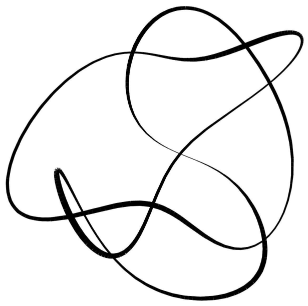
A Constraint-based Approach to Name Binding and Type Checking using Scope Graphs

Master's Thesis



Hendrik van Antwerpen

A Constraint-based Approach to Name Binding and Type Checking using Scope Graphs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Hendrik van Antwerpen
born in Rotterdam, Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

This work is part of the following publication:
Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, January 2016.

Cover picture: Prime knot 8_9. Courtesy of David Fremlin.

A Constraint-based Approach to Name Binding and Type Checking using Scope Graphs

Author:	Hendrik van Antwerpen
Student id:	1148974
Email:	H.vanAntwerpen@student.tudelft.nl

Abstract

Recently scope graphs were introduced as a formalism to specify the name binding structure of a program and do name resolution independent of the abstract syntax tree of a program. In this thesis we show how to use a constraint language based on scope graphs to do static analysis of programs. We do this by extracting constraints from a program, that specify name binding and typing. We treat binding and typing as separate building blocks, but our approach allows language constructs – such as access of record fields – where name and type resolution are mutually dependent. By using scope graphs for name resolution, our approach supports a wide range of name binding patterns that are not easily supported in existing constraint-based approaches. We present a formal semantics for our constraint language, as well as a solver algorithm, for which we discuss soundness, termination and completeness of the solver. We evaluate our approach by expressing the static semantics of PCF and Featherweight Java with our constraints, and we implemented the solver algorithm, as well as static analysis for both languages, in the Spoofax language workbench.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. G.H. Wachsmuth, Faculty EEMCS, TU Delft
Committee Member:	Dr. W.S. Swierstra, Faculty of Science, Utrecht University

Preface

Explain, before you travel on
What faculty you've settled on.

Mephistopheles to a student
In: *Goethe*, Faust

When I came to Delft, I was convinced a degree in physics is what I wanted. It would take several more years, and a few detours, before I finally found my calling as a computer scientist. With the submission of this thesis, I now enter the final stage of my studies. This thesis is the result of a year's work on a project, that was very exciting for me. Not only was I fascinated by the topic, but I was also motivated by the fact that we aimed for a publication. I want to thank Andrew Tolmach, Eelco Visser, Guido Wachsmuth, and Pierre Néron, for showing me the ropes of academic research. At times it felt as if I had four supervisors, instead of a single one, which may sound like a challenge, but was actually very positive. Our discussions have taught me a lot, and I have enjoyed working together.

During my years of study, I have enjoyed the company and support of many people. I want to thank my parents, whose continuous support was essential in getting where I am now. I am grateful for the many wonderful and interesting people I have met, and the good friends I have made during my studies, whether it was in a board, on the dance floor, at work, or elsewhere. Finally, I want to thank Nina for proofreading this thesis, but more so for regularly bringing me back to all that is called reality.

Hendrik van Antwerpen
Delft, Netherlands
December 28, 2015

Contents

List of Figures

List of Theorems

Chapter 1

Introduction

Language workbenches (Fowler, 2005) support the implementation of compilers and integrated development environments for programming languages. They provide high-level specification languages – or meta-languages – for different language aspects, such as static and dynamic semantics, and editor services. These specifications are interpreted by generic algorithms, or used to generate an implementation. This hides low-level implementation details from a language designer, both taking away a big source of errors, and speeding up development. There are several actively developed language workbenches available (Erdweg et al., 2013). Finding useful abstractions for specifying language aspects is an active area of research (Visser et al., 2014).

The specification of static semantics is one of these aspects. When talking about statically typed languages, we distinguish between static and dynamic semantics. A static semantics describes when a program is well-formed, by means of a type system. A dynamic semantics describes how such a well-formed program is executed (Harper, 2012). Type checking serves to "prove the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" (Pierce, 2002). Type checkers are algorithms that infer the types of a program, or check it against user-supplied type annotations. One approach to implement type checkers is to representing well-formedness of the program as a constraint problem, and implement the type checker as a constraint solver. A satisfying solution to the constraint problem corresponds to a valid type for the program. This allows a clean separation between specifying when a program is well-formed – by generating constraints – and checking or inferring the program types – by solving those constraints. The constraint generation is concerned with the details of the programming language, while the constraint solver can ignore those details, and only depends on the, usually much smaller, constraint language.

Constraint based type checkers have been developed for several (classes of) languages, e.g. work by Heeren et al. (2002); Pottier and Rémy (2005); Sulzmann et al. (1999) for functional languages with Hindley-Milner style type systems, like ML and Haskell, and work by Palsberg (1996) for object-oriented languages with subtyping and inheritance. These constraint systems tend to depend on the specific name binding structure and types of the object language. In more theoretical treatments, the issue of name binding is sometimes completely ignored. Reuse of these systems outside of their intended domain is therefore cumbersome.

In this thesis we present a constraint language that is not specific to one language, but can be used to express the static semantics for a range of languages. Type constraints are based on syntactic equality and unification, a well-understood theory, which is at the basis of many constraint based type systems. Name binding constraints are based on the name resolution calculus, recently introduced by Neron et al. (2015). By representing the name binding structure of a program as a scope graph, it allows us to do name resolution independent of the programs abstract syntax tree. We also introduce constraints to check that names are distinct, or that two sets of names are disjoint. By using an extended theory that allows incomplete scope graphs that contain constraint variables, we support interaction between name resolution and type checking.

This is important, because although name resolution and type checking cannot always be stratified. In languages with records or classes, named access to a field by name depends on type information. For example in Java, the expression `e.fld` refers to a field, which can only be resolved when the class type of `e` is known. The type of the field can only be determined after the name is resolved, therefore a mutual dependency between name binding and type checking sometimes exists. Our approach allows us to support language features like this.

We evaluated the expressiveness of our approach by using the constraint language to express the semantics of two languages, and the practicality by implementing a prototype of the solver.

1.1 Contributions

The work presented in this thesis is also part of an upcoming publication (van Antwerpen et al., 2016). We will first list the contributions of this thesis, and then highlight the main differences with the publication. The contributions of this thesis are the following:

- We present a constraint language that combines type checking and name resolution constraints, and give a formal semantics for the constraint language.
- We present a solver for constraint problems, and show that it is sound and terminating.
- We express the semantics of PCF and Featherweight Java using our constraint language.
- We implemented the solver, and static analysis for PCF and Featherweight Java, in the Spoofax language workbench.

Compared to the paper there are a few differences:

- The constraint language presented in the paper lacks support for subtyping.
- The presentation in the paper treats constraints for scope graph construction differently from the other constraints. Scope graph constraints are processed first, before solving other constraints. In our presentation constraints for scope graph construction are handled uniformly with other constraints, both in the semantics and in the solver.
- The paper contributes an extension of the name resolution theory of Neron et al. (2015), which is not a contribution of this thesis. Since the constraint language relies on this extension, we present it in the preliminaries, separate from our own contributions.
- We provide a soundness proof for the solver that is (1) adapted to the uniform treatment of scope graph constraints, (2) supports the new subtyping constraints, and (3) is more detailed than the proof from the paper.
- The evaluation of the constraint system we present is not part of the paper.

1.2 Outline

The rest of this thesis is organized as follows. In Chapter 2 we introduce the theory of scope graphs and typing constraints on which the rest of the work depends. In Chapter 3 we present the constraint language, and state the formal semantics of the constraints. In Chapter 4 we present an algorithm to solve problems in our constraint language, and discuss termination, soundness and completeness of the algorithm. In Chapter 5 we show how the constraint language can be used to express static analysis for PCF and Featherweight Java, and discuss the prototype implementation. In Chapter 6 we look at related work, and in Chapter 7 we discuss limitations and possible future research directions.

Chapter 2

Preliminaries

In this chapter we introduce the two building blocks of our approach, scope graphs and type constraints. We use a model language with modules and records (LMR), for our examples. We give an overview of LMR in the first section. In the second section, we introduce name resolution using scope graphs using a series of examples, that focus on different name binding patterns. We finish the section by discussing a resolution algorithm. In the last section, we introduce constraint based type checking for the simply type λ-calculus. First we introduce constraint syntax and generation, following an example program. Finally we describe unification as a technique to solve these constraints.

2.1 The Language LMR

LMR is a small model language we use throughout this thesis for examples. The syntax is shown in Figure 2.1. The language is basically a simply-typed functional language, with modules and records. We will introduce each of these below.

- LMR is a *simply-typed functional language*, with boolean and integer base types. A conditional `if` expression is provided, as well as common arithmetic and relational operators.

- Record, module, and variable definitions are introduced using top-level statements, written as `module`, `record`, and `def` respectively. Modules consist of definitions themselves, and can be nested. Modules can import other modules by name using an `import` statement. New bindings in expressions are introduced by functions, written as `fun`, and recursive let expressions, written as `letrec`.

- The language supports user-defined named *records*. A record can have a super type, resulting in nominal subtyping in the language. Field names are not allowed to shadow fields in the supertype. Record instances are created with `new` expressions. Record fields are accessed using dot nota-

$$
\begin{array}{rcl}
\mathit{prog} & := & \mathit{decl}^* \\
\mathit{decl} & := & \texttt{module}\ \mathit{id}\ \texttt{\{}\ \mathit{decl}^*\ \texttt{\}} \\
 & | & \texttt{import}\ \mathit{id} \\
 & | & \texttt{def}\ \mathit{bind} \\
 & | & \texttt{record}\ \mathit{id}\ \mathit{super}\ \texttt{\{}\ \mathit{fdecl}^*\ \texttt{\}} \\
\mathit{super} & := & \texttt{extends}\ \mathit{id} \mid \epsilon \\
\mathit{fdecl} & := & \mathit{id}\ \texttt{:}\ \mathit{tyann} \\
\mathit{tyann} & := & \texttt{Int} \mid \texttt{Bool} \mid \mathit{id} \mid \mathit{tyann}\ \texttt{->}\ \mathit{tyann} \\
\mathit{expr} & := & \mathit{int} \\
 & | & \texttt{true} \mid \texttt{false} \\
 & | & \mathit{id} \\
 & | & \mathit{expr} \oplus \mathit{expr} \\
 & | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & | & \texttt{fun}\ \texttt{(}\ \mathit{id}\ \texttt{:}\ \mathit{tyann}\ \texttt{)}\ \texttt{\{}\ \mathit{expr}\ \texttt{\}} \\
 & | & \mathit{expr}\ \mathit{expr} \\
 & | & \texttt{letrec}\ \mathit{bind}^*\ \texttt{in}\ \mathit{expr} \\
 & | & \texttt{new}\ \mathit{id}\ \texttt{\{}\ \mathit{fbind}^*\ \texttt{\}} \\
 & | & \texttt{with}\ \mathit{expr}\ \texttt{do}\ \mathit{expr} \\
 & | & \mathit{expr}\texttt{.}\mathit{id} \\
\mathit{bind} & := & \mathit{id}\ \texttt{=}\ \mathit{expr} \\
 & | & \mathit{id}\ \texttt{:}\ \mathit{tyann}\ \texttt{=}\ \mathit{expr} \\
\mathit{fbind} & := & \mathit{id}\ \texttt{=}\ \mathit{expr} \\
 & & \\
\mathit{Type} & := & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Rec}(\mathit{Decl}) \mid \texttt{Fun}[\mathit{Type},\mathit{Type}]
\end{array}
$$

Figure 2.1: Syntax of LMR

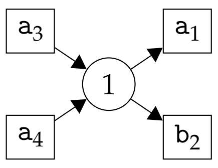
tion. The `with` expression takes a record instance as its first argument, and allows unqualified access to the record fields body expression.

2.2 Name Binding with Scope Graphs

Scope graphs are recently introduced as a formalism to represent the name binding structure of a program, and do name resolution, independent of a programs abstract syntax tree (AST) (Neron et al., 2015). The formalism consists of a calculus, and a graphical notation for scope graphs. It is based on the idea of representing name binding structure as a scope graph, consisting of scopes, references and declarations. Name resolution corresponds to finding the shortest path in the graph from references to declarations of the same

```
def a1 = 1
def b2 =
  if (a3 == 0)
    then 4
    else a4
```

(a) Program



(b) Scope graph

$$\mathtt{a}_3^{\mathsf{R}} \mapsto \mathtt{a}_1^{\mathsf{D}} \quad \mathbf{D}(\mathtt{a}_1^{\mathsf{D}})$$
$$\mathtt{a}_4^{\mathsf{R}} \mapsto \mathtt{a}_1^{\mathsf{D}} \quad \mathbf{D}(\mathtt{a}_1^{\mathsf{D}})$$

(c) Resolution

Figure 2.2: Example of identifiers in global scope

name.

The original theory was generalized and extended, and integrated with a constraint language to use it for static analysis of programs (van Antwerpen et al., 2016). In this section we will present the extended formalism, while the constraint system, which is the contribution of this thesis, will be the subject of the following chapters.

We gradually introduce the different features of scope graphs, using binding patterns of LMR as examples. When introducing a new concept, we show the graphical notation, before the formal notation of the calculus. For reference, we present a complete overview of the notation for the name resolution calculus in Figure 2.7.

2.2.1 Declarations and References

We start by considering the example in Figure 2.2, which consists of only two top-level definitions. Names in the program are subscripted with a position, to distinguish between different occurrences of the same name. The subscripts are therefore not part of the language. The graphical representation of the scope graph for this program is shown right of the program. It contains a single scope, two references to a, and two declarations, a and b. In general, a scope graph – written as $\mathcal{G}$ – consists of three types of nodes:

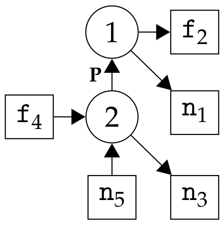
- A *scope* represents a set of AST nodes that behave uniformly with respect to name binding. Scopes are identified by elements from an abstract enumerable set. We will use numbers in the presentation to identify scopes. In the graphical notation, a scope looks like $\textcircled{1}$. In the calculus we write a scope as 1^{S}.

- An *occurrence of a name*, is a name x with a unique position i attached to it. We write the position as a subscript to the name, as in x_i. In the graphical notation, names look like $\boxed{x_i}$.

- A *declaration* introduces a new name in scope. In the graphical notation, we write $\textcircled{i} \rightarrow \boxed{x_j}$ for a declaration, an arrow from the scope to the occurrence of the name. In the calculus, a declaration is written as x_i^{D}. We may omit the position if it is not relevant. The scope of a declaration

```
def n_1 = true
def f_2 =
  fun (n_3: Int) {
    f_4(n_5)
  }
```

(a) Program



(b) Scope Graph

$\mathtt{f}_4^{\mathsf{R}} \mapsto \mathtt{f}_2^{\mathsf{D}}$ $\quad \mathbf{E}(\mathbf{P}, 1^{\mathsf{S}}) \cdot \mathbf{D}(\mathtt{f}_2^{\mathsf{D}})$

$\mathtt{n}_5^{\mathsf{R}} \mapsto \mathtt{n}_3^{\mathsf{D}}$ $\quad \mathbf{D}(\mathtt{n}_3^{\mathsf{D}})$

(c) Resolution

Figure 2.3: Example of lexical scoping

is written as $\mathcal{S}(x_i^{\mathsf{D}})$. The set of all declarations in a scope is defined as $\mathcal{D}(s) := \{d \mid \mathcal{S}(d) = s\}$.

- A *reference* refers to a declared name in scope. In the graphical notation, a reference is depicted as $\boxed{x_i} \rightarrow \textcircled{j}$, an arrow going from an occurrence of a name to a scope. In the calculus, we write a reference as x_i^{R}. The scope of a reference is written as $\mathcal{S}(x_i^{\mathsf{R}})$. The set of all references is defined as $\mathcal{R}(s) := \{r \mid \mathcal{S}(r) = s\}$.

Note that operations such as $\mathcal{D}(s)$ are implicitly parametrized by a scope graph $\mathcal{G}$. When the context requires this, we make the scope graph explicit with a subscript, e.g. $\mathcal{D}_{\mathcal{G}}(s)$.

Intuitively, a reference resolves to a declaration of the same name, if a path from the reference to the declaration exists in the scope graph. In our examples both references resolve to the declaration x_1^{D}. We write a resolution as $x_3^{\mathsf{R}} \mapsto x_1^{\mathsf{D}}$. A resolution path describes the steps in the graph necessary to get from the reference to the declaration. In our example the reference and declaration are in the same scope, so we can step directly to the declaration. This is written as $\mathbf{D}(d)$. The calculus supports ambiguous resolution, so if multiple declarations of the same name exist, a reference can resolve to a set of declarations.

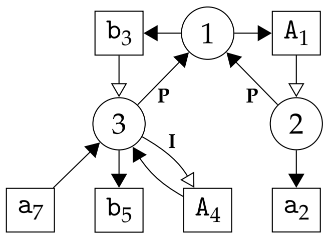
2.2.2 Lexical Scoping

The example the program in Figure 2.3, features nested lexical scopes. The global scope contains two declarations, `n` and `f`. The function introduces a new scope, which contains the declaration of the parameter `n`. The scope of the function is connected to its parent scope by an edge, labeled **P** for parent. Later we will also see other labels on edges. In general, scope graphs allow labeled edges between scopes:

- A *direct edge* between two scopes i^{S} and j^{S}, allows declarations from j^{S} scope to become visible in i^{S}. In the graphical notation we depict such an edge as $\textcircled{i} \xrightarrow{l} \textcircled{j}$, where l is its label. In the calculus we write $\mathcal{E}_l(s)$ for the set of l-labeled edges starting from s. The set of scopes reachable from s through an l-labeled edge is defined as $s_l^{\blacktriangleright} \equiv \{s' \mid (s, s') \in \mathcal{E}_l(s)\}$.

```
module A1 {
  def a2 = 4
}
module B3 {
  import A4
  def b5 = a6
}
```

(a) Program



(b) Scope graph

$\mathtt{A}^{\mathsf{R}}_4 \mapsto \mathtt{A}^{\mathsf{D}}_1$

$\mathbf{E}(\mathbf{P},1^{\mathsf{S}}) \cdot \mathbf{D}(\mathtt{A}^{\mathsf{D}}_1)$

$\mathtt{a}^{\mathsf{R}}_7 \mapsto \mathtt{a}^{\mathsf{D}}_2$

$\mathbf{N}(\mathbf{I},\mathtt{A}^{\mathsf{R}}_4,2^{\mathsf{S}}) \cdot \mathbf{D}(\mathtt{A}^{\mathsf{D}}_1)$

(c) Resolution

Figure 2.4: Example of module import

The reference n^{R}_5 in our example resolves to n^{D}_3, even though n^{D}_1 is also reachable in the graph. This is because declarations in outer scopes are shadowed by declarations with the same name in the inner scope. Consider the resolution path to the outer declaration, which would be $\mathbf{E}(\mathbf{P},1^{\mathsf{S}}) \cdot \mathbf{D}(n^{\mathsf{D}}_1)$. Since it is longer than the alternative (chosen) resolution, it is rejected. In general, the order on resolution paths is determined by a visibility order:

- The theory is parametrized by a set of *labels* $\mathcal{L}$ and a partial *label order* $< \, : \mathcal{L} \times \mathcal{L}$. The predefined label $\mathbf{D}$ is used for the step from a scope to a declaration in that scope. This label is the smallest possible label, i.e. $\forall l \in \mathcal{L}.\,(l \neq \mathbf{D} \implies \mathbf{D} < l)$.

- The *visibility order* on paths is defined inductively based on the label order. The order is the order of the first labels, unless their order is not defined, in which case the order is that of the tails of the paths.

We can now see that the desired lexical scoping behavior is achieved using $\mathcal{L} = \{\mathbf{P}\}$, and a label order $\mathbf{D} < \mathbf{P}$.

2.2.3 Imports

Our next example, shown in Figure 2.4, defines two modules, `A` and `B`. Module `B` imports module `A`, making the declaration of variable `a` accessible in the body of `B`. This behavior is modeled by two edges in the scope graph. One edge associates declaration A^{D}_1 with scope 2^{S}. Another edge – labeled $\mathbf{I}$ for import – from 3^{S} to reference A^{R}_4, imports the scope associated with the declaration that reference A^{R}_4 resolves to. One can think of these two edges as forming a higher-order edge between the scopes 3^{S} and 2^{S}. In general, scope graphs allow labeled edges from scopes to references, and edges from declarations to scopes:

- An *associated scope edge* relates a declaration x^{D}_i with a scope j^{S}. In the graphical notation, an associated scope is depicted as $\boxed{x_i} \longrightarrow \textcircled{j}$. In the calculus we write the associated scope of a declaration as $\mathcal{A}(x^{\mathsf{D}}_i)$.

- A *named edge* between a scope i^{S} and a reference x_j^{R}, can be thought of as a higher-order edge. It allows a step to the associated scopes of the declarations that x_j^{R} resolves to. The declarations from those scopes will be visible in i^{S}. In graphical notation, a named edge is depicted as $\textcircled{i} \xrightarrow{l} \boxed{x_j}$. In the calculus we write $\mathcal{N}_l(s)$ for the set of l-labeled edges starting from s. The set of scopes reachable through an l-labeled named edge is defined as $s_l^{\triangleright} \equiv \{s' \mid (s,r) \in \mathcal{N}_l(s) \wedge r \mapsto d \wedge \mathcal{A}(d) = s'\}$.

In our example we introduced a new label **I** to indicate an import edge. LMR follows the usual convention that imported declarations have precedence over declarations from outer lexical scopes. We model this by adding $\mathbf{I} < \mathbf{P}$ to the label order. However, this is not enough to ensure resolution is correct. It is still possible to resolve a reference in the lexical parent of an imported module, yielding a resolution path like $\mathbf{N}(\mathbf{I},\cdot,\cdot) \cdot \mathbf{E}(\mathbf{P},\cdot) \cdot \mathbf{D}(\cdot)$. The visibility ordering cannot prevent a resolution like this. Therefore, the theory uses a well-formedness predicate on resolution paths to restrict the set of valid resolutions:

- Path *well-formedness* is defined as $\mathrm{WF}(p) \equiv \mathit{labels}(p) \in L(\mathcal{E})$, where the function $\mathit{labels}(p)$ projects the labels of a resolution path, and $\mathcal{E}$ is a regular expression over a string of labels. The calculus is parametrized in the regular expression $\mathcal{E}$.

Now, by choosing the well-formedness regular expression to be $\mathcal{E} = \mathbf{P}^* \cdot \mathbf{I}^*$, we prevent visiting lexical parent scopes after we have used an import edge. Note that this allows transitive imports. If we would like to have non-transitive imports instead, we can use $\mathcal{E} = \mathbf{P}^* \cdot \mathbf{I}$.

2.2.4 Type-dependent Name Resolution

In the program in Figure 2.5, we define an instance `b` of a record, and then access its field `f`. The declarations and references are not enough to resolve the field reference. The the scope in which we need to resolve the field name depends on the type of `b`. Instead of the unknown scope, we use a variable ς as the target of the direct edge from 4^{S}. We use the label **S** to distinguish edges that have to do with record subtyping from module imports. In the following chapter we will explain that this variable is a constraint variable, which is instantiated by a constraint solver. For now we just show the scope graph before and after instantiating the variable to 2^{S}. In general, the calculus allows scope graphs to be incomplete:

- An *incomplete scope graph* is a graph $\mathcal{G}$ where there exist direct edges whose target is a variable. In the graphical notation, we write $\textcircled{i} \xrightarrow{l} \varsigma$.

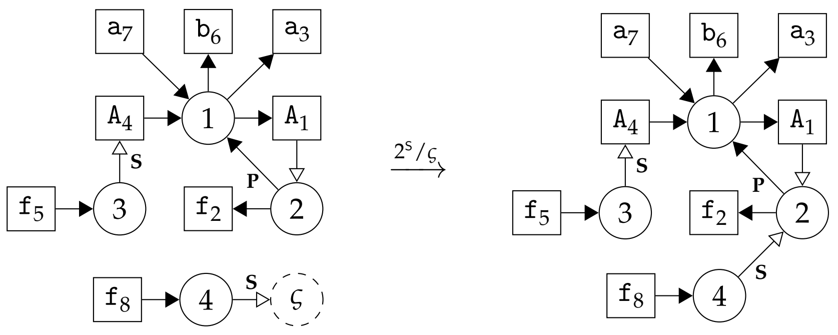
Since we have introduced a new label **S** for edges related to record subtyping, we must also update the label order and well-formedness predicate. We add $\mathbf{S} < \mathbf{P}$ to the label order. Note that we do not define the order between **S** and **I**. Since we do not create scope graphs for LMR programs where scopes

$\texttt{record}\ \texttt{A}_1\ \{\ \texttt{f}_2\ :\ \texttt{Int}\ \}$
$\texttt{def}\ \texttt{a}_3\ =\ \texttt{new}\ \texttt{A}_4\ \{\ \texttt{f}_5\ =\ 1\ \}$
$\texttt{def}\ \texttt{b}_6\ =\ \texttt{a}_7.\texttt{f}_8$

(a) Program

$$\begin{array}{ll}
\texttt{A}_4^{\mathsf{R}} \mapsto \texttt{A}_1^{\mathsf{D}} & \mathbf{D}(\texttt{A}_1^{\mathsf{D}}) \\
\texttt{f}_5^{\mathsf{R}} \mapsto \texttt{f}_2^{\mathsf{D}} & \mathbf{N}(\mathbf{S}, \texttt{A}_4^{\mathsf{R}}, 2^{\mathsf{S}}) \cdot \mathbf{D}(\texttt{f}_2^{\mathsf{D}}) \\
\texttt{a}_7^{\mathsf{R}} \mapsto \texttt{a}_3^{\mathsf{D}} & \mathbf{D}(\texttt{a}_3^{\mathsf{D}}) \\
\texttt{f}_8^{\mathsf{R}} \mapsto \texttt{f}_2^{\mathsf{D}} & \mathbf{E}(\mathbf{S}, 2^{\mathsf{S}}) \cdot \mathbf{D}(\texttt{f}_2^{\mathsf{D}})
\end{array}$$

(b) Resolution



(c) Initial scope graph (d) Instantiated scope graph

Figure 2.5: Example of record field access

have both types of edges, this is not relevant. In general though, the references visible through two edges with labels l_1 and l_2 will be treated equally if l_1 and l_2 have no defined order. Our well-formedness regular expression will be $\mathcal{E} \equiv \mathbf{P}^* \cdot (\mathbf{S}^* + \mathbf{I}^*)$. By using a different label for supertype edges, we can still choose between transitive and non-transitive module imports, without breaking record subtyping.

Resolution in an incomplete graph is still partially possible. We observe that a variable edge represents a unknown set of visible declarations. This means that any declaration visible through an edge with a larger label might be shadowed, and is therefore unsafe. Any declaration visible through edges with smaller labels can however safely be resolved. We will see in the next section that the resolution algorithm resolves as many references as possible. In Chapter 3 we will see how alternating partial resolution and variable instantiation can be used for mutually dependent name resolution and type checking.

