with variables and functions and proving type preservation for this new extended language. This would give us further evidence about the usefulness of STATIX-IN-AGDA in practical scenarios.

The first issue with this, which has already been mentioned in the previous chapter, is that user-defined constraints on evaluated expressions must be satisfied with respect to different scope graphs. While this has not been an issue in a language without variables, it would be here. Evaluating a variable expression to a literal value expression eliminates the node from the scope graph fragment, supporting the corresponding constraint. This means that using the same scope graph would break the top-level graph fragment property 4.5.

The even bigger issue is that introducing variables to the language requires us to have some value-binding model, such as environment or substitution. This binding model needs to have some correspondence to the scope graph, such that variables in the model also need to be in the scope graph. One hint on how this can be solved can be found in previous work by Bach Poulsen, Néron, et al. 2016. The authors introduce the *scopes-as-frames* approach, in which the run-time heap is organised into a graph corresponding to the scope graph.

6.2 User experience

During the experimentation stage of this thesis, we realised that some usability aspects of STATIX-IN-AGDA can be improved. The first aspect is defining the sat function for the constraints. Currently, this function returns a dependent pair of satisfiability of the constraint and its supporting graph fragment. However, this definition worsens the readability of satisfiability proofs of constraint and makes the type preservation proof more difficult. One solution could be to split the definition of the sat. This could, however, compromise the

property of disjointedness of fragments, as the satisfiability of separating conjunction constraint depends on it. Therefore, this requires further investigation.

Another usability aspect that could be improved is the way that the type-check proofs are written. Currently, even proof for a small program results in a relatively large, even though very simple proof. Due to the simplicity of some of these proofs, future work could focus on their automation in the form of tactics. This could make writing proofs less tedious, faster, more readable, and maybe even more fun. We could also analyse the proofs to find commonly used patterns to create high-level helper lemmas that users could reuse in their proofs as an alternative to proof automation. This would make proofs more straightforward to write and much more compact and readable.

6.3 Translation

Our vision for statix-in-agda is that users can create and use their type system specification using Statix, with its Agda counterpart being automatically generated based on this. Translation of Statix specifications is out of the scope of this project. Still, it is currently one of the most critical future work directions, as it is the first step to being able to reason formally about the existing Statix specifications.

One of the main challenges that were discussed during this project, related to translation, was translating the result of the constraint solver in Statix to Agda. As we have seen earlier in this thesis, we need this result because the scope graph of a program needs to be given as input to the type-check proof. In the current Java implementation of Statix[1], this result can be obtained from `ASolverResult.java`. For future work, we need to translate this result to the Agda representation of the scope graph.

6.4 New constraints

Finally, the statix-in-agda can be extended with additional constraints beyond the simplified Statix-core. These constraints include disequality(holds when two terms are not equal) or arithmetic constraint(compares terms or performs arithmetic operations such as addition). There is also the try constraint that checks whether an inner constraint that is supplied to it as an argument holds with regard to the outer context. If the inner constraint does not hold, a message will be given. Due to the side-effect nature of messages, embedding this constraint could prove challenging and requires further investigation. However, implementing these new constraints is less important than the improvements mentioned in previous sections.

[1] `https://github.com/metaborg/nabl/tree/master`

Chapter 7

Related Work

In this chapter, we discuss related work and put it in the context of our approach. Our discussion primarily centers on the formalization of scope graphs, the verification of meta-theoretical properties of type systems, recent research advancements in Statix, and other similar tools for formal specification of programming languages.

7.1 Scope Graph Formalization

Scope graphs have already been formalised previously in Bach Poulsen, Rouvoet, et al. 2017. The authors of this paper created *scope-and-frames* Agda library that combines the scope graphs with frames [Bach Poulsen, Néron, et al. 2016] to define an intrinsically typed definitional interpreter for STLC. The work of Casamento 2019 further builds upon the work of Bach Poulsen, Rouvoet, et al. 2017. The author presents a pattern for constructing an intrinsically-verified type-checker, with example Agda implementations for STLC and a toy procedural language. They also give generalised implementation of scope graphs to allow for scopes to be represented by any `Listable` type [Firsov and Uustalu 2015], instead of Agda's `Fin` type. Using the `Fin` requires the knowledge of how many scopes are in the scope graph, which is inconvenient during generalization. This paper's implementation of scope graphs is based on the works of Bach Poulsen, Rouvoet, et al. 2017 and Casamento 2019.

7.2 Meta-Theory

As we have seen in Chapters 5 and 6, one of the most important directions for future research is expanding the type preservation proof for languages with variables. The main challenge lies in connecting the program's runtime layout of memory and its corresponding scope graph. The solution to this challenge may be in Bach Poulsen, Néron, et al. 2016, where authors introduce *scope-as-types* paradigm. Using this approach, the memory layout of a program is structured into a graph-like structure consisting of heaps, frames, and slots. In this structure, the slots in the frame correspond to declarations in the scope, and the links between frames correspond to edges in the scope graph. Using this correspondence between frames and scopes, as well as the property that the value in each slot is of the correct type, we can prove type preservation. As mentioned in the Chapter 6, utilizing this approach in STATIX-IN-AGDA could be a way to prove type preservation for more complex languages.