2.2.5 Resolution Algorithm

In the previous section we have already informally talked about resolving names. In the name resolution calculus this process is formally defined with reachability, visibility, and resolution relations:

- A declaration is *reachable* from a scope s, written as $\vdash_{\mathcal{G}} s \rightarrowtail x_i^{\mathsf{D}}$, if there is a path in $\mathcal{G}$ from s to the declaration.

$$\boxed{\textsc{Res}, \textsc{Env}}$$

$$\textsc{Res}[\mathbb{I}](x_i^{\mathsf{R}}) \quad := \quad \begin{cases} \bot \text{ if } p = \mathsf{P} \text{ and } \{x_j^{\mathsf{D}} \mid x_j^{\mathsf{D}} \in D\} = \emptyset \\ \{x_j^{\mathsf{D}} \mid x_j^{\mathsf{D}} \in D\} \end{cases}$$
$$\text{where } \textsc{Env}_{\mathcal{E}}[\{x_i^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](\mathcal{S}(()x_i^{\mathsf{R}})) = (p, D)$$

$$\textsc{Env}_{re}[\mathbb{I}, \mathbb{S}](s) \quad := \quad \begin{cases} (\mathsf{T}, \emptyset) \text{ if } s \in \mathbb{S} \text{ or } re = \emptyset \\ \textsc{Env}_{re}^{\mathcal{L} \cup \{\mathbf{D}\}}[\mathbb{I}, \mathbb{S}](s) \end{cases}$$

$$\textsc{Env}_{re}^{L}[\mathbb{I}, \mathbb{S}](s) \quad := \quad \bigcup_{l \in \max(L)} \left(\textsc{Env}_{re}^{\{l' \in L \mid l' < l\}}[\mathbb{I}, \mathbb{S}](s) \lhd \textsc{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](s) \right)$$

$$\textsc{Env}_{re}^{\mathbf{D}}[\mathbb{I}, \mathbb{S}](s) \quad := \quad \begin{cases} (\mathsf{T}, \emptyset) \text{ if } \epsilon \notin re \\ (\mathsf{T}, \mathcal{D}(s)) \end{cases}$$

$$\textsc{Env}_{re}^{l}[\mathbb{I}, \mathbb{S}](s) \quad := \quad \begin{cases} \bot \text{ if } \mathrm{vars}(\mathcal{E}(s)|_l) \neq \emptyset \text{ or } \mathrm{IS}^l[\mathbb{I}](s) = \bot \\ \displaystyle\bigcup_{s' \in \left(\mathrm{IS}^l[\mathbb{I}](s) \cup (\mathcal{E}(s)|_l)\right)} \textsc{Env}_{(\partial_l re)}[\mathbb{I}, \{s\} \cup \mathbb{S}](s') \end{cases}$$

$$\mathrm{IS}^l[\mathbb{I}](s) \quad := \quad \begin{cases} \bot \text{ if } \exists y^{\mathsf{R}}.\, (y^{\mathsf{R}} \in (\mathcal{N}(s) \backslash \mathbb{I}) \wedge \textsc{Res}[\mathbb{I}](y^{\mathsf{R}}) = \bot) \\ \left\{ s' \mid y_i^{\mathsf{R}} \in (\mathcal{N}(s) \backslash \mathbb{I}) \wedge y_j^{\mathsf{D}} \in \textsc{Res}[\mathbb{I}](y_i^{\mathsf{R}}) \wedge y_j^{\mathsf{D}} \mathrel{-\!\!\!\triangleright} s' \right\} \end{cases}$$

with the following auxiliary definitions:

$$\partial_l\, re \qquad \text{Brzozowski (1964) derivative w.r.t. label } l$$

$$\max(L) \quad := \quad \{l \in L \mid \nexists l' \in L.\, l < l'\}$$

and the shadowing and union operators on environments

$$(p_1, D_1) \lhd (p_2, D_2) \quad := \quad \begin{cases} (p_1, D_1) \text{ if } D_1 \neq \mathsf{T} \\ (p_2, D_1 \cup \{x_i^{\mathsf{D}} \in D_2 \mid \nexists x^{\mathsf{D}} \in D_1\}) \end{cases}$$

$$\bigcup_{i \in I} (p_i, D_i) \quad := \quad \begin{cases} (\mathsf{T}, D) \text{ if } \forall i \in I.\, p_i = \mathsf{T} \\ (\mathsf{P}, D) \end{cases}$$
$$\text{where } D = \{x_k^{\mathsf{D}} \in \textstyle\bigcup_i D_i \mid \forall j \in I.\, (p_j = \mathsf{T} \vee \exists x^{\mathsf{D}} \in D_j)\}$$

Figure 2.6: Name resolution algorithm

- A declaration is *visible* in a scope s, written as $\vdash_{\mathcal{G}} s \mapsto x_i^{\mathsf{D}}$, if (1) the declaration is reachable from s, and (2) there is no other declaration x_j^{D} that is reachable from s with a shorter path.
- A reference *resolves* to a declaration, written as $\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}$, if the declaration is visible from the scope of the reference, i.e. if $\mathcal{S}(x_i^{\mathsf{R}}) = s$ and $\vdash_{\mathcal{G}} s \mapsto x_j^{\mathsf{D}}$ hold.

The algorithm shown in Figure 2.6 performs name resolution in a – possibly incomplete – scope graph. Like the calculus it is parametrized by the set of labels $\mathcal{L}$, the label order $<$, and the regular expression $\mathcal{E}$. Like the original resolution algorithm (Neron et al., 2015), it allows us to resolve individual references, using the function RES, or calculate all visible declarations in a scope, using the function ENV. Compared to the original algorithm, there are three main modifications:

- Instead of hard-coded parent and import edges, the algorithm works with arbitrary edge labels. The label order is used in ENV_{re}^{L} to ensure an environment for label l is properly shadowed by the environments of smaller labels.
- The algorithm checks path well-formedness by performing a step-by-step match against $\mathcal{E}$. This is done by calculating the Brzozowski (1964) derivative, written as $\partial_l re$, at every l-labeled step. If a step is invalid, $\partial_l = \emptyset$, and the calculation backtracks. If the step is valid, $\partial_l re$ is a regular expression matching any valid remaining path. The current path matches the original $\mathcal{E}$ if re matches the empty string, i.e. $\epsilon \in re$, which is therefore the condition for returning declarations.
- In an incomplete graph, the algorithm tries to include as many declarations as possible in the calculated environments. A flag is used to indicate whether an environment is partial (P), or total (T). In a total environment, no new declarations can appear because of variable instantiation. In a partial environment, new declarations may appear after variable instantiation. However, if a partial environment contains any declaration for x, it is complete with respect to the name x, i.e. instantiation variables cannot result in new declarations of x in the environment.

2.3 Type Checking with Constraints

Constraint-based type checking is an approach to type checking, based on generating and solving constraints. Constraints are typically generated from a program term by a language-specific, syntax-directed function. A constraint solver is used to test if the constraints are satisfiable, and to calculate a solution – usually a type assignment for the original program.

There are several advantages to separating constraint generation and resolution. First, the solver only needs to deal with the intermediate constraint language, which is usually small compared to the language of the program. This

name	x, y		
position	i, j		
scope	s	$:=$	i^{S}
declaration	d	$:=$	x_i^{D}
reference	r	$:=$	x_i^{R}
scope graph	$\mathcal{G}$		
declaration scope	$\mathcal{S}(r)$		
reference scope	$\mathcal{S}(d)$		
associated scope	$\mathcal{A}(d)$		
declarations in scope	$\mathcal{D}(s)$	$:=$	$\{d \mid \mathcal{S}(d) = s\}$
references in scope	$\mathcal{R}(s)$	$:=$	$\{d \mid \mathcal{S}(d) = s\}$
direct l-edges from scope	$\mathcal{E}_l(s)$		
named l-edges from scope	$\mathcal{N}_l(s)$		
direct l-edge scopes	$s_l^{\blacktriangleright}$	$:=$	$\{s' \mid (s, s') \in \mathcal{E}_l(s)\}$
named l-edge scopes	$s_l^{\triangleright}$	$:=$	$\{s' \mid (s, r) \in \mathcal{N}_l(s) \wedge r \mapsto d \wedge \mathcal{A}(d) = s'\}$
set of labels	$\mathcal{L}$	$:=$	$\ldots$
edge label	l	$\in$	$\{\mathbf{D}\} \cup \mathcal{L}$
resolution step	st	$:=$	$\mathbf{D}(d) \mid \mathbf{E}(l, s) \mid \mathbf{N}(l, r, s)$
resolution path	p	$:=$	$st \cdot p \mid \epsilon$
partial label order	$l_1 < l_2$	$:=$	$\dfrac{l \neq \mathbf{D}}{\mathbf{D} < l} \quad \cdots$
visibility order	$p_1 < p_2$	$:=$	$\dfrac{label(st_1) < label(st_2)}{st_1 \cdot p_1 < st_2 \cdot p_2} \quad \dfrac{p_1 < p_2}{st \cdot p_1 < st \cdot p_2}$
regex over labels	$\mathcal{E}$	$:=$	$\emptyset \mid \epsilon \mid l \mid \mathcal{E}^* \mid \mathcal{E} \cdot \mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} \,\&\, \mathcal{E}$
well-formedness	$\mathrm{WF}(p)$	$\equiv$	$labels(p) \in \mathcal{E}$
reachability relation	$\vdash s \rightarrowtail d$		
visibility relation	$\vdash s \mapsto d$		
resolution relation	$\vdash r \mapsto d$		

The calculus is parametrized in the set of labels $\mathcal{L}$, the label order $<$, and the well-formedness regular expression $\mathcal{E}$.

Figure 2.7: Notation overview of the resolution calculus

$$
\begin{array}{rrcl}
 & \mathit{expr} & := & \mathit{id} \mid \texttt{fun}(\mathit{id})\{\mathit{expr}\} \mid \mathit{expr}\ \mathit{expr} \\
\text{type} & t & := & \tau \mid \texttt{Fun}[t,t] \\
\text{constraint} & C & := & C \wedge C \mid t \overset{?}{=} t \\
\text{type environment} & \Gamma & & \\
\text{substitution} & \varphi,\sigma & & \\
\text{satisfiability} & \varphi \models C & \equiv & \dfrac{\varphi \models C_1 \qquad \varphi \models C_2}{\varphi \models C_1 \wedge C_2} \quad \dfrac{t_1\varphi = t_2\varphi}{\varphi \models t_1 \overset{?}{=} t_2}
\end{array}
$$

The constraint generation function is given by:

$$
\begin{array}{rcl}
[\![\Gamma \vdash x : t]\!] & := & \Gamma(x) \overset{?}{=} t \\
[\![\Gamma \vdash \texttt{fun}(x)\{e\} : t]\!] & := & [\![\Gamma, (x : \tau_1) \vdash e : \tau_2]\!] \wedge t \overset{?}{=} \texttt{Fun}[\tau_1, \tau_2] \\
[\![\Gamma \vdash e_1\ e_2 : t]\!] & := & [\![\Gamma \vdash e_1 : \texttt{Fun}[\tau, t]]\!] \wedge [\![\Gamma \vdash e_2 : \tau]\!]
\end{array}
$$

Variables τ that appear free are assumed to be fresh.

Figure 2.8: Syntax and constraints of the simply-typed λ-calculus

keeps the complexity of the program language local to the constraint generation function. Second, the constraint language is usually more stable, therefore changes to the program language do not necessarily require changes in the solver. Finally, because the constraint language is independent of a specific program language, multiple languages can use the same constraint language and solver for type checking.

We introduce constraint-based type checking using a subset of LMR, shown in Figure 2.8, that corresponds to the simply-typed λ-calculus (cf. Rémy, 2015, Section 5.2). The language has a function type, but base types are omitted from the presentation. We will see that they are not essential for constraint generation or resolution.

2.3.1 Constraint Generation

Constraints are generated from a program using a syntax directed function. Figure 2.8 shows the syntax of constraints and types, as well as the constraint generation function. Constraints are conjunctions of equality constraints between types. Constraint variables – sometimes called unification variables – are used for unknown types, which need to be resolved.

The constraint generation function matches a typing relation $\Gamma \vdash e : t$. The idea is that e has type t in the context of Γ, if the generated constraint is satisfiable. The typing environment Γ contains the types for identifiers that are in scope. A function adds its argument to the type environment, which is then used for the function body. References are resolved by a lookup in the environment.

```
(fun (f) { fun (x) { f x } }) (fun (x) { x })
```

(a) Program

$$
\begin{aligned}
\tau_1 &\overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (1)\\
\tau_2 &\overset{?}{=} \tau_3 && (2)\\
\mathtt{Fun}[\tau_1,\tau] &\overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5] && (3)\\
\tau_5 &\overset{?}{=} \mathtt{Fun}[\tau_6,\tau_7] && (4)\\
\tau_6 &\overset{?}{=} \tau_8 && (5)\\
\tau_4 &\overset{?}{=} \mathtt{Fun}[\tau_8,\tau_7] && (6)
\end{aligned}
$$

(b) Constraints

$$
\begin{array}{ll}
\tau \mapsto \mathtt{Fun}[\tau_7,\tau_7] & \tau_1 \mapsto \mathtt{Fun}[\tau_7,\tau_7]\\
\tau_2 \mapsto \tau_7 & \tau_3 \mapsto \tau_7\\
\tau_4 \mapsto \mathtt{Fun}[\tau_7,\tau_7] & \tau_5 \mapsto \mathtt{Fun}[\tau_7,\tau_7]\\
\tau_6 \mapsto \tau_7 & \tau_8 \mapsto \tau_7
\end{array}
$$

(c) Solution

Figure 2.9: Example for the simply-typed λ-calculus

Figure 2.9 shows an example program. It applies a function to the identity function. The function applies its first argument to its second argument. The whole expression reduces to an identity function. We will use the constraint generation function shown in Figure 2.8, to generate the constraints for this program. We assume there are no identifiers in global scope, so we start with the empty type environment $\Gamma = \emptyset$. We assign the type variable τ to the whole program. Therefore, we apply our constraint generation function to $\emptyset \vdash e : \tau$. We show the constraint generation step by step, marking the parts that changed:

$$
\begin{aligned}
C &\equiv \llbracket \emptyset \vdash \mathtt{(fun(f)\{fun(x)\{f\ x\}\})\ (fun(x)\{x\})} : \tau \rrbracket && (1)\\
&\equiv \llbracket \emptyset \vdash \mathtt{fun(f)\{fun(x)\{f\ x\}\}} : \mathtt{Fun}[\tau_1,\tau] \rrbracket \wedge \llbracket \emptyset \vdash \mathtt{fun(x)\{x\}} : \tau_1 \rrbracket && (2)\\
&\equiv \llbracket \emptyset \vdash \mathtt{fun(f)\{fun(x)\{f\ x\}\}} : \mathtt{Fun}[\tau_1,\tau] \rrbracket \wedge \llbracket (\mathtt{x} : \tau_2) \vdash \mathtt{x} : \tau_3 \rrbracket\\
&\qquad \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (3)\\
&\equiv \llbracket \emptyset \vdash \mathtt{fun(f)\{fun(x)\{f\ x\}\}} : \mathtt{Fun}[\tau_1,\tau] \rrbracket \wedge \tau_2 \overset{?}{=} \tau_3 \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (4)\\
&\equiv \llbracket (\mathtt{f} : \tau_4) \vdash \mathtt{fun(x)\{f\ x\}} : \tau_5 \rrbracket \wedge \mathtt{Fun}[\tau_1,\tau] \overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5] \wedge \tau_2 \overset{?}{=} \tau_3\\
&\qquad \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (5)\\
&\equiv \llbracket (\mathtt{f} : \tau_4, \mathtt{x} : \tau_6) \vdash \mathtt{f\ x} : \tau_7 \rrbracket \wedge \tau_5 \overset{?}{=} \mathtt{Fun}[\tau_6,\tau_7] \wedge \mathtt{Fun}[\tau_1,\tau] \overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5]\\
&\qquad \wedge \tau_2 \overset{?}{=} \tau_3 \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (6)\\
&\equiv \llbracket (\mathtt{f} : \tau_4, \mathtt{x} : \tau_6) \vdash \mathtt{f} : \mathtt{Fun}[\tau_8,\tau_7] \rrbracket \wedge \llbracket (\mathtt{f} : \tau_4, \mathtt{x} : \tau_6) \vdash \mathtt{x} : \tau_8 \rrbracket\\
&\qquad \wedge \tau_5 \overset{?}{=} \mathtt{Fun}[\tau_6,\tau_7] \wedge \mathtt{Fun}[\tau_1,\tau] \overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5] \wedge \tau_2 \overset{?}{=} \tau_3\\
&\qquad \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (7)\\
&\equiv \llbracket (\mathtt{f} : \tau_4, \mathtt{x} : \tau_6) \vdash \mathtt{f} : \mathtt{Fun}[\tau_8,\tau_7] \rrbracket \wedge \tau_6 \overset{?}{=} \tau_8 \wedge \tau_5 \overset{?}{=} \mathtt{Fun}[\tau_6,\tau_7]\\
&\qquad \wedge \mathtt{Fun}[\tau_1,\tau] \overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5] \wedge \tau_2 \overset{?}{=} \tau_3 \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (8)\\
&\equiv \tau_4 \overset{?}{=} \mathtt{Fun}[\tau_8,\tau_7] \wedge \tau_6 \overset{?}{=} \tau_8 \wedge \tau_5 \overset{?}{=} \mathtt{Fun}[\tau_6,\tau_7] \wedge \mathtt{Fun}[\tau_1,\tau] \overset{?}{=} \mathtt{Fun}[\tau_4,\tau_5]\\
&\qquad \wedge \tau_2 \overset{?}{=} \tau_3 \wedge \tau_1 \overset{?}{=} \mathtt{Fun}[\tau_2,\tau_3] && (9)
\end{aligned}
$$

Note that the identifiers of the program do not appear in the resulting constraint. Looking up the types of identifiers is built into the constraint generation function. For a lexically scoped language like this, this works well, since the scoping structure matches the recursive calls in the constraint generation function. This can quickly become a problem for languages with more complicated, non-lexical, binding patterns. As we will see in the next chapter, this problem can be mitigated by lifting name resolution to the constraint problem. This enables us to keep the constraint generation simple, while still allowing complicated binding patterns.

2.3.2 Constraint Solving by Unification

Constraint solving involves finding an assignment to the type variables, such that the constraints are satisfied. For equality constraints, this process is known as first-order unification.

Satisfiability is denoted as $\varphi \models C$, where φ is a substitution, mapping variables to ground types. The satisfiability relation is shown in Figure 2.8. A conjunction $C_1 \wedge C_2$ is satisfied if the substitution satisfies both C_1 and C_2. An equality constraint is satisfiable if the two side are equal after application of φ. Applying a substitution σ to a term t, written in postfix notation as $t\sigma$ is defined in the usual way as:

$$\begin{aligned} \alpha\sigma &:= \begin{cases} t' & \text{if } \sigma(\alpha) = t' \\ \alpha & \text{otherwise} \end{cases} \\ f(t_1, \ldots, t_n)\sigma &:= f(t_1\sigma, \ldots, t_n\sigma) \end{aligned}$$

The above definition of substitution, and indeed the whole process of unification, assumes that our types are terms in a Herbrand universe. That is, terms are recursively generated over a signature that defines a set of function symbols f, g and an arity for each function symbol. Terms are formed by applying a function symbol to a number of term arguments, where that number equals the arity of the symbol. For example for a symbol f with an arity of three, we could form a term $f(t_1, t_2, t_3)$, where the t_i's denote other terms. Concretely, the set of signatures for our types would be $\{\texttt{Fun}\}$, and the symbol `Fun` has an arity of two.

Before we look at the unification algorithm, we note that application of a substitution to another substitution, is interpreted as a composition of substitutions. Applying the composition to a term should yield the same result as applying the substitutions consecutively. So, given two substitutions σ, σ' and their composition $\sigma\sigma'$, it holds that

$$\forall t.\, t(\sigma\sigma') = (t\sigma)\sigma'$$

An algorithm solving equality constraints by first-order unification is shown in Figure 2.10. This algorithm is formulated as a set of rewrite rules over a

$$\boxed{C;\varphi \longrightarrow C;\varphi}$$

$$\begin{array}{rcll}
s \overset{?}{=} s \wedge C \,;\, \varphi & \longrightarrow & C \,;\, \varphi & \\
f(t_1,\ldots,t_n) \overset{?}{=} f(t'_1,\ldots,t'_n) \wedge C \,;\, \varphi & \longrightarrow & t_1 \overset{?}{=} t'_1 \wedge \ldots t_n \overset{?}{=} t'_n \wedge C \,;\, \varphi & \\
f(t_1,\ldots,t_m) \overset{?}{=} g(t'_1,\ldots,t'_n) \wedge C \,;\, \varphi & \longrightarrow & \bot & (\text{if } f \neq g) \\
t \overset{?}{=} \tau \wedge C \,;\, \varphi & \longrightarrow & \tau \overset{?}{=} t \wedge C \,;\, \varphi & \\
& & & (\text{if } t \text{ not a variable}) \\
\tau \overset{?}{=} t \wedge C \,;\, \varphi & \longrightarrow & \bot \quad (\text{if } \tau \in \text{vars}(t) \text{ and } \tau \neq t) & \\
\tau \overset{?}{=} t \wedge C \,;\, \varphi & \longrightarrow & C\{\tau \mapsto t\} \,;\, \varphi\{\tau \mapsto t\} & \\
& & & (\text{if } \tau \notin \text{vars}(t))
\end{array}$$

Figure 2.10: Unification algorithm

constraint and a substitution. It works by eliminating or simplifying the constraint, while building the substitution – also called a unifier. We will show how the rules apply to some of the constraints from our example.

Constraint (1) results in the elimination of τ_1, resulting in the substitution $\tau_1 \mapsto \texttt{Fun}[\tau_2,\tau_3]$. After the application of this substitution, constraint (3) will have become $\texttt{Fun}[\texttt{Fun}[\tau_2,\tau_3],\tau] \overset{?}{=} \texttt{Fun}[\tau_4,\tau_5]$. Since their function symbols match, it can be simplified, introducing two new constraints, $\texttt{Fun}[\tau_2,\tau_3] \overset{?}{=} \tau_4$ (7) and $\tau \overset{?}{=} \tau_5$. Constraint (7) can be oriented $\tau_4 \overset{?}{=} \texttt{Fun}[\tau_2,\tau_3]$, after which τ_4 can be eliminated. This process continues until an error occurs, or all constraints are eliminated. There are two kinds of errors that can occur during unification. One is when the two function symbols in the equation are different, e.g. $\texttt{Fun}[t_1,t_2] \overset{?}{=} \texttt{Nat}$ – assuming a base type Nat for the moment. The other is with a constraint like $\tau \overset{?}{=} \texttt{Fun}[t,\tau]$, where the variable on the left occurs as a strict subterm of the right-hand side. In a syntactic model like ours, this is impossible, the variable cannot be a subterm of itself. Therefore, the substitution produced by $\mathcal{U}$ are idempotent. A substitution is idempotent if repeated application has the same effect as applying it once, i.e. $t\sigma\sigma = t\sigma$. Composing an idempotent substitution with itself, has no effect, i.e. $\sigma\sigma = \sigma$.

A possible solution is shown in the right of Figure 2.9. We say possible solution, because solutions from unification are usually not unique. For example, the constraint $t_7 \overset{?}{=} \tau_8$ can be solved by the substitutions $\tau_7 \mapsto \tau_8$, or $\tau_8 \mapsto \tau_7$. The algorithm does however ensure that the calculated unifier is a most general one. This means that every possible unifier is an instance of the one returned by the algorithm:

$$\forall C\varphi.\, \big(C;\emptyset \longrightarrow \emptyset;\varphi \implies \big(\varphi \models C \wedge \forall \varphi' \models C.\, \exists \sigma.\, \varphi\sigma = \varphi'\big)\big)$$

Chapter 3

Constraint Language

In this chapter, we introduce the syntax and semantics of the constraint language, and show how it is used to express static analysis for LMR. The main idea of the constraint language is, that it contains name binding constraints, next to the typing constraints we already saw in the previous chapter. Specifically, the language provides primitive constraints for (1) type checking, (2) construction of the scope graph, (3) name resolution, and (4) nominal subtyping. Larger constraints are built as conjunctions, written as $C \wedge C$, of these primitive constraints.

In the first section we introduce the constraint syntax. We start by explaining our general approach to constraint-based static analysis. Then we describe the primitive constraints, following an example program. We finish the section by showing the general structure of a solution to a constraint problem. In the second section we define a satisfaction relation, which formalizes the meaning of the constraints, and describes what a solution to a constraint problem is. For reference, we include a complete overview of the constraint syntax in Figure 3.1.

3.1 Constraints for Static Analysis

Analysis of a program follows the same two-phase approach that we have seen for the simply-typed λ-calculus in the previous chapter. First, constraints are generated using a constraint generation function, specific to the programming language. Second, the generated constraints are solved using a language-independent constraint solver. The constraint generation function has the following form:

$$[\![e : t]\!]^{sort}_{args}$$

It matches on a typing judgment $e : t$, where e is a program expression, and t is the type of the expression. Because t is often the type expected by the context, we also call it the expected type of e. Other arguments, written as subscripts, can be passed to the constraint generation function. This is used to pass the current scope, or types other then the type of e itself. We often use the syntactic sort name as a superscript, to distinguish between similar constructs more easily. In the case of program terms that do not represent expressions,

constraint variables			$\alpha, \delta, \sigma, \tau$	
constraint	C	$:=$	True	(true)
		$\mid$	$C \wedge C$	(conjunction)
		$\mid$	$C^{T} \mid C^{\mathcal{G}} \mid C^{\mapsto} \mid C^{\leq}$	
type constraint	C^{T}	$:=$	$t \stackrel{?}{=} t$	(term equality)
		$\mid$	$d : t$	(type of declaration)
term	t	$:=$	$f(t, \ldots, t) \mid \alpha$	
type	t	$:=$	τ	
symbol	f, g			
scope graph constraint	$C^{\mathcal{G}}$	$:=$	$s \longrightarrow d$	(declaration in scope)
		$\mid$	$r \longrightarrow s$	(reference in scope)
		$\mid$	$s \xrightarrow{l} s$	(direct edge)
		$\mid$	$s \overset{l}{\rightarrowtriangle} r$	(named edge)
		$\mid$	$d \rightarrowtriangle s$	(associated scope)
declaration	d	$:=$	${}^{ns}x_i^{\mathsf{D}} \mid \delta$	
reference	r	$:=$	${}^{ns}x_i^{\mathsf{R}}$	
scope	s	$:=$	$i^{\mathsf{S}} \mid \varsigma$	
edge label	l			
namespace	ns			
position	i, j			
name	x, y			
resolution constraint	$C^{\mapsto}$	$:=$	$r \mapsto d$	(resolve name)
		$\mid$	$d \rightsquigarrow s$	(associated scope)
		$\mid$	$!N$	(distinct names)
		$\mid$	$N \subsetneqq N$	(subset names)
		$\mid$	$N \simeq N$	(same names)
name set	N	$:=$	${}^{ns}\overline{\mathcal{D}}(s)$	(declared names)
		$\mid$	${}^{ns}\overline{\mathcal{R}}(s)$	(referred names)
		$\mid$	${}^{ns}\overline{\mathcal{V}}(s)$	(visible names)
		$\mid$	${}^{ns}\overline{\mathcal{W}}(s)$	(reachable names)
subtyping constraint	$C^{\leq}$	$:=$	$t <: t$	(supertype definition)
		$\mid$	$t \leq: t$	(subtype)

Figure 3.1: Syntax of the constraint language

and therefore have no type assigned to them, we simply match on the term, written as $[\![e]\!]^{sort}_{args}$. Although not necessary to understand the rest of this section, we present the complete constraint generation function for LMR in Figures 3.5 and 3.6 for reference.

Our example program is split up in three parts. For each part, we will show the program, scope graph, and constraints, as well as a solution to the constraints. To ease the presentation, we build op our example in such a way, that we only depend on program parts we already presented. However, this is not required by the constraint system.

3.1.1 Record definition

Our example starts by defining two record types, `A` and `B`. Record `A` has one integer field `f`. Record `B` is declared as a subtype of record `A`, and declares one boolean field `g`. The constraints – shown below the program text – are numbered, and we will use those to refer to the constraints from the text. Figure 3.2 shows the program text and the scope graph at the top, and the constraints and a solution at the bottom. We start by explaining the scope graph, then we discuss the constraints for disambiguating names, typing field definitions, and declaring supertypes.

Scope graph The name binding structure of the program is represented by the scope graph, shown to the right of the program. The records correspond to declarations in the program scope 1^{S}. Each records has an associated record scope – 2^{S} for `A`, and 3^{S} for `B` – which contains the field declarations. Since record `B` is a subtype of record `A`, the fields of `A` are imported into the record scope of `B` with a named edge. We use the label **S** we introduced in the previous chapter.

For reasons of presentation, we show the scope graph using the graphical notation, introduced in the previous chapter. However, the scope graph is constructed from the following collected scope graph constraints:

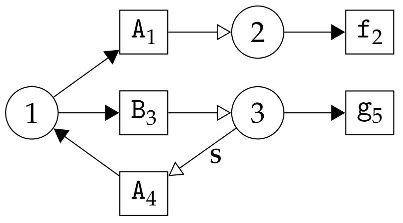
$$\begin{array}{lll}
1^{\mathsf{S}} \longrightarrow \mathtt{A}^{\mathsf{D}}_{1} & \mathtt{A}^{\mathsf{D}}_{1} \rightarrowtriangle 2^{\mathsf{S}} & \mathtt{f}^{\mathsf{D}}_{2} \longrightarrow 2^{\mathsf{S}} \\
1^{\mathsf{S}} \longrightarrow \mathtt{B}^{\mathsf{D}}_{3} & \mathtt{B}^{\mathsf{D}}_{3} \rightarrowtriangle 3^{\mathsf{S}} & \mathtt{g}^{\mathsf{D}}_{5} \longrightarrow 3^{\mathsf{S}} \\
\mathtt{A}^{\mathsf{R}}_{4} \longrightarrow 1^{\mathsf{S}} & 3^{\mathsf{S}} \overset{\mathsf{S}}{\rightarrowtriangle} \mathtt{A}^{\mathsf{R}}_{4} &
\end{array}$$

In general, scope graph constraints represent edges in the graph, and are written similarly to the graphical notation for scope graphs:

- The scope graph of a program is defined with *scope graph* constraints. The constraints are written with the arrows from the graphical notation, but use the notation of the calculus for the endpoints. Thus, $s \longrightarrow x^{\mathsf{D}}_{i}$ denotes a declaration x in scope s, $x^{\mathsf{R}}_{i} \longrightarrow s$ denotes a reference in scope s, $s \xrightarrow{l} s'$ denotes an l-labeled direct edge from scope s to scope s', $x^{\mathsf{D}}_{i} \rightarrowtriangle s$ denotes the associated scope s of declaration x, and $s \overset{l}{\rightarrowtriangle} x^{\mathsf{R}}_{i}$ denotes an l-labeled named edge through reference x.

```
record A_1 {
  f_2 : Int
}
record B_3 extends A_4 {
  g_5 : Bool
}
```

(a) Program



(b) Scope graph

$$
\begin{array}{rr}
!\overline{\mathcal{D}}(1^{\mathsf{S}}) & (1)\\
!\overline{\mathcal{D}}(2^{\mathsf{S}}) & (2)\\
\mathtt{f}_2^{\mathsf{D}} : \tau_1 & (3)\\
\tau_1 \overset{?}{=} \mathtt{Int} & (4)\\
!\overline{\mathcal{D}}(3^{\mathsf{S}}) & (5)\\
!\overline{\mathcal{W}}(3^{\mathsf{S}}) & (6)\\
\mathtt{A}_4^{\mathsf{R}} \mapsto \delta_1 & (7)\\
\mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}}) <: \mathtt{Rec}(\delta_1) & (8)\\
\mathtt{g}_5^{\mathsf{D}} : \tau_2 & (9)\\
\tau_2 \overset{?}{=} \mathtt{Bool} & (10)
\end{array}
$$

(c) Constraints

$$
\begin{array}{rcl}
\varphi & = & \tau_1 \mapsto \mathtt{Int}\\
 & & \tau_2 \mapsto \mathtt{Bool}\\
 & & \delta_1 \mapsto \mathtt{A}_1^{\mathsf{D}}\\
\psi & = & \mathtt{f}_2^{\mathsf{D}} : \mathtt{Int}\\
 & & \mathtt{g}_5^{\mathsf{D}} : \mathtt{Bool}\\
\mathcal{R} & = & \mathtt{A}_4^{\mathsf{R}} \mapsto \mathtt{A}_1^{\mathsf{D}}\\
<_T & = & \{\mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}}) <_T \mathtt{Rec}(\mathtt{A}_1^{\mathsf{D}})\}
\end{array}
$$

(d) Solution

Figure 3.2: Example of record definition

- The constraint syntax allows names to be classified by an optional *name-space*. These are written in prescript, for declarations as ${}^{ns}x_i^{\mathsf{D}}$, and similarly for references as ${}^{ns}x_i^{\mathsf{R}}$. We omit a (possible) namespace when it is irrelevant in the context.

Name disambiguation There are additional restrictions on the declarations and references in the program, that are not expressible in the scope graph. First, all definitions in a program must have distinct names, which is specified with $!\overline{\mathcal{D}}(1^{\mathsf{S}})$ (1). Similarly, we require that the fields of record `A` have distinct name (2), and the same (5) for record `B`. Note that for records, we require that all reachable field names are distinct. This is to ensure that no inherited fields are shadowed. Generally, we can restrict sets of names in the following way:

- A *name set* N is a multi-set of names that corresponds to a set of declarations or references. Given a scope s, we write ${}^{ns}\overline{\mathcal{D}}(s)$ for the names of the declarations in s, ${}^{ns}\overline{\mathcal{R}}(s)$ for the names of the references in s. ${}^{ns}\overline{\mathcal{V}}(s)$ denotes all declarations that are visible in s, either because they are declared in s, or because they are visible through one or more edge steps. We write ${}^{ns}\overline{\mathcal{W}}(s)$ for all declarations that are reachable through s, without considering shadowing. If we want to match on all namespaces, we omit the namespace, e.g. $\overline{\mathcal{D}}(s)$ for all declarations in s regardless of the namespace.

- Several constraints *disambiguate names*. A constraint of the form $!N$ specifies that all names in N must be *distinct*, i.e. every x in N has $\nu(x) = 1$. Given two name sets N_1 and N_2, we can require that one is a *subset* of the other, written as $N_1 \lesssim N_2$. We use $N_1 \simeq N_2$ as syntactic sugar for $N_1 \lesssim N_2 \wedge N_2 \lesssim N_1$, if both sets need to contain the same names.

Typing field definitions Field definitions are annotated with their type. The type of field `f` is the constraint variable τ_1, specified with $\mathtt{f}_2^{\mathsf{D}} : \tau_1$ (3). Because of the type annotation, τ_1 must be an integer type, specified with $\tau_1 \stackrel{?}{=} \mathtt{Int}$ (4). The type of field g is specified in a similar manner (9,10). In general, typing is realized using the following constraints:

- Constraints of the form $x_i^{\mathsf{D}} : t$ specify the *type of a declaration*. Such a constraint can occur multiple times for the same declaration, in which case the types must be equal.

  It should be noted that in the constraint generation for the λ-calculus, the introduction of an identifier x resulted in passing an extended environment $\Gamma, (x : t) \vdash \ldots$ to the recursive call. We use these declaration-type constraints as the replacement for that typing environment.

- Constraints of the form $t_1 \stackrel{?}{=} t_2$ specify *equality* between terms. They are the same as the equality constraints presented in the previous chapter. To be able to use these equality constraints for typing, we require that our types are built as terms.

Declaring supertype Our example declares `B` as a nominal subtype of `A`. To get correct subtyping behavior, we need to make sure that a value of type `B` can be used when a value of type `A` is expected. Using just a name to identify a record type can lead to ambiguities if, for example, multiple modules define a record of the same name. We therefore identify a record type using the unique declaration for that record. For example, we use $\mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}})$ as the type for values of record `B`.

Before we can specify anything about the subtyping of `B`, we need to create the correct type for its supertype. For that we need the declaration of `A`, which we resolve using $\mathtt{A}_4^{\mathsf{R}} \mapsto \delta_1$ (7). The resolved declaration is captured with the constraint variable δ_1. Name resolution is part of the constraint language as follows:

- A *name resolution* constraint, written as $x_i^{\mathsf{R}} \mapsto \delta$, specifies name resolution of the reference in the scope graph. A resolved declaration is unified with the constraint variable δ.

Using the declaration we can specify the fact that `A` is the supertype of `B`, with $\texttt{Rec}(\texttt{B}^{\texttt{D}}_3) <: \texttt{Rec}(\delta_1)$ (8). A nominal type hierarchy is specified as follows:

- A constraint of the form $t_1 <: t_2$ states that t_2 is a *supertype* of t_1. The types form a hierarchy, where every type can have at most one (direct) supertype.
- A *subtype* constraint, written as $t_1 \leq: t_2$, specifies that type t_1 must be a subtype of type t_2. That means that either the two types are equal, or t_2 is a supertype – direct or indirect – of t_1.

Solution to the constraint problem We can consider the constraints as a problem, for which we want to find a solution. A constraint context Δ is a solution to a constraint problem, if it is consistent with the constraints. A constraint context contains (1) a substitution φ that assigns a term to each constraint variable, (2) a type-map ψ that assigns types to declarations, (3) a scope graph $\mathcal{G}$, (4) a name resolution $\mathcal{R}$ that maps references to declarations, and (5) a subtyping relation $<_T$.

We will give a general description of each component, following the presented solution to our example constraints. In the next section we give a precise formulation of the meaning of constraints and the notion of a solution, by defining a satisfaction relation $\Delta \models C$.

We already saw the concept of a substitution, as a solution to equality constraints, in the previous chapter. For example, we can solve (4) with a substitution $\tau_1 \mapsto \texttt{Int}$. In our constraints, variables do not only occur in equality, but also in other constraints. For example, variable δ_1 occurs in the name resolution constraint (7). In this case, name resolution determines that $\delta_1 \mapsto \texttt{A}^{\texttt{D}}_1$ is a valid substitution for δ_1. We define the substitution φ as follows:

- A constraint context Δ contains a *substitution* φ, which is a mapping from constraint variables to ground terms. A substitution φ is a solution if all constraints are satisfied after application of φ.

The type-map ψ maps declarations to types. For example, constraint (3) can be solved by setting the type of $\texttt{f}^{\texttt{D}}_2$ to `Int` in ψ. Note that, to be consistent with the constraints, we also need $\tau_1 \mapsto \texttt{Int}$ as solution to (4).

- A constraint context Δ contains a mapping ψ from declarations to ground types. It is a solution if for every constraint $d : t$, we have $\psi(d\varphi) = t\varphi$. Note that we need to apply the substitution φ from the same context Δ here, because ψ is ground, but variables can occur in the constraint.

We represent a scope graph $\mathcal{G}$ as a set of ground scope graph constraints. Since our scope graph constraints map one-to-one to edges in the graph, we assume that this set of constraints can trivially be interpreted as a scope graph.

- A constraint context Δ contains a ground *scope graph* $\mathcal{G}$. The scope graph is a solution to a constraint problem, if the set of edges in the graph corresponds exactly to the set of scope graph constraints. Note that we may need to apply the substitution φ again, since variables can occur in the constraints.

A name resolution $\mathcal{R}$ maps references to the declarations they resolve to. Since name resolution can be ambiguous, $\mathcal{R}$ represents a resolution choice. In our example, the only the reference $\mathtt{A}_4^{\mathsf{R}}$ resolves to $\mathtt{A}_1^{\mathsf{D}}$, resulting in a resolution $\mathtt{A}_4^{\mathsf{R}} \mapsto \mathtt{A}_1^{\mathsf{D}}$.

- A constraint context Δ contains a *resolution* $\mathcal{R}$, mapping references to ground declarations. A resolution $\mathcal{R}$ is a solution if it is consistent with the name resolution constraints, again after application of φ.

Finally, we have a subtyping relation $<_T$. For our example, $t <_T t$ for every type t, and because of (8), $\mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}}) <_T \mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}})$.

- The constraint context Δ contains a *subtyping relation* $<_T$ on ground types. The subtyping relation is a transitive closure over types that models a forest, i.e. every type has at most one direct supertype. The subtyping relation is a solution, if it is consistent with the supertype constraints, i.e. if, for every constraint $t_1 <: t_2$, t_2 is the parent of t_1 in the type hierarchy.

3.1.2 Record instantiation

The program in Figure 3.3 creates an instance of record `B` and assigns it to the variable `a`. The declaration of the variable `a` has type τ_6 (11), which is fixed (14) as a record type, determined by the record reference `A` (13). Similarly, the type τ_7 of the **`new`** expression is the record type for `B` (15-16). The type of the expression is required to be a subtype of the type of `a` (12).

The new record instance must initialize all the fields of the record. Scope 5^{S}, which imports the record scope, contains the field references to `f` and `g`. We require that all declarations in the record are initialized, by checking that the referred names are the same as the imported field names (17). By requiring that every declaration is referred to only once (18), we ensure that every field is initialized only once. Each field is resolved to its declaration (19,23), and we check that the types of the assigned expressions (22,26) are subtypes (21,25) of the types of the declarations (20,24).

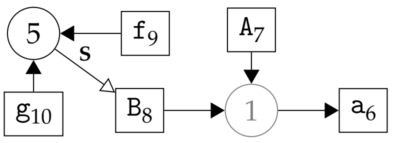
The solution is also presented in Figure 3.3. Recall that the solution to the whole example program so far is the union of both solutions and scope graphs. For example, the resolution of the references only works because scope 1^{S} is the same as scope 1^{S} in Figure 3.2.

3.1.3 Field access

In Figure 3.4 a variable `n` is defined. We assign it the value of field `f` of record instance `a`. To make the field access work, we will need the full machinery of incomplete scope graphs, which we introduced in the previous chapter. Constraints (27) and (28-29) specify the type of `n` and `a` respectively. Resolution

```
def a6 : A7 =
  new B8 {
    f9 = 1,
    g10 = true
  }
```

(a) Program



(b) Scope graph

$$
\begin{array}{lr lr lr}
\mathtt{a}_6^{\mathsf{D}} : \tau_6 & (11) & \overline{\mathcal{R}}(6^{\mathsf{S}}) \simeq \overline{\mathcal{V}}(6^{\mathsf{S}}) & (17) & \mathtt{g}_{10}^{\mathsf{R}} \mapsto \delta_6 & (23)\\
\tau_7 \leq: \tau_6 & (12) & !\overline{\mathcal{R}}(6^{\mathsf{S}}) & (18) & \delta_6 : \tau_{10} & (24)\\
\mathtt{A}_7^{\mathsf{R}} \mapsto \delta_3 & (13) & \mathtt{f}_9^{\mathsf{R}} \mapsto \delta_5 & (19) & \tau_{11} \leq: \tau_{10} & (25)\\
\tau_6 \stackrel{?}{=} \mathtt{Rec}(\delta_3) & (14) & \delta_5 : \tau_8 & (20) & \tau_{11} \stackrel{?}{=} \mathtt{Bool} & (26)\\
\mathtt{B}_8^{\mathsf{R}} \mapsto \delta_4 & (15) & \tau_9 \leq: \tau_8 & (21) & & \\
\tau_7 \stackrel{?}{=} \mathtt{Rec}(\delta_4) & (16) & \tau_9 \stackrel{?}{=} \mathtt{Int} & (22) & &
\end{array}
$$

(c) Constraints

$$
\begin{array}{lcll lcll}
\varphi & = & \tau_6 \mapsto \mathtt{Rec}(\mathtt{A}_1^{\mathsf{D}}) & \tau_7 \mapsto \mathtt{Rec}(\mathtt{A}_1^{\mathsf{D}}) & \psi & = & \mathtt{a}_9^{\mathsf{D}} : \mathtt{Rec}(\mathtt{A}_1^{\mathsf{D}}) & \\
 & & \tau_8 \mapsto \mathtt{Rec}(\mathtt{B}_3^{\mathsf{D}}) & \tau_9 \mapsto \mathtt{Int} & & & & \\
 & & \tau_{10} \mapsto \mathtt{Bool} & \tau_{11} \mapsto \mathtt{Bool} & \mathcal{R} & = & \mathtt{A}_7^{\mathsf{R}} \mapsto \mathtt{A}_1^{\mathsf{D}} & \mathtt{B}_8^{\mathsf{R}} \mapsto \mathtt{B}_3^{\mathsf{D}}\\
 & & \delta_3 \mapsto \mathtt{A}_1^{\mathsf{D}} & \delta_4 \mapsto \mathtt{B}_3^{\mathsf{D}} & & & \mathtt{f}_9^{\mathsf{R}} \mapsto \mathtt{f}_2^{\mathsf{D}} & \mathtt{g}_{10}^{\mathsf{R}} \mapsto \mathtt{g}_5^{\mathsf{D}}\\
 & & \delta_5 \mapsto \mathtt{f}_2^{\mathsf{D}} & \delta_6 \mapsto \mathtt{g}_5^{\mathsf{D}} & & & &
\end{array}
$$

(d) Solution

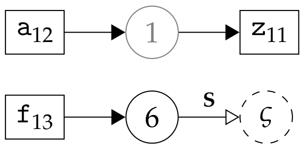
Figure 3.3: Example *(cont.)* of record instantiation

of the field `f`, requires the record scope of the type of `a`. The declaration corresponding to the record type is captured in δ_8 (29), and we use a constraint $\delta_8 \rightsquigarrow \varsigma$ (30) to get the associated scope of the declaration. As we can see in the scope graph, scope 6^{S} imports the variable scope ς. Once δ_8 is resolved to $\mathtt{A}_1^{\mathsf{D}}$, and ς instantiated to 2^{S}, we can resolve `f` to $\mathtt{f}_2^{\mathsf{D}}$. The type of `n` is equal to the type of the field, so we use τ_{12} for the field type (32) without the need for an extra equality constraint.

- An *associated scope* constraint, written as $x_i^{\mathsf{D}} \rightsquigarrow \varsigma$, finds the associated scope of the declaration in the scope graph, and unifies it with the constraint variable ς.

```
12  def n₁₁ = a₁₂.f₁₃
```

(a) Program



(b) Scope graph

$$\begin{aligned}
\mathtt{z}^{\mathsf{D}}_{11} &: \tau_{12} && (27)\\
\mathtt{a}^{\mathsf{R}}_{12} &\mapsto \delta_7 && (28)\\
\delta_7 &: \mathtt{Rec}(\delta_8) && (29)\\
\delta_8 &\rightsquigarrow \varsigma && (30)\\
\mathtt{f}^{\mathsf{R}}_{13} &\mapsto \delta_9 && (31)\\
\delta_9 &: \tau_{12} && (32)
\end{aligned}$$

(c) Constraints

$$\begin{aligned}
\varphi &= \tau_{12} \mapsto \mathtt{Int} \qquad \delta_7 \mapsto \mathtt{a}^{\mathsf{D}}_6\\
&\quad\ \delta_8 \mapsto \mathtt{A}^{\mathsf{D}}_1 \qquad \delta_9 \mapsto \mathtt{f}^{\mathsf{D}}_2\\
&\quad\ \varsigma \mapsto 2^{\mathsf{S}}\\
\psi &= \mathtt{z}^{\mathsf{D}}_{11} : \mathtt{Int}\\
\mathcal{R} &= \mathtt{a}^{\mathsf{R}}_{12} \mapsto \mathtt{a}^{\mathsf{D}}_6 \qquad \mathtt{f}^{\mathsf{R}}_{13} \mapsto \mathtt{f}^{\mathsf{D}}_2
\end{aligned}$$

(d) Solution

Figure 3.4: Example *(cont.)* of field access

$$\boxed{[\![t]\!] := C}$$

$$
\begin{aligned}
[\![D]\!]^{prog} &:= [\![D]\!]^{decl^*}_{s} \land \,!\overline{\mathcal{D}}(s) \\[1em]
[\![\texttt{module}\ x_i\ \texttt{\{}\ D\ \texttt{\}}]\!]^{decl}_{s} &:= s \longrightarrow {}^{M}x_i^{\mathsf{D}} \land {}^{M}x_i^{\mathsf{D}} \rightarrowtriangle s' \land s' \xrightarrow{\mathbf{P}} s \land [\![D]\!]^{decl^*}_{s'} \\
&\qquad \land \,!\overline{\mathcal{D}}(s') \\
[\![\texttt{import}\ x_i]\!]^{decl}_{s} &:= {}^{M}x_i^{\mathsf{R}} \longrightarrow s \land s \xrightarrow{\mathbf{I}}\!\triangleright\, {}^{M}x_i^{\mathsf{R}} \\
[\![\texttt{def}\ b]\!]^{decl}_{s} &:= [\![b]\!]^{bind}_{s} \\
[\![\texttt{record}\ x_i\ s\ \texttt{\{}\ F\ \texttt{\}}]\!]^{decl}_{s} &:= s \longrightarrow {}^{R}x_i^{\mathsf{D}} \land {}^{R}x_i^{\mathsf{D}} \rightarrowtriangle s' \land [\![s]\!]^{super}_{s,s',\texttt{Rec}({}^{R}x_i^{\mathsf{D}})} \\
&\qquad \land [\![F]\!]^{fdecl^*}_{s,s'} \land \,!\overline{\mathcal{W}}(s') \\[1em]
[\![\texttt{extends}\ x_i]\!]^{super}_{s,s_r,t} &:= {}^{R}x_i^{\mathsf{R}} \longrightarrow s \land s_r \xrightarrow{\mathbf{S}}\!\triangleright\, {}^{R}x_i^{\mathsf{R}} \land {}^{R}x_i^{\mathsf{R}} \mapsto \delta \\
&\qquad \land t <: \texttt{Rec}(\delta) \\
[\![\epsilon]\!]^{super}_{s,s_r} &:= \mathsf{True} \\[1em]
[\![x_i\ \texttt{=}\ e]\!]^{bind}_{s} &:= s \longrightarrow {}^{V}x_i^{\mathsf{D}} \land {}^{V}x_i^{\mathsf{D}} : \tau \land [\![e : \tau]\!]^{expr}_{s} \\
[\![x_i\ \texttt{:}\ ta\ \texttt{=}\ e]\!]^{bind}_{s} &:= s \longrightarrow {}^{V}x_i^{\mathsf{D}} \land {}^{V}x_i^{\mathsf{D}} : \tau_1 \land [\![ta]\!]^{tyann}_{s,\tau_1} \land [\![e : \tau_2]\!]^{expr}_{s} \\
&\qquad \land \tau_2 \overset{?}{\leq} \tau_1 \\
[\![x_i\ \texttt{:}\ ta]\!]^{fdecl}_{s,s_r} &:= s_r \longrightarrow {}^{V}x_i^{\mathsf{D}} \land {}^{V}x_i^{\mathsf{D}} : \tau \land [\![ta]\!]^{tyann}_{s,\tau} \\[1em]
[\![\texttt{Bool}]\!]^{tyann}_{s,t} &:= t \overset{?}{=} \texttt{Bool} \\
[\![\texttt{Int}]\!]^{tyann}_{s,t} &:= t \overset{?}{=} \texttt{Int} \\
[\![ta_1\ \texttt{->}\ ta_2]\!]^{tyann}_{s,t} &:= t \overset{?}{=} \texttt{Fun}[\tau_1,\tau_2] \land [\![ta_1]\!]^{tyann}_{s,\tau_1} \land [\![ta_2]\!]^{tyann}_{s,\tau_2} \\
[\![x_i]\!]^{tyann}_{s,t} &:= {}^{R}x_i^{\mathsf{R}} \longrightarrow s \land {}^{R}x_i^{\mathsf{R}} \mapsto \delta \land t \overset{?}{=} \texttt{Rec}(\delta)
\end{aligned}
$$

Any s, τ appearing free is assumed to be chosen fresh. The constraint generation function can also be applied to a set of terms. In that case the function is applied as $[\![T]\!]^{sort^*}_{s} \equiv \bigwedge_{t \in T} [\![t]\!]^{sort}_{s}$.
The parameters to the resolution calculus are the following:

$$
\begin{aligned}
\mathcal{L} &:= \{\mathbf{I}, \mathbf{S}, \mathbf{P}\} \\
l_1 < l_2 &:= \mathbf{I} < \mathbf{P} \quad \mathbf{S} < \mathbf{P} \\
\mathcal{E} &:= \mathbf{P}^* \cdot (\mathbf{S}^* + \mathbf{I}^*)
\end{aligned}
$$

Figure 3.5: Constraint generation for LMR

$$\boxed{[\![t]\!] := C}$$

$$\begin{aligned}
[\![n : t]\!]^{expr}_{s} &:= t \overset{?}{=} \texttt{Int} \\
[\![\texttt{true} : t]\!]^{expr}_{s} &:= t \overset{?}{=} \texttt{Bool} \\
[\![\texttt{false} : t]\!]^{expr}_{s} &:= t \overset{?}{=} \texttt{Bool} \\
[\![x_i : t]\!]^{expr}_{s} &:= {}^{v}x_i^{\mathsf{R}} \longrightarrow s \wedge {}^{v}x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : t \\
[\![e_1 \oplus e_2 : t]\!]^{expr}_{s} &:= [\![e_1 : t_1]\!]^{expr}_{s} \wedge [\![e_2 : t_2]\!]^{expr}_{s} \wedge t \overset{?}{=} t_3 \\
&\qquad (\text{where } \oplus : \texttt{Fun}[t_1,\texttt{Fun}[t_2,t_3]]) \\
[\![e_1 \texttt{ == } e_2 : t]\!]^{expr}_{s} &:= t \overset{?}{=} \texttt{Bool} \wedge [\![e_1 : \tau_1]\!]^{expr}_{s} \wedge [\![e_2 : \tau_2]\!]^{expr}_{s} \\
&\quad \wedge \tau_3 \overset{?}{=} \tau_1 \sqcap \tau_2 \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t]\!]^{expr}_{s} &:= [\![e_1 : \texttt{Bool}]\!]^{expr}_{s} \wedge [\![e_1 : \tau_1]\!]^{expr}_{s} \\
&\quad \wedge [\![e_2 : \tau_2]\!]^{expr}_{s} \wedge t \overset{?}{=} \tau_1 \sqcap \tau_2 \\
[\![\texttt{fun (} x_i \texttt{ : } ta \texttt{) \{ } e \texttt{ \}} : t]\!]^{expr}_{s} &:= s' \xrightarrow{\mathsf{P}} s \wedge s' \longrightarrow {}^{v}x_i^{\mathsf{D}} \wedge {}^{v}x_i^{\mathsf{D}} : \tau_1 \wedge [\![ta]\!]^{tyann}_{s,\tau_1} \\
&\quad \wedge [\![e : \tau_2]\!]^{expr}_{s'} \wedge t \overset{?}{=} \texttt{Fun}[\tau_1,\tau_2] \\
[\![e_1\ e_2 : t]\!]^{expr}_{s} &:= [\![e_1 : \texttt{Fun}[\tau_1,t]]\!]^{expr}_{s} \wedge [\![e_2 : \tau_2]\!]^{expr}_{s} \wedge \tau_2 \leq\!: \tau_1 \\
[\![\texttt{letrec } B \texttt{ in } e : t]\!]^{expr}_{s} &:= s' \xrightarrow{\mathsf{P}} s \wedge [\![B]\!]^{bind}_{s'} \wedge !\overline{\mathcal{D}}(s') \wedge [\![e : t]\!]^{expr}_{s'} \\
[\![\texttt{new } x_i \texttt{ \{ } F \texttt{ \}} : t]\!]^{expr}_{s} &:= {}^{R}x_i^{\mathsf{R}} \longrightarrow s \wedge {}^{R}x_i^{\mathsf{R}} \mapsto \delta \wedge t \overset{?}{=} \texttt{Rec}(\delta) \\
&\quad \wedge s' \longrightarrow\!\triangleright\, {}^{R}x_i^{\mathsf{R}} \wedge \overline{\mathcal{R}}(s') \simeq \overline{\mathcal{V}}(s') \\
&\quad \wedge !\overline{\mathcal{R}}(s') \wedge [\![F]\!]^{fbind^*}_{s,s'} \\
[\![\texttt{with } e_1 \texttt{ do } e_2 : t]\!]^{expr}_{s} &:= [\![e_1 : \texttt{Rec}(\delta)]\!]^{expr}_{s} \wedge \delta \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathsf{P}} s \\
&\quad \wedge s' \xrightarrow{\mathsf{S}} \varsigma \wedge [\![e_2 : t]\!]^{expr}_{s'} \\
[\![e\,.\,x_i : t]\!]^{expr}_{s} &:= [\![e : \texttt{Rec}(\delta)]\!]^{expr}_{s} \wedge \delta \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathsf{S}} \varsigma \\
&\quad \wedge [\![x_i : t]\!]^{expr}_{s'} \\
\\
[\![x_i \texttt{ = } e]\!]^{fbind}_{s,s_r} &:= {}^{v}x_i^{\mathsf{R}} \longrightarrow s_r \wedge {}^{v}x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : \tau_1 \\
&\quad \wedge [\![e : \tau_2]\!]^{expr}_{s} \wedge \tau_2 \leq\!: \tau_1
\end{aligned}$$

Any s, τ appearing free is assumed to be chosen fresh. The constraint generation function can also be applied to a set of terms. In that case the function is applied as $[\![T]\!]^{sort^*}_{s} \equiv \bigwedge_{t \in T} [\![t]\!]^{sort}_{s}$.