7.3 Statix

Given the fundamental role of Statix in this thesis, it is essential to explore other recent research directions related to it. In a recent review paper by Zwaan and Antwerpen 2023,

the authors present an overview of the past research about scope graphs and Statix. Here, some of the more recent papers focus on improving the performance of type-checking with Statix. The first attempt by Van Antwerpen and Visser 2021 focuses on concurrency in type-checking. In this approach, the scope graph parts are assigned to hierarchically organised compilation units. These units are then assigned to actors that execute the type-checking in parallel. Later, in Zwaan, Antwerpen, and Visser 2022, the authors extended the framework with incrementality. In incremental type-checking, only the parts of the code that have been updated are type-checked.

7.4 Tools for formal specification of programming languages

Now, let us zoom out and explore other tools for the specification of programming languages. One of the tools similar to Spoofax[1](the language workbench that Statix is a part of) is *K Framework*[2]. In this framework, a language specification consists of sentences that describe the language's syntax(language primitives), configuration(the state of the system), and finally, context and rules(behavior of the system). After the specification is compiled, it can be executed and used for different language tools. One of these tools is `kprove`, which can be used for proving claims about the language specification.

Another related tool is *Ott*[3] by Sewell et al. 2010. This tool takes a specification of the language's syntax and semantics, as well as the binding specification. Based on the specification, Ott can generate either informal LaTeX representations or, by adding annotations with additional information, representations in proof assistants such as Coq or HOL that can be used to reason about the specification formally.

[1] `https://spoofax.dev`

[2] `https://kframework.org`

[3] `https://github.com/ott-lang/ott`

Chapter 8

Conclusions

This thesis introduced a library called STATIX-IN-AGDA, the embedding of Statix within Agda, that allows users to formally reason about the type systems. This library formalises a set of predefined basic and scope graph constraints from the simplified subset of Statix called Statix-core. Using these predefined constraints, users can construct the typing rules for their language, also called user-defined constraints.

The first part of this library is scope graphs, which are used for name resolution during type-checking. This thesis changes and extends the existing scope graph formalization in two ways. First, it changes the way scope graphs are modeled to account for the changes in scope graphs introduced by Rouvoet et al. 2020. In this new representation, the scopes cannot have multiple declarations. Instead, each declaration has its own scope. Second, it introduces the notion of scope graph fragments used for constraint support, ensuring sound type-checking.

The second part of this library is constraints. One of the challenges we tackled in this thesis was the choice of depth of the embedding for the Statix. For the basic and scope graph constraints, we chose deep embedding. In this style of embedding, the syntax and the semantics of the constraints are defined separately, giving us a way to restrict users from freely creating new, possibly unsound, basic, and scope graph constraints. This way, user-defined constraints can only be defined in terms of existing constraints.

We also showed two examples of the use of STATIX-IN-AGDA. In the first example, we specified a simple language with variables and functions and showed how we can prove that programs in this language are well-typed. In the second, we give proof of type preservation for a very simple language with numbers and addition.

For future research, we suggested a few improvements and research directions. One of the most important ones is a translator from Statix to Agda, giving users a way to be able to verify their existing type system specifications. Another research direction could be exploring the possibilities of reasoning about language meta-theory. Finally, STATIX-IN-AGDA could benefit from improved usability and additional constraints.

Bibliography

Antwerpen, Hendrik van et al. (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

Bach Poulsen, Casper, Pierre Néron, et al. (2016). "Scopes describe frames: A uniform model for memory layout in dynamic semantics". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Bach Poulsen, Casper, Arjen Rouvoet, et al. (2017). "Intrinsically-typed definitional interpreters for imperative languages". In: *Proceedings of the ACM on Programming Languages* 2.POPL, pp. 1–34.

Casamento, Katherine Imhoff (2019). "Correct-by-Construction Typechecking with Scope Graphs". PhD thesis. Portland State University.

Firsov, Denis and Tarmo Uustalu (2015). "Dependently typed programming with finite sets". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pp. 33–44.

Gibbons, Jeremy and Nicolas Wu (2014). "Folding domain-specific languages: deep and shallow embeddings (functional Pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, pp. 339–347. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628138. URL: http://doi.acm.org/10.1145/2628136.2628138.

Néron, Pierre et al. (2015). "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: 10.1007/978-3-662-46669-8_9. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

Rouvoet, Arjen et al. (2020). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA. DOI: 10.1145/3428248. URL: https://doi.org/10.1145/3428248.

Sewell, Peter et al. (2010). "Ott: Effective tool support for the working semanticist". In: *Journal of functional programming* 20.1, pp. 71–122.

Van Antwerpen, Hendrik and Eelco Visser (2021). "Scope states: Guarding safety of name resolution in parallel type checkers". In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Zwaan, Aron and Hendrik van Antwerpen (2023). "Scope Graphs: The Story so Far". In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASIcs.EVCS.2023.32. URL: https://doi.org/10.4230/OASIcs.EVCS.2023.32.

Zwaan, Aron, Hendrik van Antwerpen, and Eelco Visser (2022). "Incremental type-checking for free: Using scope graphs to derive incremental type-checkers". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pp. 424–448.