Figure 3.6: Constraint generation for LMR *(cont.)*

3.2 Constraint Semantics

In this section we want to make the notions from the previous section precise by giving a formal semantics for our constraint language. Our semantics is expressed as a satisfaction relation $\Delta \models C$, defined by a set of inference rules of the following form:

$$\frac{P_1 \qquad \dots \qquad P_n}{\Delta \models C}$$

A constraint context Δ satisfies a constraint C, written as $\Delta \models C$, if all premises P_1 to P_n are true. The constraint context Δ, which we introduced in the previous section, is formally defined as follows:

DEFINITION 3.1 (Constraint context Δ). A constraint context Δ is a five-tuple

$$\Delta \equiv \langle \varphi, \psi, \mathcal{G}, \mathcal{R}, <_T \rangle$$

with the following components

$$\begin{aligned}
\varphi &: \mathit{Var} \rightarrow \mathit{Term} && \text{(substitution)}\\
\psi &: \mathit{Decl} \rightarrow \mathit{Type} && \text{(types of declarations)}\\
\mathcal{G} & && \text{(scope graph)}\\
\mathcal{R} &: \mathit{Ref} \rightarrow \mathit{Decl} && \text{(resolution)}\\
<_T &: \mathit{Type} \times \mathit{Type} && \text{(subtyping relation)}
\end{aligned}$$

where the substitution φ is a multi-sorted mapping from variables to ground terms, ψ is a ground mapping from declarations to types, $\mathcal{G}$ is a ground scope graph, the resolution $\mathcal{R}$ is a ground mapping from references to declarations, and the subtyping relation $<_T$ is a partial order on ground types.

We will describe the components of the constraint context – and their well-formedness criteria – in more detail later in this section, when we explain the semantic rules that use them. We write $\mathrm{WF}(X)$ to denote the well-formedness predicate for a component of the context, and $\mathrm{WF}(\Delta)$ for well-formedness of the whole context. When we are only interested in a few components of the context, we write $\Delta, X \models \dots$ to make component X from the context explicit without having to state all components. For example $\Delta, \varphi \models \dots$ is equivalent to $\langle \varphi, \cdot, \cdot, \cdot, \cdot \rangle \models \dots$.

The semantic rules for our constraint language are listed in Figure 3.7. We will explain the first two rules here, and the rules for type checking, scope graph construction, name resolution and subtyping in the next subsections. The first two rules deal with trivial truth and conjunctions of constraints. Rule C-TRUE states that every context Δ models True. Rule C-CONJ states that if a context Δ models a constraint C_1, and it also models a constraint C_2, then it models the conjunction $C_1 \wedge C_2$. Because of this general rule for conjunction, it suffices to express the rest of the semantic rules in terms of single constraints.

Constraint context

$$\begin{array}{rccl}
\text{context} & \Delta & \equiv & \langle \varphi, \psi, \mathcal{G}, \mathcal{R}, <_T \rangle \\
\text{substitution} & \varphi & : & \mathit{Var} \rightarrow \mathit{Term} \\
\text{declaration types} & \psi & : & \mathit{Decl} \rightarrow \mathit{Type} \\
\text{scope graph} & \mathcal{G} & & \\
\text{resolution} & \mathcal{R} & : & \mathit{Ref} \rightarrow \mathit{Decl} \\
\text{subtyping relation} & <_T & : & \mathit{Type} \times \mathit{Type} \\
\text{non-strict subtyping relation} & \leq_T & \equiv & t \leq_T t \quad \dfrac{t_1 <_T t_2}{t_1 \leq_T t_2}
\end{array}$$

Constraint semantics $\boxed{\Delta \models C}$

$$\Delta \models \mathsf{True} \qquad \text{(C-TRUE)}$$

$$\frac{x_i \longrightarrow j^{\mathsf{S}} \in \mathcal{G}}{\Delta, \mathcal{G} \models x_i \longrightarrow j^{\mathsf{S}}} \qquad \text{(C-GREF)}$$

$$\frac{\Delta \models C_1 \qquad \Delta \models C_2}{\Delta \models C_1 \wedge C_2} \qquad \text{(C-CONJ)}$$

$$\frac{i^{\mathsf{S}} \xrightarrow{l} s\varphi \in \mathcal{G}}{\Delta, \varphi, \mathcal{G} \models i^{\mathsf{S}} \xrightarrow{l} s} \qquad \text{(C-GEDGE)}$$

$$\frac{t_1\varphi = t_2\varphi}{\Delta, \varphi \models t_1 \overset{?}{=} t_2} \qquad \text{(C-EQUAL)}$$

$$\frac{i^{\mathsf{S}} \overset{l}{\rightarrowtriangle} x_j \in \mathcal{G}}{\Delta, \mathcal{G} \models i^{\mathsf{S}} \overset{l}{\rightarrowtriangle} x_j} \qquad \text{(C-GNEDGE)}$$

$$\frac{\psi(d\varphi) = t\varphi}{\Delta, \varphi, \psi \models d : t} \qquad \text{(C-TYPEOF)}$$

$$\frac{x_i \rightarrowtriangle j^{\mathsf{S}} \in \mathcal{G}}{\Delta, \mathcal{G} \models x_i \rightarrowtriangle j^{\mathsf{S}}} \qquad \text{(C-GASSOC)}$$

$$\frac{i^{\mathsf{S}} \longrightarrow x_j \in \mathcal{G}}{\Delta, \mathcal{G} \models i^{\mathsf{S}} \longrightarrow x_j} \qquad \text{(C-GDECL)}$$

$$\frac{d\varphi \rightarrowtriangle i^{\mathsf{S}} \in \mathcal{G} \qquad s\varphi = i^{\mathsf{S}}}{\Delta, \varphi, \mathcal{G} \models d \rightsquigarrow s} \qquad \text{(C-ASSOC)}$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}} \qquad \mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}} \qquad d\varphi = x_j^{\mathsf{D}}}{\Delta, \varphi, \mathcal{G}, \mathcal{R} \models x_i^{\mathsf{R}} \mapsto d} \qquad \text{(C-RESOLVE)}$$

$$\frac{\forall x \in [\![N\varphi]\!]_{\mathcal{G}} . \nu(x) = 1}{\Delta, \varphi, \mathcal{G} \models !N} \qquad \text{(C-DISTINCT)}$$

$$\frac{[\![N_1\varphi]\!]_{\mathcal{G}} \subseteq [\![N_2\varphi]\!]_{\mathcal{G}}}{\Delta, \varphi, \mathcal{G} \models N_1 \subsetsim N_2} \qquad \text{(C-SUBSET)}$$

$$\frac{t_1\varphi <_T t_2\varphi \qquad \forall t . (t_1\varphi <_T t \implies t_2\varphi \leq_T t)}{\Delta, \varphi, <_T \models t_1 <: t_2} \qquad \text{(C-SUPERTYPE)}$$

$$\frac{t_1\varphi \leq_T t_2\varphi}{\Delta, \varphi, <_T \models t_1 \leq: t_2} \qquad \text{(C-SUBTYPE)}$$

Nameset interpretation $\boxed{[\![N]\!] := X}$

$$[\![{}^{ns}\overline{\mathcal{D}}(i^{\mathsf{S}})]\!]_{\mathcal{G}} := \{ x \mid {}^{ns}x_j^{\mathsf{D}} \in \mathcal{D}_{\mathcal{G}}(i^{\mathsf{S}}) \} \qquad [\![{}^{ns}\overline{\mathcal{V}}(i^{\mathsf{S}})]\!]_{\mathcal{G}} := \{ x \mid \vdash_{\mathcal{G}} i^{\mathsf{S}} \mapsto {}^{ns}x_j^{\mathsf{D}} \}$$

$$[\![{}^{ns}\overline{\mathcal{R}}(i^{\mathsf{S}})]\!]_{\mathcal{G}} := \{ x \mid {}^{ns}x_j^{\mathsf{R}} \in \mathcal{R}_{\mathcal{G}}(j^{\mathsf{S}}) \} \qquad [\![{}^{ns}\overline{\mathcal{W}}(i^{\mathsf{S}})]\!]_{\mathcal{G}} := \{ x \mid \vdash_{\mathcal{G}} i^{\mathsf{S}} \rightarrowtail {}^{ns}x_j^{\mathsf{D}} \}$$

Figure 3.7: Semantics of the constraint language

3.2.1 Type Checking

An equality constraint $t_1 \stackrel{?}{=} t_2$ is satisfied by a context Δ with a substitution component φ, if application of φ makes t_1 and t_2 syntactically equal, i.e. if $t_1\varphi = t_2\varphi$ (rule C-EQUAL). The substitution is multi-sorted, so e.g. declaration variables δ are mapped to declarations, and type variables τ are mapped to types. Applying the substitution twice should have the same affect as applying it once. We define well-formedness of φ as follows:

DEFINITION 3.2 (Well-formedness of φ). The substitution φ is well-formed, written as $\mathrm{WF}(\varphi)$, if it is idempotent.

A declaration-type constraint $d : t$ is satisfied by a context Δ, if the type of d in the type-map ψ equals t (rule C-TYPEOF). Because variables can occur in d, we apply the substitution φ before doing the lookup $\psi(d\varphi)$. Similarly, we apply φ to t before doing the comparison.

3.2.2 Scope Graph

Scope graph constraints are satisfied, if the scope graph $\mathcal{G}$ from the constraint context contains an edge corresponding to the constraint (rules C-GDECL to C-GASSOC). Since the constraint notation corresponds one-to-one with the graphical notation for scope graphs, we represent $\mathcal{G}$ simply as a set of scope graph constraints. We allow variables only as the target of a direct edge, which is why we apply the substitution φ to s in rule C-GEDGE. Well-formedness of a scope graph is defined as follows (Neron et al., 2015):

DEFINITION 3.3 (Well-formedness of $\mathcal{G}$). A scope graph $\mathcal{G}$ is well-formed, written as $\mathrm{WF}(\mathcal{G})$, if each declaration x_i^{D} and reference x_i^{R} appear in exactly one scope.

In this form, the rule allows the scope graph to be bigger than the set of edges specified in the constraints. This is a problem, because a solver could 'invent' edges to satisfy name resolution constraints. Therefore, we also require that the scope graph is the smallest scope graph that models the scope graph constraints:

DEFINITION 3.4 (Minimality of $\mathcal{G}$). A scope graph $\mathcal{G}$ from a constraint context Δ, must be the smallest graph that models the scope graph constraints in C.

3.2.3 Name resolution

An associated scope constraint $d \rightsquigarrow s$ is satisfied, according to rule C-ASSOC, if the declaration d has an associated scope equal to s in the scope graph. Because d may not be ground, we apply the substitution φ to it, before looking for an associated scope edge in $\mathcal{G}$. Similarly, we apply φ to s before testing if it is equal to the found scope i^{S}.

Satisfaction of name resolution constraints is based on the decidable resolution judgment $\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$ (rule C-RESOLVE). A constraint $x_i^{\mathsf{R}} \longmapsto d$ is satisfied if the reference x_i^{R} resolved to a declaration x_j^{D}, and this declaration is equal to d. Because d can be a variable, we apply the substitution φ in the equality test.

The name resolution calculus allows ambiguous resolutions, so there could be multiple x_i^{D} for which the resolution judgment is true. We want a reference to resolve to the same declaration, even if multiple resolution constraints for the reference exist. The choice of resolution is fixed by the resolution component $\mathcal{R}$. Because $\mathcal{R}$ must be consistent with the scope graph, well-formedness of $\mathcal{R}$ is defined with respect to the scope graph from the same context as follows:

DEFINITION 3.5 (Well-formedness of $\mathcal{R}$). A name resolution $\mathcal{R}$ is well-formed with respect to a scope graph $\mathcal{G}$, if

$$\mathrm{WF}_{\mathcal{G}}(\mathcal{R}) \equiv \forall x_i^{\mathsf{R}}. \left(\mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}} \implies \vdash_G x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}} \right)$$

Satisfaction of name disambiguation constraint depends on the interpretation of name sets. Namesets are multi-sets of names, corresponding to sets of declarations or references. We write $\nu(x)$ for the multiplicity of x in a name set. The sets of declared and referred names in a scope s follow simply from the sets of declarations and references in s. The set of visible names contains the names of all the declarations that are visible in s, and is defined using the visibility relation $s \mapsto x_i^{\mathsf{D}}$ from the resolution calculus. The set of names reachable from s includes shadowed names as well. It is defined using the reachability relation $s \rightarrowtail x_i^{\mathsf{D}}$ from the resolution calculus.

A distinct names constraint $!N$ is satisfied if the names corresponding to N all have multiplicity $\nu(x) = 1$ (rule C-DISTINCT). Since N might contain variables, we apply the substitution φ to it before calculating the names. A constraint $N_1 \subsetneq N_2$ is satisfied if the names corresponding to N_1 are a subset of the names corresponding to N_2. Again, we must apply the substitution to N_1 and N_2.

3.2.4 Subtyping

In the nominal subtyping model of the constraint language, types are modeled as a forest. This means that every type has either zero or one supertypes. Well-formedness of the subtyping relation $<_T$ is – following the set-theoretic definition of a tree – defined as follows:

DEFINITION 3.6 (Well-formedness of $<_T$). The subtyping relation $<_T$ is well-formed if it is a partial order that models a forest, i.e.

$$\mathrm{WF}(<_T) \equiv \forall t. \left(\{t' \mid t <_T t'\} \text{ is totally ordered by } <_T \right)$$

Supertype constraints specify the subtyping relation. The subtyping relation $<_T$ satisfies a supertype constraint $t_1 <: t_2$, if t_1 is a strict subtype of t_2, and t_2 is the least supertype of t_1 (rule S-SUPERTYPE). Application of φ is necessary, since the types may contain variables. We use the non-strict subtyping relation $\leq_T$, which simply adds reflexivity to the strict subtyping relation $<_T$ from the constraint context.

Finally rule C-SUBTYPE states that a subtype constraint $t_1 \leq: t_2$ is satisfied if t_1 is a non-strict subtype of t_2. Again, the substitution needs to be applied, because the types may not be ground.

Subtype constraints could unexpectedly be satisfied, by making the subtyping relation larger than is necessary to satisfy the supertype constraints. To prevent this, we require the subtyping relation to be the smallest relation that satisfies the supertype constraints:

DEFINITION 3.7 (Minimality of $<_T$). The subtyping relation $<_T$ must be the smallest relation that satisfies the supertype constraints in C.

Now we have a formal description of the constraint language, and its meaning in terms of the satisfaction relation. In the next chapter we will look at an algorithm that calculates solutions that are sound with respect to this specification.

Chapter 4

Solver

In this chapter, we present an algorithm to solve a constraint problem for the constraint language defined in Chapter 3. The main idea of the solver is to rewrite a constraint and an empty constraint context to a trivial constraint and a context that satisfies the original constraint. In the first section we describe the general setup of the solver, followed by the rules for solving type checking, name resolution, and subtyping constraints. In the second section we will discuss formal properties of the algorithm. We will show that the algorithm terminates, and that it is sound with respect to the satisfaction relation. We will also show that the algorithm is not complete, but that this has not been a problem in practice.

4.1 Solver Algorithm

The solver solves a constraint C by constructing a constraint context Δ that models that constraint according to our semantics. The algorithm is based on a non-deterministic rewrite system on a pair $\langle C;\Delta\rangle$ of a constraint and a constraint context, which we call the solver state. The rules of our rewrite system have the following form:

$$\begin{array}{c}\langle C\,;\,\Delta\rangle \longrightarrow \mathcal{P}_{\geq 1}(\langle C'\,;\,\Delta'\rangle)\\ \text{if } \textit{conditions}\end{array}$$

The rules rewrite the state $\langle C;\Delta\rangle$ to a possibly changed state $\langle C';\Delta'\rangle$. Extra 'if' clauses make the rule conditional, and can introduce auxiliary definitions used in the resulting solver state. If we only need to make a few components of the context explicit, we will use the same Δ, X notation as in the previous chapter. Constraints are matched module associativity and commutativity of conjunction, and rules can be applied in any order. Note that a conjunction match can easily be satisfied for a single primitive constraint, by using $C = \mathsf{True}$. Rewrite rules can make non-deterministic choices by returning a positive, finite number of alternative states. However, since most rules will be deterministic in this sense, we will omit the set-delimiters and write $\langle C;\Delta\rangle$ instead of $\{\langle C;\Delta\rangle\}$, if a single state is returned.

```
function SOLVE(C)
    $\Delta \leftarrow \Delta_0$
    while $C \neq \mathsf{True} \wedge \exists \langle C;\Delta\rangle \longrightarrow R$ do
        branch on every $\langle C';\Delta'\rangle \in R$
        $\langle C;\Delta\rangle \leftarrow \langle C';\Delta'\rangle$
    end while
    $R \leftarrow$ join on $\langle C\,;\,\Delta\rangle$
    if $\exists! \langle C;\Delta'\rangle \in R. C = \mathsf{True}$ then
        return $\Delta'$                                   ▷ Satisfied
    else
        return $\bot$                                      ▷ Unsatisfied
    end if
end function
```

Figure 4.1: Constraint solving algorithm

The algorithm, presented in Figure 4.1, starts from an initial constraint C and a constraint context Δ_0. In the initial context Δ_0, all mappings are undefined on their whole domain and the scope graph is empty. The loop on line 3 takes one rewrite step each iteration, until $C = \mathsf{True}$ or there are no more possible rewrite steps. If multiple states are returned, we branch non-deterministically on all of them. The resulting states of all branches are collected in R by the *join* operation on line 7. If exactly one branch reduced the constraint to $C = \mathsf{True}$, we return the corresponding constraint context as our final result. Otherwise we fail, since there was no solution, or there were multiple ambiguous solutions.

Now we will explain the rules of our rewrite system. We will follow a similar structure as in Chapter 3, starting with an explanation of the general rules in this section, then discussing the rules for type checking, name resolution and subtyping in the next subsections. The trivial constraint True is satisfied by any context, and it can therefore just be eliminated without making any changes to the context (rule S-TRUE).

4.1.1 Type Checking

The rules for type checking constraints are presented in Figure 4.2. Equality constraints are solved by first-order unification. Rules S-TRIVIAL to S-DECOMPOSE are the same as the rules from the unification algorithm $\mathcal{U}$, presented in Figure 2.10. We have no explicit rules for error cases, such as in $\mathcal{U}$. Any error case will result in an unsolved constraint, causing the algorithm to return $\bot$. In rule S-ELIMINATE, the substitution is applied to the solver state, i.e. $\langle C;\Delta\rangle\sigma$. This means that the substitution is applied to the constraint C and all the components of Δ. Since we defined application of a substitution to another substitution as a composition, the resulting $\varphi\sigma$ is actually the composition of φ and σ.

$$\boxed{\langle C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle}$$

$$\langle \mathsf{True} \wedge C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle \qquad \text{(S-TRUE)}$$

$$\langle t \overset{?}{=} t \wedge C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle \qquad \text{(S-TRIVIAL)}$$

$$\langle t \overset{?}{=} \alpha \wedge C;\Delta\rangle \longrightarrow \langle \alpha \overset{?}{=} t \wedge C;\Delta\rangle \quad \text{if } t \notin \mathit{Var} \qquad \text{(S-ORIENT)}$$

$$\langle \alpha \overset{?}{=} t \wedge C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle\sigma \quad \text{if } \alpha \notin \text{vars}(t),\ \sigma := \{\alpha \mapsto t\} \qquad \text{(S-ELIMINATE)}$$

$$\langle f(t_1,\ldots,t_n) \overset{?}{=} f(t'_1,\ldots,t'_n) \wedge C;\Delta\rangle \longrightarrow \langle \left(\bigwedge_{i=1}^{n} t_i \overset{?}{=} t'_i\right) \wedge C;\Delta\rangle \qquad \text{(S-DECOMPOSE)}$$

$$\langle x_i^{\mathsf{D}} : t \wedge C;\Delta,\psi\rangle \longrightarrow \langle C;\Delta,\{(x_i^{\mathsf{D}},t)\} \cup \psi\rangle \quad \text{if } x_i^{\mathsf{D}} \notin \text{dom}(\psi) \qquad \text{(S-TYPEOF1)}$$

$$\langle x_i^{\mathsf{D}} : t \wedge C;\Delta,\psi\rangle \longrightarrow \langle t \overset{?}{=} t' \wedge C;\Delta,\psi\rangle \quad \text{if } \psi(x_i^{\mathsf{D}}) = t' \qquad \text{(S-TYPEOFN)}$$

Figure 4.2: Constraint solver rewrite rules: type checking

When solving type-of constraints, like $x_i^{\mathsf{D}} : t$, we can distinguish two cases. Either this is the first time we encounter a type for this declaration, or we have seen a constraint for it before. In the first case, we can simply add the type for this declaration to the type map ψ, such that $\psi(x_i^{\mathsf{D}}) = t$. In the second case, we need to make sure that the type t in the constraint is equal to the type $\psi(x_i^{\mathsf{D}})$ in the type map. Note that, although our semantics are defined in terms of a ground type map ψ, the types in ψ can contain variables in an intermediate solver state. Therefore, we do not check equality directly, but generate an equality constraint between the type from the constraint and the type from ψ.

4.1.2 Name Resolution

We will now explain the rules for scope graph construction, name resolution, and name disambiguation constraints, presented in Figure 4.3.

Scope graph construction Since we represent a scope graph as the set of edges, rules S-GDECL to S-GASSOC are straightforward. When a scope graph constraint is encountered, we add it to $\mathcal{G}$. The conditions on the rules prevent us from adding a declaration or reference to multiple scopes, importing a reference that is not in the graph, or associating multiple scopes with a sin-

Solver rules $\boxed{\langle C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle}$

$$\begin{array}{rll}
\langle i^{\mathsf{S}} \mathrel{-\!\!\blacktriangleright} x_j^{\mathsf{D}} \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\{i^{\mathsf{S}} \mathrel{-\!\!\blacktriangleright} x_j^{\mathsf{D}}\} \cup \mathcal{G}\rangle & \\
& \text{if } \nexists s. s \mathrel{-\!\!\blacktriangleright} x_j^{\mathsf{D}} \in \mathcal{G} & \text{(S-GD}\textsc{ecl}\text{)} \\
\langle x_i^{\mathsf{R}} \mathrel{-\!\!\blacktriangleright} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\{x_i^{\mathsf{R}} \mathrel{-\!\!\blacktriangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle & \\
& \text{if } \nexists s. x_i^{\mathsf{R}} \mathrel{-\!\!\blacktriangleright} s \in \mathcal{G} & \text{(S-GR}\textsc{ef}\text{)} \\
\langle i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\blacktriangleright}} s \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\{i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\blacktriangleright}} s\} \cup \mathcal{G}\rangle & \text{(S-GE}\textsc{dge}\text{)} \\
\langle i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}} \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\{i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}}\} \cup \mathcal{G}\rangle & \\
& \text{if } \exists s. x_j^{\mathsf{R}} \mathrel{-\!\!\blacktriangleright} s \in \mathcal{G} & \text{(S-GNE}\textsc{dge}\text{)} \\
\langle x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle & \\
& \text{if } \nexists s. x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} s \in \mathcal{G} & \text{(S-GA}\textsc{ssoc}\text{)} \\
\langle x_i^{\mathsf{D}} \rightsquigarrow s \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle s \overset{?}{=} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\rangle & \\
& \text{if } x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \in \mathcal{G} & \text{(S-A}\textsc{ssoc}\text{)} \\
\langle x_i^{\mathsf{R}} \mapsto d \wedge C;\Delta,\mathcal{G},\mathcal{R}\rangle \longrightarrow & \left\{\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C;\Delta,\mathcal{G},\{(x_i^{\mathsf{R}},x_j^{\mathsf{D}})\} \cup \mathcal{R}\rangle \;\middle|\; x_j^{\mathsf{D}} \in D\right\} & \\
& \text{if } C \cap C^{\mathcal{G}} = \emptyset,\ x_i^{\mathsf{R}} \notin \mathrm{dom}(\mathcal{R}), & \\
& \quad \exists D.\, \textsc{Res}_{\mathcal{G}}(x_i^{\mathsf{R}}) = D,\ |D| > 0 & \text{(S-R}\textsc{esolve}\text{1)} \\
\langle x_i^{\mathsf{R}} \mapsto d \wedge C;\Delta,\mathcal{G},\mathcal{R}\rangle \longrightarrow & \langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C;\Delta,\mathcal{G},\mathcal{R}\rangle & \\
& \text{if } \mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}} & \text{(S-R}\textsc{esolve}\text{N)} \\
\langle !N \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\mathcal{G}\rangle & \\
& \text{if } C \cap C^{\mathcal{G}} = \emptyset,\ \mathrm{vars}(N) = \emptyset, & \\
& \quad \exists X.\, \textsc{Ns}_{\mathcal{G}}(N) = X, & \text{(S-D}\textsc{istinct}\text{)} \\
& \quad \forall x \in X.\, \nu(x) = 1 & \\
\langle N_1 \subsetneqq N_2 \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow & \langle C;\Delta,\mathcal{G}\rangle & \\
& \text{if } C \cap C^{\mathcal{G}} = \emptyset, & \\
& \quad \mathrm{vars}(N_1) \cup \mathrm{vars}(N_2) = \emptyset, & \text{(S-S}\textsc{ubset}\text{)} \\
& \quad \exists X_1.\, \textsc{Ns}_{\mathcal{G}}(N_1) = X_1, & \\
& \quad \exists X_2.\, \textsc{Ns}_{\mathcal{G}}(N_2) = X_2,\ X_1 \subseteq X_2 &
\end{array}$$

Nameset calculation

$$\begin{aligned}
\textsc{Ns}_{\mathcal{G}}({}^{ns}\overline{\mathcal{D}}(i^{\mathsf{S}})) &:= \{x \mid x_j^{\mathsf{D}} \in \mathcal{D}(i^{\mathsf{S}})\} \\
\textsc{Ns}_{\mathcal{G}}({}^{ns}\overline{\mathcal{R}}(i^{\mathsf{S}})) &:= \{x \mid x_j^{\mathsf{R}} \in \mathcal{R}(i^{\mathsf{S}})\} \\
\textsc{Ns}_{\mathcal{G}}({}^{ns}\overline{\mathcal{V}}(i^{\mathsf{S}})) &:= \{x \mid x_j^{\mathsf{D}} \in D \wedge \textsc{Env}_{\mathcal{E}}[\emptyset,\emptyset](i^{\mathsf{S}}) = (\top,D)\} \\
\textsc{Ns}_{\mathcal{G}}({}^{ns}\overline{\mathcal{W}}(i^{\mathsf{S}})) &:= \{x \mid x_j^{\mathsf{D}} \in D \wedge \textsc{Env}_{\mathcal{E}}[\emptyset,\emptyset](i^{\mathsf{S}}) = (\top,D)\}
\end{aligned}$$

(using the empty label order $<_{\emptyset}$)

Figure 4.3: Constraint solver rewrite rules: name resolution

gle declaration. Note that the only rule that allows a variable to appear in the scope graph, is rule S-GEDGE. This is of course, because the resolution algorithm only admits graphs where variables appear as the target of a direct edge.

Name resolution An associated scope constraint is solved using a simple lookup in $\mathcal{G}$. To be able to carry out the lookup, the declaration has to be ground. If s is a constraint variable, we want it to be unified with the discovered scope j^{S}. Therefore, we generate an equality constraint instead of directly testing for equality.

Name resolution constraints are solved using the name resolution algorithm presented in Figure 2.6. Similar to the type-of constraints, we have cases for the first time we encounter a reference, and for subsequent times. When we resolve a reference, we add the resolved declaration to $\mathcal{R}$. If the reference is not in $\mathcal{R}$, we need to resolve it (rule S-RESOLVE1). We need to make sure that we do not invoke the resolution algorithm if new edges can still be added to the scope graph. A new edge could easily invalidate a resolution, e.g. by introducing a declaration with a shorter path. The condition $C \cap C^{\mathcal{G}} = \emptyset$ requires that there are no scope graph constraints left in C, and therefore no new edges could be added after solving the name resolution constraint. Although all the edges are in $\mathcal{G}$, it may still contain variables, resulting in an unknown resolution $\bot$. We write $\exists D.\, \textsc{Res}_{\mathcal{G}}(x_i^{\mathsf{D}})$, to ensure D is a set of declarations, and not $\bot$. We require that the reference resolves to at least one declaration, but it can resolve to multiple. This is where we introduce a non-deterministic choice between the possible resolutions. Every solver state we return picks one of the declarations, and fixes this choice in $\mathcal{R}$. We also introduce an equality constraint between d and the declaration, to allow unification if d is a variable. If the resolution $\mathcal{R}$ already contains a resolved declaration for the reference, we simply generate the equality constraint without invoking the name resolution algorithm (rule S-RESOLVEN).

Name disambiguation Solving name disambiguation constraints involves calculating name sets. The rules for calculating name sets are presented in Figure 4.3. The sets of declared and referred names in a scope are easily calculated from the declarations and reference of the scope. For the set of visible names, we use the ENV function, which calculates the set of visible declarations in a scope. To get the correct set, we need to invoke it with the same parameters as a top-level RES does, i.e. with seen imports $\mathbb{I} = \emptyset$ and seen scopes $\mathbb{S} = \emptyset$, and the regular expression $\mathcal{E}$. There is no separate function that calculates the set of reachable declarations. However, the difference between reachable and visible declarations is, that the latter are restricted to the declarations with the shortest path. By using an empty path order, where no step, not even the builtin $\mathbf{D}(\cdot)$, is smaller than any other step, we can use the same environment calculation. Because no path will be shorter than any other path, the reachable and visible sets will be the same. To solve distinct names constraints, we just check that every name in the names calculated for N appears

$$\boxed{\langle C;\Delta\rangle \longrightarrow \langle C;\Delta\rangle}$$

$$\begin{aligned}
&\langle t_1 <: t_2 \wedge C;\Delta,<_T\rangle \longrightarrow \langle C;\Delta,<'_T\rangle \\
&\qquad\text{if } \mathrm{vars}(t_1)\cup\mathrm{vars}(t_2)=\emptyset, \\
&\qquad\quad t_1 \notin \mathrm{dom}(<_T),\ t_2 \not\leq_T t_1, \\
&\qquad\quad <'_T = \{(t,t') \mid t \leq_T t_1, t_2 \leq_T t'\} \cup <_T \qquad (\text{S-SUPERTYPE})
\end{aligned}$$

$$\begin{aligned}
&\langle t_1 \leq: t_2 \wedge C;\Delta,<_T\rangle \longrightarrow \langle C;\Delta,<_T\rangle \\
&\qquad\text{if } \mathrm{vars}(t_1)\cup\mathrm{vars}(t_2)=\emptyset,\ t_1 \leq_T t_2 \qquad (\text{S-SUBTYPE})
\end{aligned}$$

Figure 4.4: Constraint solver rewrite rules: subtyping

only once, i.e. the multiplicity $\nu(x) = 1$ (rule S-DISTINCT). Equally, a subset constraint is solved if the names calculated for N_1 are a subset of the names calculated for N_2 (rule S-SUBSET). For name set calculation we have the same requirements as for name resolution, i.e. that all the graph edges are present before the algorithm is invoked.

4.1.3 Subtyping

Supertype constraints $t_1 <: t_2$ are solved by building the subtyping relation $<_T$. We have seen in the semantics that the relation must be a forest, and that t_2 must be the smallest supertype of t_1. The condition $t_2 \not\leq_T t_1$ of rule S-SUPERTYPE prevents us from creating a cycle. The condition $t_1 \notin \mathrm{dom}(\leq_T)$ ensures that we only add one supertype for t_1. Together they make sure that the relation models a tree, and that t_2 will be the smallest supertype. We also need to ensure that the subtyping relation $<_T$ is transitive. Just adding $t_1 <_T t_2$ is not enough, because t_1 might have subtypes and t_2 might have supertypes. Therefore, we add a relation between every $t \leq_T t_1$ and every $t_2 \leq_T t'$. Subtype constraints $t_1 \leq: t_2$ are solved, if t_1 is a subtype of t_2 in the subtyping relation (rule S-SUBTYPE).

4.2 Formal Properties

In this section we discuss several formal properties of the algorithm we introduced. We start with soundness of the solver with respect to the semantics, then we show termination, and finally we discuss several examples of incompleteness.

4.2.1 Soundness

The algorithm should only give an answer if the answer is indeed a solution, that is, if it satisfies the constraint. Therefore, we define soundness as follows:

THEOREM 4.1 (Soundness of SOLVE). *The algorithm is sound, if the result of* SOLVE *is well-formed and satisfies the constraint according to the semantics, i.e.*

$$\forall C\Delta.\,(\textsc{Solve}(C) = \Delta \implies \mathrm{WF}(\Delta) \wedge \Delta \models C)$$

Before we can prove soundness of the algorithm, we need to introduce some auxiliary definitions and lemmas. First we define full instantiation of the solver state as follows:

DEFINITION 4.2 (Full Instantiation of $\langle C;\Delta\rangle$). We say a constraint C and context Δ are fully instantiated, i.e. $\langle C;\Delta,\varphi\rangle\downarrow$, if $\langle C;\Delta,\varphi\rangle\varphi = \langle C;\Delta,\varphi\rangle$.

This gives us two useful properties. First, the substitution in the constraint context is idempotent, i.e. applying it multiple times will have the same effect as applying it once. Second, any variable that occurs in the domain of the substitution, does not occur in C.

COROLLARY 4.3. *If $\langle C;\Delta,\varphi\rangle\downarrow$, then $\varphi\varphi = \varphi$, therefore φ is idempotent.*

COROLLARY 4.4. *If $\langle C;\Delta,\varphi\rangle\downarrow$, then* $\mathrm{vars}(C) \cap \mathrm{dom}(\varphi) = \emptyset$.

Let $C \cap C^{\mathcal{G}}$ denote the set of subterms of C from the sort $C^{\mathcal{G}}$. We define well-formedness of the solver state, consisting of several conditions. First, the constraint context must be well-formed, as defined in Section 3.2. Second, the state must be fully instantiated. Last, if the constraint C contains scope graph construction constraints, no names must be resolved yet, because the edges resulting from the remaining constraints, could invalidate those resolutions.

DEFINITION 4.5 (Well-formedness of $\langle C;\Delta\rangle$). We say a solver state consisting of a constraint C and a context Δ is well-formed, if

$$\begin{aligned}\mathrm{WF}(\langle C;\Delta,\mathcal{G},<_T\rangle) \equiv\, & \mathrm{WF}(\Delta,\mathcal{G},<_T) \wedge \langle C;\Delta,\mathcal{G},<_T\rangle\downarrow \\ & \wedge \left(C \cap C^{\mathcal{G}} \neq \emptyset \implies \mathrm{dom}(\mathcal{R}) = \emptyset\right)\end{aligned}$$

Now we define a preorder $\subseteq$ on constraint contexts, which is used to ensure that the solution grows monotonically during solving. Let $X|_D$ be the mapping X restricted to the domain D. Our preorder is defined as follows:

DEFINITION 4.6 (Context preorder $\subseteq$). For contexts Δ, Δ', we define a preorder $\Delta \subseteq \Delta'$ as

$$\begin{aligned}\langle \varphi,\psi,\mathcal{G},\mathcal{R},<_T\rangle \subseteq \langle \varphi',\psi',\mathcal{G}',\mathcal{R}',<'_T\rangle \equiv \exists\sigma.\,(\ & \varphi\sigma = \varphi' \\ & \wedge \psi\sigma = \psi'|_{\mathrm{dom}(\psi)} \\ & \wedge \mathcal{G}\sigma \subseteq \mathcal{G}' \\ & \wedge \mathcal{R} = \mathcal{R}'|_{\mathrm{dom}(\mathcal{R})} \\ & \wedge <_T = <'_T|_{\mathrm{dom}(<_T)})\end{aligned}$$

Sometimes we want to make the substitution σ explicit, in which case we write $\Delta \subseteq_\sigma \Delta'$ instead.

Using these definitions, we can now state that for every reduction step, three properties should hold. The first states that well-formedness of the solver state is preserved. The second states that the context before the reduction is included in the context after reduction. The last one states that if there is a reduction that ends with $C = \mathsf{True}$, and its context models the output constraint, it models the input constraint as well.

LEMMA 4.7 (Invariant of $\longrightarrow$). *The following invariant holds for the reduction:*

$$\begin{aligned}
\forall C\Delta C'\Delta'.\Big(\langle C;\Delta\rangle \longrightarrow \langle C';\Delta'\rangle \implies \\
&\big(\mathrm{WF}(\langle C;\Delta\rangle) \implies \mathrm{WF}(\langle C';\Delta'\rangle)\big) && \text{(WF)}\\
&\wedge\, \Delta \subseteq \Delta' && \text{(Sub)}\\
&\wedge\, \forall \Delta''.\,\big(\langle C';\Delta'\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta''\rangle \wedge \mathrm{WF}(\langle C';\Delta'\rangle) && \text{(Star)}\\
&\qquad\qquad \wedge\, \Delta'' \models C' \implies \Delta'' \models C\big)\Big)
\end{aligned}$$

Transitivity of implication and $\subseteq$ give us the following useful properties for multi-step reductions, which we will use in the proofs of (Star):

COROLLARY 4.8. *For any multi-step reduction, Transitivity of implication, and* (WF) *give us*

$$\forall C\Delta C'\Delta'.\,\big(\langle C;\Delta\rangle \longrightarrow^{*} \langle C';\Delta'\rangle \wedge \mathrm{WF}(\langle C;\Delta\rangle) \implies \mathrm{WF}(\langle C';\Delta'\rangle)\big)$$

COROLLARY 4.9. *For any multi-step reduction, transitivity of $\subseteq$, and* (Sub) *give us*

$$\forall C\Delta C'\Delta'.\,\big(\langle C;\Delta\rangle \longrightarrow^{*} \langle C';\Delta'\rangle \implies \Delta \subseteq \Delta'\big)$$

The proof of Lemma 4.7 by case analysis can be found in Appendix A.1. Now we can state and prove soundness for multi-step reductions.

LEMMA 4.10 (Soundness of $\longrightarrow$). *The reduction is sound, i.e.*

$$\forall nC\Delta\Delta''.\,\big(\langle C;\Delta\rangle \longrightarrow^{n} \langle \mathsf{True};\Delta''\rangle \wedge \mathrm{WF}(\langle C;\Delta\rangle) \implies \Delta'' \models C\big)$$

Proof. By induction on the length n of the reduction.

$\langle 1\rangle$1. ASSUME: 1. $\langle C;\Delta\rangle \longrightarrow^{0} \langle \mathsf{True};\Delta''\rangle$
2. $\mathrm{WF}(\langle C;\Delta\rangle)$
PROVE: $\Delta'' \models C$
PROOF: Because of the 0-step reduction, we know that $C = \mathsf{True} \wedge \Delta = \Delta''$. Therefore $\Delta'' \models C$ follows from rule C-TRUE.

$\langle 1\rangle$2. ASSUME: 1. $\langle C;\Delta\rangle \longrightarrow^{n+1} \langle \mathsf{True};\Delta''\rangle$
2. $\mathrm{WF}(\langle C;\Delta\rangle)$
IH. $\forall C'\Delta'\Delta''.\,(\langle C';\Delta'\rangle \longrightarrow^{n} \langle \mathsf{True};\Delta''\rangle \wedge \mathrm{WF}(\langle C';\Delta'\rangle) \implies \Delta'' \models C')$
PROVE: $\Delta'' \models C$
$\langle 2\rangle$1. $\exists C'\Delta'.\,\big([\langle C;\Delta\rangle \longrightarrow \langle C';\Delta'\rangle]^{(1)} \wedge [\langle C';\Delta'\rangle \longrightarrow^{n} \langle \mathsf{True};\Delta''\rangle]^{(2)}\big)$
PROOF: By assump. $\langle 1\rangle$2-1 and property of reductions.
$\langle 2\rangle$2. $\mathrm{WF}(\langle C';\Delta'\rangle)$

PROOF: By assump. $\langle 1\rangle$2-2 and (WF) of $\langle 2\rangle$1-1.

$\langle 2\rangle$3. $\Delta'' \models C'$

PROOF: By assump. $\langle 2\rangle$1-2, $\langle 2\rangle$2, and IH.

$\langle 2\rangle$4. Q.E.D.

PROOF: By $\langle 2\rangle$2, assump. $\langle 2\rangle$1-2, $\langle 2\rangle$3, and (Star) of assump. $\langle 2\rangle$1-1.

$\langle 1\rangle$3. Q.E.D.

PROOF: By $\langle 1\rangle$1, $\langle 1\rangle$2, and the principle of induction. □

Given all the above, we can now prove soundness of the solver.

THEOREM 4.1 (Soundness of SOLVE). *The algorithm is sound, if the result of* SOLVE *is well-formed and satisfies the constraint according to the semantics, i.e.*

$$\forall C\Delta.\,(\textsc{Solve}(C) = \Delta \implies \mathrm{WF}(\Delta) \wedge \Delta \models C)$$

Proof. For the initial empty context Δ_0 and any constraint C, it holds that $\mathrm{WF}(\langle C;\Delta_0\rangle)$. The algorithm returns Δ'' only if $\langle C;\Delta_0\rangle \longrightarrow \langle \mathsf{True};\Delta''\rangle$. We conclude by application of Lemma 4.10. □

We have not yet developed a formal proof for the minimality criteria on the scope graph and the subtyping relation. We believe our algorithm does indeed produce minimal solutions, but a proof is subject of future work.

4.2.2 Termination

We say that the solver is terminating, if it halts for every input constraint C.

THEOREM 4.11 (Termination of SOLVE). $\forall C.\,(\textsc{Solve}(C)$ *terminates*$)$.

Before we can prove this, we will prove termination for multi-step reductions.

LEMMA 4.12 (Termination of $\longrightarrow$). $\forall C\Delta.\,(\langle C;\Delta\rangle \longrightarrow^* \ldots$ *terminates*$)$.

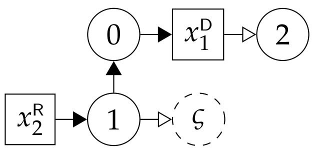
The proof by case analysis can be found in Appendix A.2. Using this we can prove termination of the solver.

THEOREM 4.11 (Termination of SOLVE). $\forall C.\,(\textsc{Solve}(C)$ *terminates*$)$.

Proof. From Lemma 4.12 we know that every multi-step reduction terminates. This means our algorithm terminates, if only a finite number of branches are created. The only rule that creates branches is rule S-RESOLVE1. The scope graph is finite, so every reference resolves to a finite number of declarations. Therefore, every time rule S-RESOLVE1 is applied, a finite number of branches is created. Given that there are also finite references in the scope graph, and the rule is only applied once for every reference, the total number of branches will be finite. Hence, the algorithm terminates. □

$$x_2^{\mathsf{R}} \mapsto \delta$$
$$\delta \rightsquigarrow \varsigma$$

(a) Constraints



(b) Scope graph

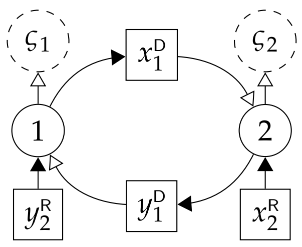
$$\delta \mapsto x_1^{\mathsf{D}}$$
$$\varsigma \mapsto 2^{\mathsf{S}}$$

(c) Possible solution

Figure 4.5: Example of incompleteness

$$x_2^{\mathsf{R}} \mapsto \delta_1$$
$$\delta_1 \rightsquigarrow \varsigma_1$$
$$y_2^{\mathsf{R}} \mapsto \delta_2$$
$$\delta_2 \rightsquigarrow \varsigma_2$$

(a) Constraints



(b) Scope graph

$$\delta_1 \mapsto x_1^{\mathsf{D}}$$
$$\delta_2 \mapsto y_1^{\mathsf{D}}$$
$$\varsigma_1 \mapsto 2^{\mathsf{S}}$$
$$\varsigma_2 \mapsto 1^{\mathsf{S}}$$

(c) Possible solution

Figure 4.6: Example of incompleteness

4.2.3 On Completeness

Completeness states that, if a solution exists for a constraint C, the algorithm will find it:

DEFINITION 4.13 (Completeness of SOLVE). Algorithm SOLVE is complete, if the algorithm finds a solution for every C that is satisfiable:

$$\forall C\Delta.\,(\Delta \models C \implies (\exists \Delta.\,\textsc{Solve}(C) = \Delta))$$

Unfortunately our algorithm is not complete. As it is now we see two major reasons for this incompleteness, namely incomplete scope graphs, and lack of subtype inference. We will give some examples for the name resolution case. The first example is presented in Figure 4.5. The solver needs to resolve x_2^{R} to the declaration x_1^{D} in the parent scope, to be able to find the value for ς and solve the constraints. Because of the import ς, which could shadow the declaration in the parent, the resolution fails, and the constraints remain unsolved.

In our second example, presented in Figure 4.6, there is a mutual dependency between the references. They can only be resolved through the variable scope import, which can only be instantiated if the references are resolved. This example is arguably less realistic, but also harder to solve than the first one. In the first example there existed a resolution path, even though it was disregarded because it *might* be unsafe. In the second example, there is no path at all, and the algorithm would need to resort to guessing variables to solve it.

Despite the fact that these incomplete cases exist, we have not run into them in practice. We expect that for many practical programming languages, this incompleteness will not be a problem. Testing this hypothesis is the subject of future work.

Chapter 5

Evaluation

We evaluated our approach with respect to expressiveness and realizability. To evaluate expressiveness, we defined the static semantics of two languages from different paradigms that are well-known in the literature. The first language is PCF, a functional language with integer base type. The second language is Featherweight Java, an object-oriented language with inheritance. To evaluate realizability, we implemented the solver, and static analysis for the languages LMR, PCF and Featherweight Java in the Spoofax workbench. In the first section we will introduce the syntax and important features of PCF, and explain how we expressed its static semantics using our constraint language. In the second section we do the same for Featherweight Java. In the last section we discuss how we implemented the constraint solver in Spoofax, and where this implementation differs from the theoretical presentation.

5.1 Static Semantics of PCF

PCF is a small functional language with integer base types, first introduced by Plotkin (1977). We first describe the syntax of the language, and then discuss the constraint function for static analysis.

$$\begin{array}{rcl} \textit{expr} & := & \textit{id} \mid n \mid E \oplus E \mid \textit{expr}\ \textit{expr} \\ & \mid & \texttt{fun}\ \textit{id}\ \texttt{->}\ \textit{expr} \mid \texttt{fix}\ \textit{id}\ \texttt{->}\ \textit{expr} \\ & \mid & \texttt{ifz}\ \textit{expr}\ \texttt{then}\ \textit{expr}\ \texttt{else}\ \textit{expr} \\ & \mid & \texttt{let}\ \textit{id}\ \texttt{=}\ \textit{expr}\ \texttt{in}\ \textit{expr} \end{array}$$

Numerical operators are drawn from the set $\oplus \in \{+,-,*,/\}$.

$$\textit{Type} \quad := \quad \texttt{Nat} \mid \textit{Type} \rightarrow \textit{Type}$$

Figure 5.1: Syntax of PCF

$$
\begin{aligned}
[\![\texttt{fun}\ x_i\ \texttt{->}\ e : t]\!]_s &:= s' \xrightarrow{\mathsf{P}} s \wedge s' \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau_1 \wedge [\![e : \tau_2]\!]_{s'} \\
&\quad \wedge t \stackrel{?}{=} \tau_1 \to \tau_2 \\
[\![\texttt{fix}\ x_i\ \texttt{->}\ e : t]\!]_s &:= s' \xrightarrow{\mathsf{P}} s \wedge s' \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau_1 \to \tau_2 \\
&\quad \wedge [\![e : \tau_1 \to \tau_2]\!]_s \wedge t \stackrel{?}{=} (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2) \\
[\![\texttt{let}\ x_i\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t]\!]_s &:= s' \xrightarrow{\mathsf{P}} s \wedge s' \longrightarrow x_i^{\mathsf{D}} \wedge x_i^{\mathsf{D}} : \tau \wedge [\![e_1 : \tau]\!]_s \wedge [\![e_2 : t]\!]_{s'} \\
[\![e_1\ e_2 : t]\!]_s &:= [\![e_1 : \tau \to t]\!]_s \wedge [\![e_2 : \tau]\!]_s \\
[\![\texttt{x} : t]\!]_s &:= x_i^{\mathsf{R}} \longrightarrow s \wedge x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : t \\
[\![n : t]\!]_s &:= t \stackrel{?}{=} \texttt{Nat} \\
[\![e_1 \otimes e_2 : t]\!]_s &:= t \stackrel{?}{=} \texttt{Nat} \wedge [\![e_1 : \texttt{Nat}]\!]_s \wedge [\![e_2 : \texttt{Nat}]\!]_s \\
[\![\texttt{ifz}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t]\!]_s &:= [\![e_1 : \texttt{Nat}]\!]_s \wedge [\![e_2 : t]\!]_s \wedge [\![e_3 : t]\!]_s
\end{aligned}
$$

Any s, δ or τ that appear free in the constraints, are assumed to be fresh. Numerical operators are drawn from the set $\oplus \in \{+, -, *, /\}$.

Figure 5.2: Constraint generation function for PCF

5.1.1 The Language

The syntax of LMR is presented in Figure 5.1. The language features the following constructs:

- The language features *lexical scoping* of identifiers. New declarations are introduced by **`fun`**, **`fix`**, and **`let`** expressions. Declarations in nested scopes shadow declarations with the same name from outer scopes.
- *Abstractions* are created with the **`fun`** and **`fix`** expressions. The fixpoint expression enables recursion, and expects its argument to be a function. Function application is written as juxtaposition of expressions, as in $e_1\ e_2$.
- The language provides *integer* literals, as well as arithmetic operations addition, subtraction, multiplication, and division. A conditional **`ifz`** branches on an integer argument, evaluating the **`then`** branch if the integer is zero, and the **`else`** branch otherwise. The type of the branches is free, as long as it is the same for both.

5.1.2 Constraints

We specify the static semantics of PCF using the constraint generation function presented in Figure 5.2. The constraints generated for LMR programs are similar to the constraints we have seen for LMR expressions:

- The scopes in the scope graph for LMR programs will form a tree, connected by parent edges, that models the lexical scoping. The constraint

$$
\begin{array}{rcl}
\mathit{program} & := & \mathit{class}^* \\
\mathit{class} & := & \texttt{\textbf{class}}\ \mathit{id}\ \texttt{\textbf{extends}}\ \mathit{id}\ \texttt{\{}\ \mathit{fdecl}^*\ \mathit{init}\ \mathit{mdecl}^*\ \texttt{\}} \\
\mathit{fdecl} & := & \mathit{id}\ \mathit{id}\texttt{;} \\
\mathit{init} & := & \mathit{id}\texttt{(}\mathit{arg}^*\texttt{)}\ \texttt{\{ super(}\mathit{id}^*\texttt{);}\ \mathit{finit}^*\ \texttt{\}} \\
\mathit{finit} & := & \texttt{this.}\mathit{id}\ \texttt{=}\ \mathit{id}\texttt{;} \\
\mathit{mdecl} & := & \mathit{id}\ \mathit{id}\texttt{(}\mathit{arg}^*\texttt{)}\ \texttt{\{}\ \texttt{\textbf{return}}\ \mathit{expr}\texttt{;}\ \texttt{\}} \\
\mathit{arg} & := & \mathit{id}\ \mathit{id} \\
\mathit{expr} & := & \mathit{id} \mid \mathit{expr}\texttt{.}\mathit{id} \mid \mathit{expr}\texttt{.}\mathit{id}\texttt{(}\mathit{expr}^*\texttt{)} \mid \texttt{\textbf{new}}\ \mathit{id}\texttt{(}\mathit{expr}^*\texttt{)} \mid \texttt{(}\mathit{id}\texttt{)}\mathit{expr} \\
 & & \\
\mathit{Type} & := & \texttt{C}(\mathit{decl}) \mid \texttt{I}[\mathit{Type}^*] \mid \texttt{M}[\mathit{Type}^*,\mathit{Type}]
\end{array}
$$

Figure 5.3: Syntax of Featherweight Java

generation function passes down the current lexical scope s. The rules for **`fun`**, **`fix`**, and **`let`** all introduce a new subscope s' of scope s, and add a declaration in the new scope s'. References are resolved in the current lexical scope.

- The types of PCF are the base type `Int` of integers, and the function type $t_1 \to t_2$, where t_1 is the argument type, and t_2 the result type. The type equality constraints that are generated are the same as the constraints we have seen for the expressions in LMR.

5.2 Static Semantics of Featherweight Java

Featherweight Java is a small object-oriented language, introduced by Igarashi et al. (2001). The language is a lightweight version of Java, aiming to be a "minimal core calculus for modeling Java's type system." It features classes with fields, methods, and class inheritance. We first introduce the syntax of the language, and explain the different language constructs. Then we discuss how we model its static semantics using our constraint language.

5.2.1 The Language

The syntax of Featherweight Java is presented in Figure 5.3. The language features the following constructs:

- A Featherweight Java program consists of zero or more `class` definitions. Classes form a nominal *class hierarchy*, and declaring the superclass with `extends` is required. We assume the existence of an `Object` class as the root of the type hierarchy. Cyclic inheritance is not allowed.

- A class defines *fields*, which are explicitly typed by prefixing it with a class name. Field names must be distinct, and cannot shadow fields from the superclass.
- Every class defines exactly one *constructor*, which has the same name as the class. The constructor takes a number of explicitly typed arguments. The arguments are passed to the constructor of the super class in a `super` call, and used to initialize the class fields. The constructor must initialize every class field exactly once.

  In the original presentation of Featherweight Java, the constructor arguments had to match the names and order of the superclass constructor and the fields exactly. As we will explain later, we drop this restriction in our specification of the semantics.
- A class defines *methods*, which have explicitly typed parameters, and an explicit return type. The body of the method is one expression, which must be a subtype of the explicit return type. Method names must be distinct, and cannot shadow methods from the superclass. Inside a method body, there is an implicit variable `this` available, that refers to the object the method is invoked on.
- *New objects* are created using the `new` expression. The number of expressions passed to `new` must match the number of constructor parameters of the class. The expression types must be subtypes of the specified parameter types.
- *Access to methods and fields* is provided by dot-notation on an object. In case of method invocation, the number of passed expressions must match the number of method parameters, and the expression types must be subtypes of the defined parameter types.
- The type of an expression can be forced by using a *cast*, written as $(x)e$, where x is the class name one casts to.

5.2.2 Constraints

The constraint generation function for Featherweight Java is presented in Figures 5.4 and 5.5. We will explain how we model name binding and type checking for class definitions, fields, constructors, methods, and expressions.

Class definitions A definition of a class x introduces a new declaration ${}^{\mathsf{C}}x_i^{\mathsf{D}}$ in the global scope. Declarations and references for classes use the namespace C. For every class we introduce two new scopes. The first is the class scope s_c, associated with the class declaration. The class scope will contain the declarations of the constructor, and the fields and methods of the class. Overloading or overriding is forbidden in Featherweight Java, therefore we require all reachable definitions in the class scope to be distinct. The other scope is the lexical scope s' of the class body, which will contain the declaration for `this`.

$$
\begin{array}{rcl}
\llbracket C \rrbracket^{program} & := & \llbracket C \rrbracket^{class^*}_{s} \\[1ex]
\left\llbracket \begin{array}{l} \texttt{class } x_i \texttt{ extends } y_j \texttt{ \{} \\ \quad F \;\; I \;\; M \\ \texttt{\}} \end{array} \right\rrbracket^{class}_{s} & := & s \longrightarrow {}^{C}x_i \wedge {}^{C}x_i \rightarrowtriangle s_c \wedge {}^{C}y_j \longrightarrow s \\
 & & \wedge\, s_c \xrightarrow{\mathbf{S}}\!\!\triangleright {}^{C}y_j \wedge {}^{C}y_j^{\mathsf{R}} \mapsto \delta \wedge !\overline{\mathcal{W}}(s_c) \\
 & & \wedge\, \texttt{C}({}^{C}y_i^{\mathsf{D}}) <: \texttt{C}(\delta) \wedge s' \xrightarrow{\mathbf{P}} s \\
 & & \wedge\, s' \longrightarrow {}^{V}\mathit{this} \wedge {}^{V}\mathit{this}^{\mathsf{D}} : \texttt{C}({}^{C}x_i^{\mathsf{D}}) \\
 & & \wedge\, \llbracket F \rrbracket^{fdecl^*}_{s',s_c} \wedge \llbracket I \rrbracket^{init}_{s,s_c,x,y} \wedge \llbracket M \rrbracket^{mdecl^*}_{s',s_c} \\[1ex]
\llbracket x_i\ y_j \rrbracket^{fdecl}_{s,s_c} & := & s_c \longrightarrow {}^{V}y_j \wedge \llbracket x_i \rrbracket_{s,\tau} \wedge {}^{V}y_j^{\mathsf{D}} : \tau \\[1ex]
\llbracket x_i\texttt{(}A\texttt{) \{ super(}E\texttt{); } F \texttt{ \}} \rrbracket^{init}_{s,s_c,y,z} & := & x \stackrel{?}{=} y \wedge s_c \longrightarrow {}^{I}x_i^{\mathsf{D}} \wedge {}^{I}z^{\mathsf{R}} \longrightarrow s_c \\
 & & \wedge\, s' \xrightarrow{\mathbf{P}} s \wedge \llbracket A \rrbracket^{arg^*}_{s',\tau_1^*} \wedge {}^{I}x_i^{\mathsf{D}} : \texttt{I}[\tau_1^*] \\
 & & \wedge\, {}^{I}z^{\mathsf{R}} \mapsto \delta \wedge \delta : \texttt{I}[\tau_2^*] \wedge \llbracket E : \tau_3^* \rrbracket^{expr^*}_{s'} \\
 & & \wedge\, \tau_3^* \leq: \tau_2^* \wedge \llbracket F \rrbracket^{finit^*}_{s',s_c} \wedge !\overline{\mathcal{D}}(s') \\
 & & \wedge\, !^{V}\overline{\mathcal{R}}(s_c) \wedge {}^{V}\overline{\mathcal{R}}(s_c) \simeq {}^{V}\overline{\mathcal{D}}(s_c) \\[1ex]
\llbracket \texttt{this.}x_i \texttt{ = } y_j \rrbracket^{finit}_{s,s_c} & := & {}^{V}x_i^{\mathsf{R}} \longrightarrow s_c \wedge {}^{V}x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : \tau_1 \\
 & & \wedge\, \llbracket y_j : \tau_2 \rrbracket^{expr}_{s} \wedge \tau_2 \leq: \tau_1 \\[1ex]
\llbracket x_i\ \ y_j\texttt{(}A\texttt{) \{ }\textbf{\texttt{return}}\ e\texttt{; \}} \rrbracket^{mdecl}_{s,s_c} & := & s_c \longrightarrow {}^{M}y_j \wedge s' \xrightarrow{\mathbf{P}} s \wedge \llbracket x_i \rrbracket^{tyann}_{s,\tau_1} \\
 & & \wedge\, \llbracket A \rrbracket^{arg^*}_{s',\tau_2^*} \wedge \llbracket e : \tau_3 \rrbracket^{expr}_{s'} \wedge \tau_3 \leq: \tau_1 \\
 & & \wedge\, {}^{M}x_i^{\mathsf{D}} : \texttt{M}[\tau_2^*, \tau_1] \wedge !\overline{\mathcal{D}}(s') \\[1ex]
\llbracket x_i\ y_j : t \rrbracket^{arg}_{s} & := & s \longrightarrow {}^{V}y_j \wedge \llbracket x_i \rrbracket^{tyann}_{s,t} \wedge {}^{V}y_j^{\mathsf{D}} : t
\end{array}
$$

Any s, τ, or δ appearing free are assumed to be chosen fresh. The constraint generation function can also be applied to a list of terms. In that case the function is applied as $\llbracket T \rrbracket^{sort^*}_{s} \equiv \bigwedge_{t \in T} \llbracket t \rrbracket^{sort}_{s}$. Variables in a list invocation can be starred, as in $\llbracket T \rrbracket^{sort^*}_{s,\tau^*}$. This means a fresh τ is assumed for every invocation. We can refer to those variables as a list, using the same notation τ^*.
The parameters to the resolution calculus are the following:

$$
\begin{array}{rcl}
\mathcal{L} & := & \{\mathbf{S}, \mathbf{P}\} \\
l_1 < l_2 & := & \mathbf{S} < \mathbf{P} \\
\mathcal{E} & := & \mathbf{P}^* \cdot (\mathbf{S}^*)
\end{array}
$$

Figure 5.4: Constraint generation function for Featherweight Java

$$
\begin{array}{rcl}
[\![x_i : t]\!]^{expr}_{s} & := & {}^{V}x_i \longrightarrow s \wedge {}^{V}x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : t \\
[\![e\,.\,x_i : t]\!]^{expr}_{s} & := & [\![e]\!]^{expr}_{s,\mathtt{C}(\delta_1)} \wedge \delta_1 \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathsf{I}} \varsigma \wedge {}^{V}x_i \longrightarrow s' \\
 & & \quad \wedge {}^{V}x_i^{\mathsf{R}} \mapsto \delta_2 \wedge \delta_2 : t \\
[\![e\,.\,x_i(E) : t]\!]^{expr}_{s} & := & [\![e]\!]^{expr}_{s,\mathtt{C}(\delta_1)} \wedge \delta_1 \rightsquigarrow \varsigma \wedge s' \xrightarrow{\mathsf{I}} \varsigma \wedge {}^{V}x_i \longrightarrow s' \\
 & & \quad \wedge {}^{V}x_i^{\mathsf{R}} \mapsto \delta_2 \wedge \delta_2 : \mathtt{M}[\tau_1^*, t] \wedge [\![E : \tau_2^*]\!]^{expr^*}_{s} \\
 & & \quad \wedge \tau_2^* \leq: \tau_1^* \\
[\![\mathtt{new}\ x_i(E) : t]\!]^{expr}_{s} & := & {}^{C}x_i \longrightarrow s \wedge s' \xrightarrow{\mathsf{I}} {}^{C}x_i^{\mathsf{R}} \wedge {}^{C}x_i \mapsto \delta_1 \wedge t \overset{?}{=} \mathtt{C}(\delta_1) \\
 & & \quad \wedge {}^{I}x_i \longrightarrow s' \wedge {}^{I}x_i^{\mathsf{R}} \mapsto \delta_2 \wedge \delta_2 : \mathtt{I}[\tau_1] \\
 & & \quad \wedge [\![E : \tau_2^*]\!]^{expr^*}_{s} \wedge \tau_2^* \leq: \tau_1 \\
[\![(x_i)e : t]\!]^{expr}_{s} & := & [\![x_i]\!]^{tyann}_{s,t} \wedge [\![e : \cdot]\!]^{expr}_{s} \\
[\![x_i]\!]^{tyann}_{s,t} & := & {}^{C}x_i \longrightarrow s \wedge {}^{C}x_i^{\mathsf{R}} \mapsto \delta \wedge t \overset{?}{=} \mathtt{C}(\delta)
\end{array}
$$

Any s, τ, or δ appearing free are assumed to be chosen fresh. The constraint generation function can also be applied to a list of terms. In that case the function is applied as $[\![T]\!]^{sort^*}_{s} \equiv \bigwedge_{t \in T} [\![t]\!]^{sort}_{s}$. Variables in a list invocation can be starred, as in $[\![T]\!]^{sort^*}_{s,\tau^*}$. This means a fresh τ is assumed for every invocation. We can refer to those variables as a list, using the same notation τ^*.

Figure 5.5: Constraint generation function for Featherweight Java *(cont.)*

The semantic type of a class is written as $\mathtt{C}(d)$, where d is the class declaration. We give `this` the class type of the class being defined, and specify the subtyping using a supertype constraints. We will assume that the scope graph includes a declaration and class scope for the special type `Object`.

Fields Field definitions introduce new declarations in the class scope s_c. Declarations and references for fields use the variable namespace V. Fields are explicitly typed by an annotated class name. The class names are resolved in the lexical scope s, and we assign the corresponding class type to the field declaration.

Constructor The constructor introduces a declaration with the same name as the class, but using the namespace I. The constructor name should be the same as the class name. We therefore expect the class name as an argument y, and generate an equality constraint between y and the constructor name x. The constructor introduces a new lexical scope s', which contains the declarations of the constructor parameters. Because referring to `this` is not allowed in the constructor, the outer scope of the constructor is the outer scope of the class, not the scope of the class body. Constructor arguments are again explicitly typed, and use the V namespace. The type of a constructor is written as $\mathtt{I}[t^*]$, where t^* is the list of argument types.

To be able to invoke the super constructor, we expect the name of the super class as an argument z. We create a reference $\ulcorner z^{\mathsf{R}}$ to the constructor of the superclass. To type check the super call, we get the type $\mathtt{I}[\tau_2]$ of the super constructor, as well as the types τ_3 of the expressions that we pass to it. We check that the expression types are subtypes of the expected parameter types. We assume a simple extension to the solver, that interprets a subtype check between two lists of types as a list of subtype checks. The constraint will be satisfied if the two lists have the same length, and each type in the first list is a subtype of the type at the same position in the second list.

To initialize a field, we resolve it in the class scope, and capture its type in τ_1. We use τ_2 for the type of the expression, and specify that the expression type must be a subtype of the field type. Note that the syntax of Featherweight Java only allows references to be passed to super, or to initialize fields. However, our constraint generation function would not need to change if we allowed arbitrary expressions instead. We use name disambiguation constraints to ensure that all fields of the class are initialized exactly once.

Methods Methods are declared in the class scope s_c using the namespace M. A method definition introduces a new scope s' for the method body, which is a subscope of the scope of the class body. Method arguments are declared scope s'. The fact that the scope of the method body is a subscope of the scope of the class body, ensures that the special variable `this` is available in the method body. We use the type $\mathtt{M}[\tau_2^*, \tau_1]$ for methods, where τ_2^* is a list of parameter types, and τ_1 is the return type.

Expressions There are five expression forms in Featherweight Java: variable references, object creation, field access, method invocation, and type casts.

- Variable references introduce a reference in the current scope s. The type of the reference is made equal to the type of the declaration it resolves to.

- Objects are created by a **`new`** expression, which invokes the class constructor. To create a new x, a new scope s' is created, which imports the class scope of x. We create a reference for the constructor of x inside that scope, and use that to get the constructor type $\mathtt{I}[\tau_1]$. We require that the types of the expression we pass to the constructor are subtypes of the types of the constructor arguments.

- Method invocation and field access are specified similarly to record field access for LMR. We assume the expression has a class type $\mathtt{C}(\delta_1)$ with an associated scope ς. Scope ς is imported into a new scope s', in which we resolve the field or method name. In case of field access, the whole expression gets the type of the field. In case of a method call, the expression gets the return type of the method. We check that the types of the expressions passed as parameters to the method, are subtypes of the specified parameter types.

- A type cast changes the static type of the sub-expression, to an explicitly specified one. The type of the cast expression is simply the type corresponding to the specified class name x. Although we ignore the type of the sub-expression e, we do generate constraints for it, since we still want to catch type errors that can occur there.

Differences with original presentation There are two differences between the original presentation of the Featherweight Java semantics, and our specification:

- The original presentation is very specific about the order and names of the constructor parameters. The first parameters should have the exact same names and order as the parameter names for the super constructor, and should be passed directly to the `super` call. The parameters after that should match exactly the names and order of the fields defined in the class, and should be used to initialize those fields. This made the typing rules very simple to write down. In our constraint-based formulation, this restriction would not simplify the constraint generation, and is actually impossible to express. We can only reason about declaration and references as sets, and have no information about the order they appear in the code.

- The original presentation differentiates between three different possible scenarios for casts, (1) an upcast if the cast is to a supertype of the expression type, (2) a downcast if the cast is to a subtype of the expression type, and (3) a stupid cast otherwise, since the cast could never succeed at runtime. The typing is the same in all three rules, but in the last case it generates a warning. We have only one rule, and lose the possibility of a warning. Expressing the three rules is currently impossible in our constraint language, because we do not support disjunctions.

5.3 Prototype Implementation

We implemented the constraint solver from Chapter 4 in the Spoofax language workbench. We used this implementation to implement static analysis for the languages LMR, PCF, and Featherweight Java. We will first discuss the current state of static analysis in Spoofax, and some of its shortcomings. Then we will show how our approach addresses these shortcomings. Finally we explain the solver implementation, and discuss how it differs from the theoretical presentation we have given.

5.3.1 Static Analysis with Spoofax

Spoofax (Kats and Visser, 2010) is a language workbench, currently developed at the Software Engineering Research Group at the TU Delft. It provides domain specific languages for specifying syntax, name binding and type checking, and dynamic semantics. From these specifications it produces plugins for the Eclipse IDE, allowing easy and immediate integration of the language in a familiar development environment.

Name binding is specified using the NaBL language (Konat et al., 2012), and type checking using the TS language. Both languages generate name resolution and type checking code, which is executed using a generic Task Engine (Wachsmuth et al., 2013), which provides some incrementality. Name resolution and type analysis are executed simultaneously, which allows mutual dependence between the two. It is therefore possible to express forms of type-dependent name resolution, and type-based disambiguation.

Although NaBL and TS have proven useful in many projects, their ad hoc design and lack of formal basis are a problem. For example, NaBL has no clearly defined semantics, and some of its features are the result of the implementation more than of language design. This makes it unnecessarily hard to understand complex sets of rules. TS lacks support for parametric polymorphism or generics, ruling out important classes of languages. Furthermore, the implementation requires that all calculations happen in an down-up traversal, which makes most forms of non-local type inference impossible. And, despite its incremental design, the Task Engine does not scale to large projects as well as was hoped. Finally, proving properties such as type soundness about a language whose static analysis is specified in NaBL and TS rules is practically impossible, since their semantics are not formally defined.

5.3.2 Benefits of the Constraint-based Approach

By implementing our constraint approach in Spoofax, we have shown that it is a realistic approach to specifying static analysis. We have been able to address some of current shortcomings, while others remain future work. Specifically, our approach addresses the following issues:

- Name binding and type checking have a formal basis in the constraint semantics. This gives a formal notion of solver soundness, and also opens up possibilities for type-soundness proofs for in constraint-based semantics.

- The separation between constraint generation and constraint solving, enables non-local reasoning, such as global type inference.

Efficiency and scalability of the solver, especially in the face of large projects and compilation units is still an open question, and is the subject of future work.

5.3.3 Implementation Review

The solver algorithm was implemented using the Stratego transformation language. This allowed us to implement the solver as a set of rewrite rules on a context, close to the presentation in the previous chapter. There are however some deviations from our presentation.

- Stratego does not support matching modulo commutativity. Transformation rules match on one constraint only, and not on a conjunction. The solver algorithm ensures that if the first constraint of the conjunction cannot be solved, the next will be tried.

- Although performance was not an objective for this thesis, we included some optimizations, which improved usability of the system. Since constraints sometimes depend on other constraints, e.g. name resolution depends on a complete scope graph, and subtyping on a complete subtyping relation, we order the constraints. Equality constraints are always solved first, since they have no side conditions. Scope graph constraints are solved before name resolution constraints, and supertype constraints before subtype checks. Finally, instead of applying the substitution in rule S-ELIMINATE to all remaining constraints immediately, we only apply it to an individual constraint just before we try to solve it.

- Just returning $\perp$ to the user when no solution to the constraints is found is not very informative. Therefore, instead of aborting the solver, we generate an error, but continue to solve the remaining constraints. To be able to present more informative error messages to the user, the implementation contains extra rules for error cases, which create messages appropriate for the constraint that failed.

  Error reporting for constraint problems is a known problem (Hage and Heeren, 2006), because the order of constraint solving can cause the conflict to be detected on different constraints. For example, errors might be reported on the declaration, when it makes more sense on the reference, or vice versa. Our current approach is naive in this sense, and generating better error messages is the subject of future work.

Chapter 6

Related Work

In this chapter we discuss related research on static analysis. First we will discuss other constraint-based solutions. Then we will look at other language-independent approaches to static analysis. We will focus specifically on the handling of name binding, and interaction between name binding and type checking.

6.1 Constraint-based Approaches to Static Analysis

There is a long tradition of formulating Hindley/Milner type systems (Milner, 1978) using constraint. All of these approaches are based on types as terms, and first-order unification to solve equalities. The original inference algorithm $\mathcal{W}$ is already expressed in terms of Robinson's unification algorithm to solve type equalities. However, there is no separate constraint collection, instead equalities are solved as soon as they are encountered. The algorithm propagates partial solutions, by threading a substitution and a typing environment through the computation. Odersky et al. (1999) present an extension to Hindley/Milner type systems, that is parametrized in a constraint domain X. Their type inference algorithm is defined over the judgment $\psi, C, \Gamma \vdash e : t$, where ψ is a substitution, C a constraint in X, and Γ a typing environment. Apart from being parametrized in a constraint domain, the inference algorithm is similar to the original algorithm $\mathcal{W}$. This work was improved on by Sulzmann (2001), who describes a typing algorithm based on the judgment $C, \Gamma, e : t \vdash D$. Instead of calculating a type assignment, there is a hypothetical type assignment before the turnstile, whose validity depends on the satisfiability of the output constraint D. This allows separating the constraint solver from the constraint collection.

This idea is taken further by Pottier and Rémy (2005) in their presentation of HM(X) and an efficient solver for it. They maintain the separation between constraint generation and solving, but instead of relying on a type environment Γ in the constraint generation, they mimic the binding structure of the program in the constraints. They introduce a constraint $\texttt{def x} : t \texttt{ in } C$ to introduce a binding in C, and a constraint $\texttt{x} \preceq T$ to instantiate the type of the binding. Their constraint generation function is written as $[\![e : t]\!]$, and indeed does not depend on a typing environment anymore. Although name binding is part of the constraint problem, it is limited to the lexical structure of the original program, and the constraints do not introduce more flexibility with respect to name binding patterns.

Compared to our approach, the support for polymorphism and the parametrization in an arbitrary constraint domain X are very attractive. Integrating them in our approach is a topic of future work. On the other hand, none of them support the flexibility in name binding our approach offers.

Erdweg et al. (2015) introduced an interesting variant on traditional type system specifications with co-contextual typing rules. It is based on equality constraints as well, and relies on a simple lexical scoping model. However, instead of passing down a type environment in a top-down traversal, they proceed in a bottom-up manner. They use a judgment $e : t \mid C \mid R$, where C consists of equality constraints, and R is a set of requirements. The requirements are dual to a traditional typing environment, and contain declarations and type assignments that need to be fulfilled by the surrounding program. The bottom-up approach allows them to do parallel and incremental type checking. It is certainly be possible to collect our constraints in a bottom-up manner, by using constraint variables to stand for yet unknown concrete scopes. However, the incrementality in their work relies on the possibility to solve constraints locally, and minimize the information that is passed up. The possibility of shadowing declarations makes name resolution in our model non-monotonic. Whether it is possible to adept this to fit in a co-contextual model is still an open question.

Work by Hage and Heeren (2009); Heeren et al. (2003), based on a traditional Hindley/Milner type system, exploits the fact that there is no inherent order in constraint solving to improve error reporting. If an error occurs, i.e. if some constraints are not satisfiable, constraints are removed from the constraint set until the remaining constraints can be satisfied. The eliminated constraints are turned into error messages. Heuristics are used to select the constraints that result in the most informative error messages. Some of these are generic, e.g. a constraint involved in many conflicts is more likely to result in an error. However, most of them are specific to the language Haskell, to which the authors applied their work. Given that our approach tries to target a range of languages, it is unclear how well their work is applicable.

For static analysis of object-oriented type systems, the work by Palsberg (1996) takes a different approach, and uses set inclusion constraints. Unfortunately name binding issues are explicitly ignored, so we cannot compare e.g. how name and type resolution interact for field access.

6.2 Other Approaches to Static Analysis

Since our approach aims to support static analysis for a range of languages, we will have a look at two other language-independent approaches, attribute grammars, and generated type checkers.

A common approach to the implementation of static semantics of programming languages is to use attribute grammars (Knuth, 1968). Attribute grammars are defined as properties of AST nodes, and do not enjoy the separation between the specifics of the language, and the logic of name resolution and type checking. Originally, attributes were values, which meant name resolution was implemented as computing and propagating typing environments. Kastens and Waite (1991) provide a reusable abstract data type for name resolution, that has some similarities with the scope graph model. Reference attribute grammars (Hedin, 2000) allow references (as in names from the program), to link directly to their corresponding declaration. Ekman and Hedin (2006) use this to provide a generic name binding library. Although this library is developed as part of the JastAdd compiler, it can be used independently. This generic framework is instantiated for a language, by specifying language specific name lookup functions per language construct. This allows a specification close to e.g. the Java Language Specification. It does however mean, that many aspects, such as lexical scoping or shadowing, are encoded programmatically. Our use of scope graphs allows a more declarative specification of such aspects.

Another approach is generating type checkers from high-level specification of type systems. Work by Gast (2005) generates type checkers based on proof search, and supports type systems for functional, imperative, and object-oriented languages. The specification can be annotated with optimizations to create efficient implementations. Wittmann (2014) proposes a simpler system, that replaces proof search with unification based constraint solving. This allows him to automate the search for optimizations in the type system rules. Both these approaches support high-level specification languages, close to the presentation of type systems in text books and research literature. Our constraint language is low-level, but could be used as a back-end for such higher-level specification languages.

Chapter 7

Discussion

In this chapter we will discuss the capabilities and limitations of the constraint language, the solver, and its implementation in Spoofax. We will also point out possible topics for future research.

7.1 Constraint Language

To evaluate the expressiveness of the constraint language, we used it to specify the static semantics of two languages from the research literature. We use the simply-typed functional language PCF, and the object-oriented language Featherweight Java. This shows that our approach is suitable for specification of the static semantics of a variety of languages, represent different language paradigms. There are however limitations and open questions, that we want to discuss here. We will start by discussing name binding, then type checking and we finish with considering the possibilities of formal reasoning.

Name binding The name resolution calculus allows us to express a wide range of name binding patterns. The introduction of incomplete scope graphs, and therefore interaction between name resolution and type checking, has increased this range further. However, until now we have not attempted to express the semantics for a full, real-word language, such as Java, or C#. For example, methods and fields often have access modifiers – like *public, private* – that influence their visibility in different contexts. It is not possible to express such patterns in our constraint language. Several interesting issues arise when considering method overloading and overriding for languages such as Java, or C#.

- Sometimes duplicate method names are allowed, as long as they are distinguishable by their type, e.g. with method overloading in Java. A limited form of this is supported by exploiting the non-determinism resulting from ambiguous resolutions. However, it would be impossible to enforce this, for example, for a method that is never called. Especially in the face of multiple compilation units, it becomes necessary to specify such requirements explicitly.

- Shadowing sometimes depends on type information. Consider the following Java classes:

```
// Y extends X
class A { void m(Y y) { ... } }
class B extends A { void m(X x) { ... } }
```

  Given an instance `b` of class `B`, a method call `b.m(y)` should resolve to the method in `A`, since its type is more specific than that of the method in `B`. This cannot be expressed in our current system, since the method in `B` shadows the method in `A`.

An interesting direction for future work is to investigate if the ability to add extra meta-data to nodes or edges in the scope graph, and using this data in the path ordering or the well-formedness predicate, allows specification of the patterns described above.

Finally, we want to discuss ambiguous resolutions. Currently, ambiguous resolutions introduce a non-deterministic choice. This enables a limited form of type-based disambiguation between visible declarations. If unwanted, ambiguous names can be restricted using disambiguation constraints. For some languages one might want to be able to reason about the full set of resolved declarations for a reference.

Type checking The focus of this thesis is on enabling a wider variety of name binding patterns in constraint-based static analysis. Although syntactic equality and unification are at the basis of many type systems, it is limited on its own. An important addition to the constraint language would be to support various forms of parametric polymorphism. Polymorphism comes in many flavors, from completely explicit generics in Java, to completely inferred in languages with let-polymorphism, such as ML and Haskell. Both styles are similar in the sense that they require instantiation of type variables at the use site of an identifier. In let-polymorphic languages the variables that need to be instantiated are part of the type, often written as $\forall\tau.\,t$. In object-oriented generics, it is more complicated. The type variable is defined on the class level, but might be used in the types of class members. The instantiation is determined by the object, but must be applied for every member access on that object. An interesting direction for future research is, to investigate if it is possible to encode this information in the scope graph, for example by attaching type variable substitutions to edges. Ideally, we would find a uniform model for let-polymorphism, where quantification is local to a single declaration, and generics-style polymorphism, where the quantification is – loosely speaking – over a scope.

The subtyping model we support is limited, and a somewhat ad hoc addition to the system. It suffices to model the simple, single-inheritance model required by Featherweight Java. However, to model real-world languages, we need to consider features such as inheritance, interface-based subtyping, traits, value boxing and so on. We hope to be able to separate the details of such specific type system features from the general setup of the constraint language.

This might lead to a formulation similar to HM(X), in the sense that constraint system is parametrized in an arbitrary constraint domain X.

Formal reasoning In the study and design of programming languages, type soundness proofs play an important role. In the introduction we stated that a formal language for static semantics was desired, specifically to enable such soundness proofs. Making the connection between a constraint-based static semantics, and the dynamic semantics of a language, is an important aspect of further work.

7.2 Solver

The solver is at its core a non-deterministic rewrite system. It supports progressive solving of constraints, but no backtracking or search. We have already seen from the incompleteness examples, that this system has clear limitations. But even though the system is incomplete in general, the incomplete cases never seem to occur for the example languages we have investigated. An interesting question is therefore whether this holds for a larger set of languages, or if this will quickly become a problem in practice. It is also interesting to consider the point made by Vytiniotis et al. (2011), i.e. that "by explaining that the inference algorithm does not 'guess' types, or 'search' among possible substitutions, we have found that programmers can, after some experience, accurately predict what should and should not type-check".

Implementing our own constraint solver, instead of using an existing solver, improved our understanding of many issues related to constraint solving. However, optimizing the implementation for performance has not been carried out yet. This aspect can be addressed by building on existing constraint solving literature and implementations. For example, our implementation of unification has exponential run time in the worst case. Implementing an algorithms based on union-find data structures, which yield linear or mostly linear run times, should considerably benefit solver performance.

Another interesting issue is separate compilation. In a large project with multiple compilation units, it is desirable that compilation is incremental. If one compilation unit is changed, the work necessary to check other compilation units should be minimized, by caching and reusing previous analysis results. Since connections between compilation units are usually based on name resolution, we hope that an analysis of the scope graph of a compilation unit can automatically tell us which part of the analysis is local and which part depends on changes in other compilation units.

The usability of static analysis greatly depends on the quality of the error messages it generates. We have discussed existing work on error reporting for constraint-based type checking. Application of these or similar approaches are left for for future work.

Finally, our experience learned that checking and maintaining the soundness proof of our solver is laborious and error prone. Having a mechanized proof would be a great improvement, and a topic of future research.

7.3 Prototype Implementation

As we mentioned in the previous section, when moving forward to a production-quality solver, it is valuable to look at existing constraint solvers. Given that the performance of constraint solvers – e.g. SMT solvers – has increased many-fold in recent years, we might consider using one of them as a basis for our system. We should consider several aspects. First, no existing solvers support the name resolution calculus, so any chosen solver would need to be extended from the start. Second, the solver should be integrated in Spoofax, which is difficult if the solver is not JVM-based. Generally, many aspects of the solver might be hard to influence. For example, currently the user can influence the error messages that are generated when a constraint cannot be solved. If we do not have influence on the error messages or how they are propagated, this might not be possible when using an external solver. How to go from our prototype to a high-quality, scalable solver is therefore still an open research question.

Bibliography

Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964. doi: db/journals/jacm/Brzozowski64.html.

Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 2006. ISBN 3-540-45778-X. doi: http://dx.doi.org/10.1007/11877028_18.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches - conclusions from the language workbench challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. ISBN 978-3-319-02653-4. doi: http://dx.doi.org/10.1007/978-3-319-02654-1_11.

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 880–897. ACM, 2015. ISBN 978-1-4503-3689-5. doi: http://doi.acm.org/10.1145/2814270.2814277.

Martin Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

Holger Gast. *A generator for type checkers*. PhD thesis, Eberhard Karls University of Tübingen, 2005. http://d-nb.info/977024180.

Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2006. ISBN 978-3-540-74129-9. doi: http://dx.doi.org/10.1007/978-3-540-74130-5_12.

Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electronic Notes in Theoretical Computer Science*, 236:163–183, 2009. doi: http://dx.doi.org/10.1016/j.entcs.2009.03.021.

Robert Harper. *Practical Foundations for Programming Languages*. 2012.

Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing hindley-milner type inference algorithms. Technical Report UU-CS-2002-031, Department of Information and Computing Sciences, Utrecht University, 2002.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 3–13. ACM, 2003. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944707.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. doi: http://doi.acm.org/10.1145/503502.503505.

Uwe Kastens and William M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28(6):539–558, 1991.

Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869497.

Donald E. Knuth. Semantics of context-free languages. *Theory Comput. Syst.*, 2(2):127–145, 1968. doi: http://www.springerlink.com/content/m2501m07m4666813/.

Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. ISBN 978-3-642-36089-3. doi: http://dx.doi.org/10.1007/978-3-642-36089-3_18.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *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*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. ISBN 978-3-662-46668-1. doi: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

Jens Palsberg. Type inference for objects. *ACM Computing Surveys*, 28(2):358–359, 1996.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):225–255, 1977.

François Pottier and Diddier Rémy. The essence of ml type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005. ISBN 0-262-16228-8.

Didier Rémy. Type systems for programming languages. `http://pauillac.inria.fr/~remy/mpri/cours3.pdf`, 2015. Course notes.

Martin Sulzmann. A general type inference framework for hindley/milner style systems. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, volume 2024 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2001. ISBN 3-540-41739-7. doi: http://link.springer.de/link/service/series/0558/bibs/2024/20240248.htm.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/milner style type systems in constraint form. Technical Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, January 2016. doi: http://dx.doi.org/10.1145/2847538.2847543.

Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. ISBN 978-1-4503-3210-1. doi: http://doi.acm.org/10.1145/2661136.2661149.

Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. A language independent task engine for incremental name and type analysis. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2013. ISBN 978-3-319-02653-4. doi: http://dx.doi.org/10.1007/978-3-319-02654-1_15.

Pascal Wittmann. A language for the specification and efficient implementation of type systems. Master's thesis, 2014.

Appendix A

Proofs

A.1 Reduction Invariant

We will prove the invariants of the reduction by case analysis on the reduction rules. Before we do this, we recall some properties we will use in the proof. We have the following properties of substitutions (Baader and Snyder, 2001):

1. The composition of two idempotent substitutions σ,σ' is idempotent if $\mathrm{dom}(\sigma) \cap \mathrm{vars}(\mathrm{ran}(\sigma')) = \emptyset$.
2. If t is ground, i.e. $\mathrm{vars}(t) = \emptyset$, substitution has no effect, so $\forall\sigma.\, t\sigma = t$.

Regarding set union and the subset relation, we have:

$$\forall S S'.\, \big(S \subseteq S' \cup S\big) \tag{†}$$

And finally, if the constraint C does not contain scope graph constraints, the set of edges in the graph remains the same for the rest of the reduction, i.e.

LEMMA A.1 (Stable scope graph).

$$\forall C,\Delta,\mathcal{G},\Delta'',\mathcal{G}''.\, \Big(C \cap C^{\mathcal{G}} = \emptyset \wedge \langle C;\Delta,\mathcal{G}\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta'',\mathcal{G}''\rangle \implies \exists\sigma.\,\mathcal{G}\sigma = \mathcal{G}''\Big)$$

Proof. Assume a context with scope graph $\mathcal{G}$, and a constraint C that contains no scope graph constraints, i.e. $C \cap C^{\mathcal{G}}$. According to our reduction invariant, the scope graph $\mathcal{G}''$ is a larger or equal instantiation of $\mathcal{G}$, i.e. $\exists\sigma.\,\mathcal{G}\sigma \subseteq \mathcal{G}''$. There are two possible cases, either $\exists\sigma.\,\mathcal{G}\sigma = \mathcal{G}''$ or $\exists\sigma.\,\mathcal{G}\sigma \subset \mathcal{G}''$. We will show that the second case results in contradiction. Only rules that match on constraints from $C^{\mathcal{G}}$ add elements to $\mathcal{G}$. None of the reduction rules introduces new constraints from $C^{\mathcal{G}}$, so the constraints that caused the extension of $\mathcal{G}$ must be in C already. This is a contradiction, therefore the conclusion holds. □

Note that we use the reduction invariant Equation (WF) in our proof, and we use this lemma in the proof of the reduction invariant. Because the multi-step reduction in the invariant proof never includes the reduction step that we are proving the invariant for, this is well-founded.

Now we can prove the following invariants for every possible reduction step.

LEMMA 4.7 (Invariant of $\longrightarrow$). *The following invariant holds for the reduction:*

$$\begin{aligned}
\forall C \Delta C' \Delta'. \Big(\langle C; \Delta\rangle \longrightarrow \langle C'; \Delta'\rangle \implies \\
& \big(\mathrm{WF}(\langle C;\Delta\rangle) \implies \mathrm{WF}(\langle C';\Delta'\rangle)\big) && \text{(WF)}\\
& \land\, \Delta \subseteq \Delta' && \text{(Sub)}\\
& \land\, \forall \Delta''. \,\big(\langle C';\Delta'\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta''\rangle \land \mathrm{WF}(\langle C';\Delta'\rangle) && \text{(Star)}\\
& \qquad \land\, \Delta'' \models C' \implies \Delta'' \models C\big)\Big)
\end{aligned}$$

Proof. By case analysis on the reduction rules.

⟨1⟩1. CASE: Rule S-TRUE
 ASSUME: 1. $\langle \mathsf{True} \land C; \Delta\rangle \longrightarrow \langle C;\Delta\rangle$
 ⟨2⟩1. ASSUME: $\mathrm{WF}(\langle \mathsf{True} \land C;\Delta\rangle)$
  PROVE: $\mathrm{WF}(\langle C;\Delta\rangle)$
  PROOF: Trivial.
 ⟨2⟩2. PROVE: $\Delta \subseteq \Delta$
  PROOF: By reflexivity of $\subseteq$.
 ⟨2⟩3. ASSUME: 1. Any Δ''
   2. $\langle C;\Delta\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta''\rangle$
   3. $\mathrm{WF}(\langle C;\Delta\rangle)$
   4. $\Delta'' \models C$
  PROVE: $\Delta'' \models \mathsf{True} \land C$
  ⟨3⟩1. SUFFICES: $\Delta'' \models \mathsf{True}$
   PROOF: By application of rule C-CONJ, given that $\Delta'' \models C$ follows from assump. ⟨2⟩3-4.
  ⟨3⟩2. Q.E.D.
   PROOF: By rule C-TRUE.

⟨1⟩2. CASE: Rule S-TRIVIAL
 ASSUME: 1. $\langle t \overset{?}{=} t \land C; \Delta\rangle \longrightarrow \langle C;\Delta\rangle$
 ⟨2⟩1. ASSUME: $\mathrm{WF}(\langle t \overset{?}{=} t \land C;\Delta\rangle)$
  PROVE: $\mathrm{WF}(\langle C;\Delta\rangle)$
  PROOF: Trivial.
 ⟨2⟩2. PROVE: $\Delta \subseteq \Delta$
  PROOF: By reflexivity of $\subseteq$.
 ⟨2⟩3. ASSUME: 1. Any Δ''
   2. $\langle C;\Delta\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta''\rangle$
   3. $\mathrm{WF}(\langle C;\Delta\rangle)$
   4. $\Delta'' \models C$
  PROVE: $\Delta'' \models t \overset{?}{=} t \land C$
  ⟨3⟩1. SUFFICES: $\Delta'' \models t \overset{?}{=} t$
   PROOF: By application of rule C-CONJ, given that $\Delta'' \models C$ follows from assump. ⟨2⟩3-4.
  ⟨3⟩2. Q.E.D.
   PROOF: By trivial term equality, and rule C-EQUAL.

⟨1⟩3. CASE: Rule S-ORIENT
ASSUME: 1. $\langle t \overset{?}{=} \alpha \wedge C; \Delta \rangle \longrightarrow \langle \alpha \overset{?}{=} t \wedge C; \Delta \rangle$
2. $t \notin Var$
⟨2⟩1. ASSUME: $\mathrm{WF}(\langle t \overset{?}{=} \alpha \wedge C; \Delta \rangle)$
PROVE: $\mathrm{WF}(\langle \alpha \overset{?}{=} t \wedge C; \Delta \rangle)$
PROOF: Trivial.
⟨2⟩2. PROVE: $\Delta \subseteq \Delta$
PROOF: By reflexivity of $\subseteq$.
⟨2⟩3. ASSUME: 1. Any Δ''
2. $\langle \alpha \overset{?}{=} t \wedge C; \Delta \rangle \longrightarrow^* \langle \mathsf{True}; \Delta'' \rangle$
3. $\mathrm{WF}(\langle \alpha \overset{?}{=} t \wedge C; \Delta \rangle)$
4. $\Delta'' \models \alpha \overset{?}{=} t \wedge C$
PROVE: $\Delta'' \models t \overset{?}{=} \alpha \wedge C$
⟨3⟩1. SUFFICES: $\Delta'' \models t \overset{?}{=} \alpha$
PROOF: By application of rule C-CONJ, given that $\Delta'' \models C$ follows from assump. ⟨2⟩3-4.
⟨3⟩2. Q.E.D.
PROOF: Let φ'' be the substitution in Δ''. By assump. ⟨2⟩3-4 and inversion of rules C-CONJ and C-EQUAL we get $\alpha\varphi'' = t\varphi''$. Commutativity of $=$ and application of rule C-EQUAL give us $\Delta'' \models t \overset{?}{=} \alpha$.

⟨1⟩4. CASE: Rule S-ELIMINATE
ASSUME: 1. $\langle \alpha \overset{?}{=} t \wedge C; \Delta, \varphi \rangle \longrightarrow \langle C; \Delta, \varphi \rangle\, \sigma$
2. $\alpha \notin \mathrm{vars}(t)$
3. $\sigma := \{\alpha \mapsto t\}$
⟨2⟩1. ASSUME: $\mathrm{WF}(\langle \alpha \overset{?}{=} t \wedge C; \Delta, \varphi \rangle)$
PROVE: $\mathrm{WF}(\langle C; \Delta, \varphi \rangle\, \sigma)$
⟨3⟩1. $\varphi\sigma$ is idempotent
PROOF: By assump. ⟨1⟩4-2, we conclude that σ is idempotent. By the assumption that Δ is well-formed, and Corollary 4.3, we conclude that φ is idempotent. By the assumption that Δ is well-formed, and Corollary 4.4, we know that $\mathrm{vars}(t) \cap \mathrm{dom}(\varphi) = \emptyset$. Therefore, the composition $\varphi\sigma$ is idempotent.
⟨3⟩2. $\langle C; \Delta, \varphi \rangle \sigma\varphi\sigma = \langle C; \Delta, \varphi \rangle \sigma$
PROOF: We show equality by equational reasoning:

$$\begin{aligned} & \langle C; \Delta, \varphi \rangle \sigma\varphi\sigma \\ &= \langle C; \Delta, \varphi \rangle \varphi\sigma\varphi\sigma && (\langle C; \Delta \rangle \varphi = \langle C; \Delta \rangle \text{ by assump. } \langle 2 \rangle 1) \\ &= \langle C; \Delta, \varphi \rangle \varphi\sigma && (\text{by idempotence of } \varphi\sigma) \\ &= \langle C; \Delta, \varphi \rangle \sigma && (\langle C; \Delta \rangle \varphi = \langle C; \Delta \rangle \text{ by assump. } \langle 2 \rangle 1) \end{aligned}$$

⟨3⟩3. $\forall x_i^{\mathsf{R}}. \left(\mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}} \implies \vdash_{\mathcal{G}\sigma} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}} \right)$
PROOF: Given that $\forall x_i^{\mathsf{R}}. \left(\mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}} \right)$ follows from our assumption, and $\vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x_i^{\mathsf{D}} \implies \forall \sigma. \vdash_{\mathcal{G}\sigma} x^{\mathsf{R}} \mapsto x_i^{\mathsf{D}}$ (van Antwerpen et al., 2016, ⧫).

⟨3⟩4. Q.E.D.
PROOF: By ⟨3⟩2.

⟨2⟩2. PROVE: $\Delta, \varphi \subseteq (\Delta, \varphi)\sigma$
PROOF: By using σ in the definition of $\subseteq$.

⟨2⟩3. ASSUME: 1. Any Δ'', φ''
2. $\langle C; \Delta, \varphi\rangle\, \sigma \longrightarrow^* \langle \mathsf{True}; \Delta'', \varphi''\rangle$
3. $\mathrm{WF}(\langle C; \Delta, \varphi\rangle\, \sigma)$
4. $\Delta'', \varphi'' \models C\sigma$

PROVE: $\Delta'', \varphi'' \models \alpha \overset{?}{=} t \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \varphi'' \models \alpha \overset{?}{=} t$
PROOF: By application of rule C-CONJ, given that $\Delta'', \varphi'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. SUFFICES: $\alpha\varphi = t\varphi$
PROOF: By inversion of rule C-EQUAL.

⟨3⟩3. $(\Delta, \varphi)\sigma \subseteq \Delta'', \varphi''$
PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩4. Q.E.D.
PROOF: We show by equational reasoning.

$$\begin{aligned} &\alpha\varphi'' = t\varphi'' \\ \Rightarrow\ &\alpha\varphi\sigma\sigma' = t\varphi\sigma\sigma' && (\exists\sigma'.\, \varphi\sigma\sigma' = \varphi'' \text{ by } \langle 3\rangle 3) \\ \Rightarrow\ &\alpha\sigma\sigma' = t\sigma\sigma' && (\alpha\varphi = \alpha \text{ and } t\varphi = t \text{ by assump. } \langle 2\rangle 3\text{-}3) \\ \Rightarrow\ &t\sigma' = t\sigma' && (\text{by application of } \sigma \text{ and assump. } \langle 1\rangle 4\text{-}2) \end{aligned}$$

⟨1⟩5. CASE: Rule S-DECOMPOSE
ASSUME: 1. $\langle f(t_1, \ldots, t_n) \overset{?}{=} f(t'_1, \ldots, t'_n) \wedge C; \Delta\rangle \longrightarrow \langle (\bigwedge_{i=1}^n t_i \overset{?}{=} t'_i) \wedge C; \Delta\rangle$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle f(t_1, \ldots, t_n) \overset{?}{=} f(t'_1, \ldots, t'_n) \wedge C; \Delta\rangle)$
PROVE: $\mathrm{WF}(\langle (\bigwedge_{i=1}^n t_i \overset{?}{=} t'_i) \wedge C; \Delta\rangle)$
PROOF: Trivial.

⟨2⟩2. PROVE: $\Delta \subseteq \Delta$
PROOF: By reflexivity of $\subseteq$.

⟨2⟩3. ASSUME: 1. Any Δ''
2. $\langle (\bigwedge_{i=1}^n t_i \overset{?}{=} t'_i) \wedge C; \Delta\rangle \longrightarrow^* \langle \mathsf{True}; \Delta''\rangle$
3. $\mathrm{WF}(\langle (\bigwedge_{i=1}^n t_i \overset{?}{=} t'_i) \wedge C; \Delta\rangle)$
4. $\Delta'' \models (\bigwedge_{i=1}^n t_i \overset{?}{=} t'_i) \wedge C$

PROVE: $\Delta'' \models f(t_1, \ldots, t_n) \overset{?}{=} f(t'_1, \ldots, t'_n) \wedge C$

⟨3⟩1. SUFFICES: $\Delta'' \models f(t_1, \ldots, t_n) \overset{?}{=} f(t'_1, \ldots, t'_n)$
PROOF: By application of rule C-CONJ, given that $\Delta'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. Q.E.D.
PROOF: Because of assump. ⟨2⟩3-4, we know $\forall t_i, t'_i.\, t_1\varphi = t'_i\varphi$. Therefore $f(t_1\varphi, \ldots, t_n\varphi) = f(t'_1\varphi, \ldots, t'_n\varphi)$, and by definition of substitution, $f(t_1, \ldots, t_n\varphi) = f(t'_1, \ldots, t'_n\varphi)$. Rule C-EQUAL gives $\Delta'' \models f(t_1, \ldots, t_n) \overset{?}{=} f(t'_1, \ldots, t'_n)$.

⟨1⟩6. CASE: Rule S-TYPEOF1
ASSUME: 1. $\langle x_i^{\mathsf{D}} : t \wedge C; \Delta, \psi\rangle \longrightarrow \langle C; \Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi\rangle$

2. $\psi(x_i^{\mathsf{D}}) = \bot$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle x_i^{\mathsf{D}} : t \wedge C; \Delta, \psi\rangle)$
PROVE: $\mathrm{WF}(\langle C; \Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi\rangle)$
PROOF: Trivial.

⟨2⟩2. PROVE: $\Delta, \psi \sqsubseteq \Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi$
PROOF: By Equation (†) on ψ.

⟨2⟩3. ASSUME: 1. Any Δ'', ψ''
2. $\langle C; \Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi\rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \psi''\rangle$
3. $\mathrm{WF}(\langle C; \Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi\rangle)$
4. $\Delta'', \psi'' \models C$

PROVE: $\Delta'', \psi'' \models x_i^{\mathsf{D}} : t \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \psi'' \models x_i^{\mathsf{D}} : t$
PROOF: By application of rule C-CONJ, given that $\Delta'', \psi'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. SUFFICES: $\psi''(x_i^{\mathsf{D}}\varphi'') = t\varphi''$
PROOF: By inversion of rule C-TYPEOF.

⟨3⟩3. $\Delta, \{(x_i^{\mathsf{D}}, t)\} \cup \psi \sqsubseteq \Delta'', \psi''$
PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩4. Q.E.D.
PROOF: Let φ, φ'' be the substitution components of Δ, Δ'' respectively. By equational reasoning:

$$\begin{aligned} & \exists\sigma.\, \psi''(x_i^{\mathsf{D}}) = t\sigma && \text{(By ⟨3⟩3)} \\ \Rightarrow\ & \psi''(x_i^{\mathsf{D}}) = t\varphi\sigma && (t\varphi = t \text{ by assump. ⟨2⟩3-3}) \\ \Rightarrow\ & \psi''(x_i^{\mathsf{D}}) = t\varphi'' && (\varphi\sigma = \varphi'' \text{ by ⟨3⟩3}) \\ \Rightarrow\ & \psi''(x_i^{\mathsf{D}}\varphi'') = t\varphi'' && (x_i^{\mathsf{D}}\varphi'' = x_i^{\mathsf{D}}, \text{ since } x_i^{\mathsf{D}} \text{ is ground}) \end{aligned}$$

⟨1⟩7. CASE: Rule S-TYPEOFN
ASSUME: 1. $\langle x_i^{\mathsf{D}} : t \wedge C; \Delta, \varphi, \psi\rangle \longrightarrow \langle t \overset{?}{=} t' \wedge C; \Delta, \varphi, \psi\rangle$
2. $\psi(x_i^{\mathsf{D}}) = t'$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle x_i^{\mathsf{D}} : t \wedge C; \Delta, \varphi, \psi\rangle)$
PROVE: $\mathrm{WF}(\langle t \overset{?}{=} t' \wedge C; \Delta, \varphi, \psi\rangle)$
PROOF: Trivial.

⟨2⟩2. PROVE: $\Delta, \varphi, \psi \sqsubseteq \Delta, \varphi, \psi$
PROOF: By reflexivity of $\sqsubseteq$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \varphi'', \psi''$
2. $\langle t \overset{?}{=} t' \wedge C; \Delta, \varphi, \psi\rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \varphi'', \psi''\rangle$
3. $\mathrm{WF}(\langle t \overset{?}{=} t' \wedge C; \Delta, \varphi, \psi\rangle)$
4. $\Delta'', \varphi'', \psi'' \models t \overset{?}{=} t' \wedge C$

PROVE: $\Delta'', \varphi'', \psi'' \models x_i^{\mathsf{D}} : t \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \varphi'', \psi'' \models x_i^{\mathsf{D}} : t$
PROOF: By application of rule C-CONJ, given that $\Delta'', \varphi'', \psi'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. SUFFICES: $\psi''(x_i^{\mathsf{D}}\varphi'') = t\varphi''$
PROOF: By inversion of rule C-TYPEOF.

⟨3⟩3. $\Delta, \varphi, \psi \sqsubseteq \Delta'', \varphi'', \psi''$
PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩4. Q.E.D.
PROOF: Assump. ⟨2⟩3-4, inversion of rule C-EQUAL, commutativity of $=$ and substituting t' gives us

$$\begin{aligned} &\psi(x_i^{\mathsf{D}})\varphi'' = t\varphi'' \\ \Rightarrow\ &\psi''(x_i^{\mathsf{D}}) = t\varphi'' && (\text{by } \langle 3\rangle 3 \text{ and } x_i^{\mathsf{D}} \in \mathrm{dom}(\psi)) \\ \Rightarrow\ &\psi''(x_i^{\mathsf{D}}\varphi'') = t\varphi'' && (x_i^{\mathsf{D}}\varphi'' = x_i^{\mathsf{D}} \text{ since } x_i^{\mathsf{D}} \text{ is ground}) \end{aligned}$$

⟨1⟩8. CASE: Rule S-GDECL
ASSUME: 1. $\left\langle i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G} \right\rangle \longrightarrow \left\langle C; \Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G} \right\rangle$
2. $\nexists s.\, s \twoheadrightarrow x_j^{\mathsf{D}} \in \mathcal{G}$

⟨2⟩1. ASSUME: $\mathrm{WF}(\left\langle i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G} \right\rangle)$
PROVE: $\mathrm{WF}(\left\langle C; \Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G} \right\rangle)$
PROOF: Assump. ⟨1⟩8-2 ensures $\mathcal{G}$ remains well-formed. Since we match on a constraint from $C^{\mathcal{G}}$, our assumption ensures $\mathrm{dom}(\mathcal{R}) = \emptyset$, and therefore we trivially have $\forall x^{\mathsf{R}}.\, (\mathcal{R}(x^{\mathsf{R}}) = x^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x^{\mathsf{D}})$.

⟨2⟩2. PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G}$
PROOF: By Equation (†) on $\mathcal{G}$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}''$
2. $\left\langle C; \Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G} \right\rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$
3. $\mathrm{WF}(\left\langle C; \Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G} \right\rangle)$
4. $\Delta'', \mathcal{G}'' \models C$
PROVE: $\Delta'', \mathcal{G}'' \models i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}} \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'' \models i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}$
PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\Delta, \{i^{\mathsf{S}} \twoheadrightarrow x_j^{\mathsf{D}}\} \cup \mathcal{G} \subseteq \Delta'', \mathcal{G}''$
PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩3. Q.E.D.
PROOF: By ⟨3⟩2 and application of rule C-GDECL.

⟨1⟩9. CASE: Rule S-GREF
ASSUME: 1. $\langle x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}} \wedge C; \Delta, \mathcal{G} \rangle \longrightarrow \langle C; \Delta, \{x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}}\} \cup \mathcal{G} \rangle$
2. $\nexists s.\, x_i^{\mathsf{R}} \twoheadrightarrow s \in \mathcal{G}$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}} \wedge C; \Delta, \mathcal{G} \rangle)$
PROVE: $\mathrm{WF}(\langle C; \Delta, \{x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}}\} \cup \mathcal{G} \rangle)$
PROOF: Assump. ⟨1⟩9-2 ensures $\mathcal{G}$ remains well-formed. Since we match on a constraint from $C^{\mathcal{G}}$, our assumption ensures $\mathrm{dom}(\mathcal{R}) = \emptyset$, and therefore we trivially have $\forall x^{\mathsf{R}}.\, (\mathcal{R}(x^{\mathsf{R}}) = x^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x^{\mathsf{D}})$.

⟨2⟩2. PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \{x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}}\} \cup \mathcal{G}$
PROOF: By Equation (†) on $\mathcal{G}$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}''$
2. $\langle C; \Delta, \{x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}}\} \cup \mathcal{G} \rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$
3. $\mathrm{WF}(\langle C; \Delta, \{x_i^{\mathsf{R}} \twoheadrightarrow j^{\mathsf{S}}\} \cup \mathcal{G} \rangle)$

4. $\Delta'', \mathcal{G}'' \models C$

PROVE: $\Delta'', \mathcal{G}'' \models x_i^{\mathsf{R}} \relbar\!\!\blacktriangleright j^{\mathsf{S}} \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'' \models x_i^{\mathsf{R}} \relbar\!\!\blacktriangleright j^{\mathsf{S}}$

PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\Delta, \{x_i^{\mathsf{R}} \relbar\!\!\blacktriangleright j^{\mathsf{S}}\} \cup \mathcal{G} \subseteq \Delta'', \mathcal{G}''$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩3. Q.E.D.

PROOF: By ⟨3⟩2 and application of rule C-GREF.

⟨1⟩10. CASE: Rule S-GEDGE

ASSUME: 1. $\langle i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s \wedge C; \Delta, \mathcal{G} \rangle \longrightarrow \langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G} \rangle$

⟨2⟩1. ASSUME: WF($\langle i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s \wedge C; \Delta, \mathcal{G} \rangle$)

PROVE: WF($\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G} \rangle$)

PROOF: Since we match on a constraint from $C^{\mathcal{G}}$, our assumption ensures $\mathrm{dom}(\mathcal{R}) = \emptyset$, and therefore we trivially have $\forall x^{\mathsf{R}}. (\mathcal{R}(x^{\mathsf{R}}) = x^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x^{\mathsf{D}})$.

⟨2⟩2. PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G}$

PROOF: By Equation (†) on $\mathcal{G}$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}''$

2. $\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G} \rangle \longrightarrow^{*} \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$

3. WF($\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G} \rangle$)

4. $\Delta'', \mathcal{G}'' \models C$

PROVE: $\Delta'', \mathcal{G}'' \models i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'' \models i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s$

PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\blacktriangleright} s\} \cup \mathcal{G} \subseteq \Delta'', \mathcal{G}''$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩3. Q.E.D.

PROOF: By ⟨3⟩2 and application of rule C-GEDGE.

⟨1⟩11. CASE: Rule S-GNEDGE

ASSUME: 1. $\left\langle i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}} \wedge C; \Delta, \mathcal{G} \right\rangle \longrightarrow \left\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}}\} \cup \mathcal{G} \right\rangle$

2. $\exists s. x_j^{\mathsf{R}} \relbar\!\!\blacktriangleright s \in \mathcal{G}$

⟨2⟩1. ASSUME: WF($\left\langle i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}} \wedge C; \Delta, \mathcal{G} \right\rangle$)

PROVE: WF($\left\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}}\} \cup \mathcal{G} \right\rangle$)

PROOF: Assump. ⟨1⟩11-2 ensures $\mathcal{G}$ remains well-formed. Since we match on a constraint from $C^{\mathcal{G}}$, our assumption ensures $\mathrm{dom}(\mathcal{R}) = \emptyset$, and therefore we trivially have $\forall x^{\mathsf{R}}. (\mathcal{R}(x^{\mathsf{R}}) = x^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x^{\mathsf{D}})$.

⟨2⟩2. PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}}\} \cup \mathcal{G}$

PROOF: By Equation (†) on $\mathcal{G}$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}''$

2. $\left\langle C; \Delta, \{i^{\mathsf{S}} \overset{l}{\relbar\!\!\rhd} x_j^{\mathsf{R}}\} \cup \mathcal{G} \right\rangle \longrightarrow^{*} \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$

3. $\mathrm{WF}(\Big\langle C;\Delta,\{i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}}\} \cup \mathcal{G}\Big\rangle)$
4. $\Delta'',\mathcal{G}'' \models C$

Prove: $\Delta'',\mathcal{G}'' \models i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}} \wedge C$

⟨3⟩1. Suffices: $\Delta'',\mathcal{G}'' \models i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}}$
Proof: By application of rule C-Conj, given that $\Delta'',\mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\Delta,\{i^{\mathsf{S}} \overset{l}{\mathrel{-\!\!\triangleright}} x_j^{\mathsf{R}}\} \cup \mathcal{G} \subseteq \Delta'',\mathcal{G}''$
Proof: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩3. Q.E.D.
Proof: By ⟨3⟩2 and application of rule C-GNEdge.

⟨1⟩12. Case: Rule S-GAssoc
Assume: 1. $\langle x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow \langle C;\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle$
2. $\nexists s.\, x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} s \in \mathcal{G}$

⟨2⟩1. Assume: $\mathrm{WF}(\langle x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\rangle)$
Prove: $\mathrm{WF}(\langle C;\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle)$
Proof: Assump. ⟨1⟩12-2 ensures $\mathcal{G}$ remains well-formed. Since we match on a constraint from $C^{\mathcal{G}}$, our assumption ensures $\mathrm{dom}(\mathcal{R}) = \emptyset$, and therefore we trivially have $\forall x^{\mathsf{R}}.\, (\mathcal{R}(x^{\mathsf{R}}) = x^{\mathsf{D}} \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x^{\mathsf{D}})$.

⟨2⟩2. Prove: $\Delta,\mathcal{G} \subseteq \Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}$
Proof: By Equation (†) on $\mathcal{G}$.

⟨2⟩3. Assume: 1. Any $\Delta'',\mathcal{G}''$
2. $\langle C;\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta'',\mathcal{G}''\rangle$
3. $\mathrm{WF}(\langle C;\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G}\rangle)$
4. $\Delta'',\mathcal{G}'' \models C$

Prove: $\Delta'',\mathcal{G}'' \models x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \wedge C$

⟨3⟩1. Suffices: $\Delta'',\mathcal{G}'' \models x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}$
Proof: By application of rule C-Conj, given that $\Delta'',\mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\Delta,\{x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}}\} \cup \mathcal{G} \subseteq \Delta'',\mathcal{G}''$
Proof: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩3. Q.E.D.
Proof: By ⟨3⟩2 and application of rule C-GAssoc.

⟨1⟩13. Case: Rule S-Assoc
Assume: 1. $\langle x_i^{\mathsf{D}} \rightsquigarrow s \wedge C;\Delta,\mathcal{G}\rangle \longrightarrow \big\langle s \overset{?}{=} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\big\rangle$
2. $x_i^{\mathsf{D}} \mathrel{-\!\!\triangleright} j^{\mathsf{S}} \in \mathcal{G}$

⟨2⟩1. Assume: $\mathrm{WF}(\langle x_i^{\mathsf{D}} \rightsquigarrow s \wedge C;\Delta,\mathcal{G}\rangle)$
Prove: $\mathrm{WF}(\big\langle s \overset{?}{=} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\big\rangle)$
Proof: Trivial.

⟨2⟩2. Prove: $\Delta,\mathcal{G} \subseteq \Delta,\mathcal{G}$
Proof: By reflexivity of $\subseteq$.

⟨2⟩3. Assume: 1. Any $\Delta'',\mathcal{G}''$
2. $\big\langle s \overset{?}{=} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\big\rangle \longrightarrow^{*} \langle \mathsf{True};\Delta'',\mathcal{G}''\rangle$
3. $\mathrm{WF}(\big\langle s \overset{?}{=} j^{\mathsf{S}} \wedge C;\Delta,\mathcal{G}\big\rangle)$

4. $\Delta'', \mathcal{G}'' \models s \overset{?}{=} j^{\mathsf{S}} \wedge C$

PROVE: $\Delta'', \mathcal{G}'' \models x_i^{\mathsf{D}} \rightsquigarrow s \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'' \models x_i^{\mathsf{D}} \rightsquigarrow s$

PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $x_i^{\mathsf{D}} \longrightarrow\!\!\triangleright\, j^{\mathsf{S}} \in \mathcal{G}''$

PROOF: By assump. ⟨1⟩13-2 and assump. ⟨2⟩3-3, we have $\exists \sigma . \mathcal{G}\sigma \subseteq \mathcal{G}''$. Since $x_i^{\mathsf{D}} \longrightarrow\!\!\triangleright\, j^{\mathsf{S}}$ is ground, it is also in $\mathcal{G}''$.

⟨3⟩3. $s\varphi'' = j^{\mathsf{S}}$

PROOF: By assump. ⟨2⟩3-4 and inversion of rule C-EQUAL, we have $s\varphi'' = j^{\mathsf{S}}\varphi''$. We conclude by observing that, since j^{S} is ground, $j^{\mathsf{S}}\varphi'' = j^{\mathsf{S}}$.

⟨3⟩4. Q.E.D.

PROOF: By rule C-ASSOC.

⟨1⟩14. CASE: Rule S-RESOLVE1

ASSUME: 1. $\langle x_i^{\mathsf{R}} \mapsto d \wedge C; \Delta, \mathcal{G}, \mathcal{R} \rangle \longrightarrow \left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R} \right\rangle$

2. $\mathcal{R}(x_i^{\mathsf{R}}) = \bot$

3. $C \cap C^{\mathcal{G}} = \emptyset$

4. $\mathrm{RES}_{\mathcal{G}}(x_i^{\mathsf{R}}) = D \wedge D \neq \mathsf{U} \wedge x_j^{\mathsf{D}} \in D$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle x_i^{\mathsf{R}} \mapsto d \wedge C; \Delta, \mathcal{G}, \mathcal{R} \rangle)$

PROVE: $\mathrm{WF}(\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R} \right\rangle)$

PROOF: The extension of $\mathcal{R}$ requires $\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}$, which we have by assump. ⟨1⟩14-4 and $x_i^{\mathsf{D}} \in \mathrm{RES}_{\mathcal{G}}(x^{\mathsf{R}}) \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \mapsto x_i^{\mathsf{D}}$ (van Antwerpen et al., 2016, Lemma 2). Assump. ⟨1⟩14-3 discharges the well-formedness condition $C \cap C^{\mathcal{G}} = \emptyset \implies \mathrm{dom}(\mathcal{R}) = \emptyset$.

⟨2⟩2. PROVE: $\Delta, \mathcal{G}, \mathcal{R} \subseteq \Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R}$

PROOF: By Equation (†) on $\mathcal{R}$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}'', \mathcal{R}''$

2. $\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R} \right\rangle \longrightarrow^{*} \langle \mathsf{True}; \Delta'', \mathcal{G}'', \mathcal{R}'' \rangle$

3. $\mathrm{WF}(\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R} \right\rangle)$

4. $\Delta'', \mathcal{G}'', \mathcal{R}'' \models d \overset{?}{=} x_j^{\mathsf{D}} \wedge C$

PROVE: $\Delta'', \mathcal{G}'', \mathcal{R}'' \models x_i^{\mathsf{R}} \mapsto d \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'', \mathcal{R}'' \models x_i^{\mathsf{R}} \mapsto d$

PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'', \mathcal{R}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\mathrm{WF}(\langle \mathsf{True}; \Delta'', \mathcal{G}'', \mathcal{R}'' \rangle)$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.8.

⟨3⟩3. $\Delta, \mathcal{G}, \{(x_i^{\mathsf{R}}, x_j^{\mathsf{D}})\} \cup \mathcal{R} \subseteq \Delta'', \mathcal{G}'', \mathcal{R}''$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩4. $\mathcal{R}''(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}}$

PROOF: Given ⟨3⟩3 and the fact that $\mathcal{R}$ is ground, we have $\mathcal{R}''(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}}$.

⟨3⟩5. $\vdash_{\mathcal{G}''} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}$

PROOF: By ⟨3⟩2 and ⟨3⟩4.

⟨3⟩6. $d\varphi'' = x_j^{\mathsf{D}}$

PROOF: By assump. ⟨2⟩3-4 and inversion of rule C-EQUAL, we have $d\varphi'' = x_j^{\mathsf{D}}\varphi''$. We conclude by observing that, since x_j^{D} is ground, $x_j^{\mathsf{D}}\varphi'' = x_j^{\mathsf{D}}$.

⟨3⟩7. Q.E.D.

PROOF: By ⟨3⟩4, ⟨3⟩5, ⟨3⟩6 and rule C-RESOLVE.

⟨1⟩15. CASE: Rule S-RESOLVEN

ASSUME: 1. $\langle x_i^{\mathsf{R}} \mapsto d \wedge C; \Delta, \mathcal{G}, \mathcal{R}\rangle \longrightarrow \left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \mathcal{R}\right\rangle$

2. $\mathcal{R}(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}}$

⟨2⟩1. ASSUME: $\mathrm{WF}(\langle x_i^{\mathsf{R}} \mapsto d \wedge C; \Delta, \mathcal{G}, \mathcal{R}\rangle)$

PROVE: $\mathrm{WF}(\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \mathcal{R}\right\rangle)$

PROOF: Trivial.

⟨2⟩2. PROVE: $\Delta, \mathcal{G}, \mathcal{R} \subseteq \Delta, \mathcal{G}, \mathcal{R}$

PROOF: By reflexivity of $\subseteq$.

⟨2⟩3. ASSUME: 1. Any $\Delta'', \mathcal{G}'', \mathcal{R}''$

2. $\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \mathcal{R}\right\rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \mathcal{G}'', \mathcal{R}''\rangle$

3. $\mathrm{WF}(\left\langle d \overset{?}{=} x_j^{\mathsf{D}} \wedge C; \Delta, \mathcal{G}, \mathcal{R}\right\rangle)$

4. $\Delta'', \mathcal{G}'', \mathcal{R}'' \models d \overset{?}{=} x_j^{\mathsf{D}} \wedge C$

PROVE: $\Delta'', \mathcal{G}'', \mathcal{R}'' \models x_i^{\mathsf{R}} \mapsto d \wedge C$

⟨3⟩1. SUFFICES: $\Delta'', \mathcal{G}'', \mathcal{R}'' \models x_i^{\mathsf{R}} \mapsto d$

PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'', \mathcal{R}'' \models C$ follows from assump. ⟨2⟩3-4.

⟨3⟩2. $\mathrm{WF}(\langle \mathsf{True}; \Delta'', \mathcal{G}'', \mathcal{R}''\rangle)$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.8.

⟨3⟩3. $\Delta, \mathcal{G}, \mathcal{R} \subseteq \Delta'', \mathcal{G}'', \mathcal{R}''$

PROOF: By assump. ⟨2⟩3-2, and Corollary 4.9.

⟨3⟩4. $\mathcal{R}''(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}}$

PROOF: Given assump. ⟨1⟩15-2, ⟨3⟩3 and the fact that $\mathcal{R}$ is ground, we have $\mathcal{R}''(x_i^{\mathsf{R}}) = x_j^{\mathsf{D}}$.

⟨3⟩5. $\vdash_{\mathcal{G}''} x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}$

PROOF: By ⟨3⟩2 and ⟨3⟩4.

⟨3⟩6. $d\varphi'' = x_j^{\mathsf{D}}$

PROOF: By assump. ⟨2⟩3-4 and inversion of rule C-EQUAL, we have $d\varphi'' = x_j^{\mathsf{D}}\varphi''$. We conclude by observing that, since x_j^{D} is ground, $x_j^{\mathsf{D}}\varphi'' = x_j^{\mathsf{D}}$.

⟨3⟩7. Q.E.D.

PROOF: By ⟨3⟩4, ⟨3⟩5, ⟨3⟩6 and rule C-RESOLVE.

⟨1⟩16. CASE: Rule S-DISTINCT

ASSUME: 1. $\langle !N \wedge C; \Delta, \mathcal{G}\rangle \longrightarrow \langle C; \Delta, \mathcal{G}\rangle$

2. $\mathrm{vars}(N) = \emptyset$

3. $C \cap C^{\mathcal{G}} = \emptyset$
4. $\text{Ns}_{\mathcal{G}}(N) = X \wedge X \neq \bot$
5. $\forall x \in X. v(x) = 1$

$\langle 2 \rangle 1.$ ASSUME: $\text{WF}(\langle !N \wedge C; \Delta, \mathcal{G} \rangle)$
PROVE: $\text{WF}(\langle C; \Delta, \mathcal{G} \rangle)$
PROOF: Trivial.

$\langle 2 \rangle 2.$ PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \mathcal{G}$
PROOF: By reflexivity of $\subseteq$.

$\langle 2 \rangle 3.$ ASSUME: 1. Any $\Delta'', \mathcal{G}''$
2. $\langle C; \Delta, \mathcal{G} \rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$
3. $\text{WF}(\langle C; \Delta, \mathcal{G} \rangle)$
4. $\Delta'', \mathcal{G}'' \models C$

PROVE: $\Delta'', \mathcal{G}'' \models !N \wedge C$

$\langle 3 \rangle 1.$ SUFFICES: $\Delta'', \mathcal{G}'' \models !N$
PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. $\langle 2 \rangle$3-4.

$\langle 3 \rangle 2.$ $\exists \sigma. \mathcal{G}'' = \mathcal{G}\sigma$
PROOF: By assump. $\langle 1 \rangle$16-3, assump. $\langle 2 \rangle$3-2 and Lemma A.1.

$\langle 3 \rangle 3.$ $[\![N]\!]_{\mathcal{G}''} = X$
PROOF: We know that $\text{Ns}_{\mathcal{G}}(N) = X \implies \forall \sigma. [\![N]\!]_{\mathcal{G}\sigma} = X$ (van Antwerpen et al., 2016, Lemma 3). We conclude by assump. $\langle 1 \rangle$16-4 and $\langle 3 \rangle 2$.

$\langle 3 \rangle 4.$ Q.E.D.
PROOF: By $\langle 3 \rangle 3$, assump. $\langle 1 \rangle$16-5 and rule C-DISTINCT.

$\langle 1 \rangle 17.$ CASE: Rule S-SUBSET
ASSUME: 1. $\langle N_1 \subsetneqq N_2 \wedge C; \Delta, \mathcal{G} \rangle \longrightarrow \langle C; \Delta, \mathcal{G} \rangle$
2. $\text{vars}(N_1) \cup \text{vars}(N_2) = \emptyset$
3. $C \cap C^{\mathcal{G}} = \emptyset$
4. $\text{Ns}_{\mathcal{G}}(N_1) = X_1 \wedge X_1 \neq \bot$
5. $\text{Ns}_{\mathcal{G}}(N_2) = X \wedge X_2 \neq \bot$
6. $X_1 \subseteq X_2$

$\langle 2 \rangle 1.$ ASSUME: $\text{WF}(\langle N_1 \subsetneqq N_2 \wedge C; \Delta, \mathcal{G} \rangle)$
PROVE: $\text{WF}(\langle C; \Delta, \mathcal{G} \rangle)$
PROOF: Trivial.

$\langle 2 \rangle 2.$ PROVE: $\Delta, \mathcal{G} \subseteq \Delta, \mathcal{G}$
PROOF: By reflexivity of $\subseteq$.

$\langle 2 \rangle 3.$ ASSUME: 1. Any $\Delta'', \mathcal{G}''$
2. $\langle C; \Delta, \mathcal{G} \rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', \mathcal{G}'' \rangle$
3. $\text{WF}(\langle C; \Delta, \mathcal{G} \rangle)$
4. $\Delta'', \mathcal{G}'' \models C$

PROVE: $\Delta'', \mathcal{G}'' \models N_1 \subsetneqq N_2 \wedge C$

$\langle 3 \rangle 1.$ SUFFICES: $\Delta'', \mathcal{G}'' \models N_1 \subsetneqq N_2$
PROOF: By application of rule C-CONJ, given that $\Delta'', \mathcal{G}'' \models C$ follows from assump. $\langle 2 \rangle$3-4.

$\langle 3 \rangle 2.$ $\exists \sigma. \mathcal{G}'' = \mathcal{G}\sigma$

PROOF: By assump. $\langle 1\rangle$16-3, assump. $\langle 2\rangle$3-2 and Lemma A.1.

$\langle 3\rangle$3. $[\![N_i]\!]_{\mathcal{G}''} = X_i$ for $i \in \{1,2\}$

PROOF: We know that $\mathrm{NS}_{\mathcal{G}}(N) = X \implies \forall\sigma.\, [\![N]\!]_{\mathcal{G}\sigma} = X$ (van Antwerpen et al., 2016, Lemma 3). We conclude by assump. $\langle 1\rangle$16-4 or assump. $\langle 1\rangle$16-5, and $\langle 3\rangle$2.

$\langle 3\rangle$4. Q.E.D.

PROOF: By $\langle 3\rangle$3, assump. $\langle 1\rangle$16-6 and rule C-SUBSET.

$\langle 1\rangle$18. CASE: Rule S-SUPERTYPE

ASSUME: 1. $\langle t_1 <: t_2 \wedge C;\Delta,<_T\rangle \longrightarrow \langle C;\Delta,\{(t,t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T\rangle$

2. $\mathrm{vars}(t_1) \cup \mathrm{vars}(t_2) = \emptyset$

3. $t_1 \notin \mathrm{dom}(<_T)$

4. $t_2 \not\leq_T t_1$

$\langle 2\rangle$1. ASSUME: $\mathrm{WF}(\langle t_1 <: t_2 \wedge C;\Delta,<_T\rangle)$

PROVE: $\mathrm{WF}(\langle C;\Delta,\{(t,t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T\rangle)$

$\langle 3\rangle$1. LET: $T_1 = \{t \mid t \leq_T t_1\}$

$T_2 = \{t \mid t_2 \leq_T t\}$

$<'_T = \{(t,t') \mid t \in T_1 \wedge t' \in T_2\} \cup <_T$

$\langle 3\rangle$2. $\forall t \in T_1, t' \in T_2.\, (t \not\leq_T t' \wedge t' \not\leq_T t)$

PROOF: We show by contradiction. Assume there is a $t \in T_1$, $t \in T_2$ such that $t' \leq_T t$. By definition, we have $t \leq_T t_1$ and $t_2 \leq_T t'$. Transitivity of $<_T$ implies $t_2 \leq t_1$, but this contradicts assump. $\langle 1\rangle$18-4. Conversely, if we assume $t \leq_T t'$, we must have $t' \in T_1$ by transitivity, and therefore $t' \leq_T t_1$. This would also contradict assump. $\langle 1\rangle$18-4.

$\langle 3\rangle$3. $\forall t,t'.\, (t <'_T t' \implies t' \not\leq'_T t)$

PROOF: We distinguish two cases, based on whether the inequality appears in the original subtyping relation $<_T$:

1. $t <_T t'$: We know by anti-symmetry of $<_T$, that $t' \not\leq_T t$. Now assume we would have $t' <'_T t$. This would contradict $\langle 3\rangle$2, since we have $t <_T t'$ in the original relation. So the new relation will not violate anti-symmetry for existing pairs in the relation.
2. $t \not<_T t'$: We know by $\langle 3\rangle$2 that T_1 and T_2 are disjoint and are not related in $<_T$. By construction of $<'_T$ types from T_1 only appear as smaller than types from T_2.

$\langle 3\rangle$4. $\forall t,t',t''.\, (t <'_T t' \wedge t' <'_T t'' \implies t <'_T t'')$

We distinguish four cases, based on whether the inequalities are in the original subtyping relation $<_T$:

1. $t <_T t' \wedge t' <_T t''$: We conclude by transitivity of $<_T$.
2. $t \not<_T t' \wedge t' \not<_T t''$: An equality that does not appear in the original $<_T$, must be an inequality between a type from T_1 and a type from T_2. Since t' appears on both the left and the right of the inequality, it should be in T_1 and T_2. This contradicts $\langle 3\rangle$2, so this case does not occur.
3. $t <_T t' \wedge t' \not<_T t''$: By similar reasoning as case (2), we

must have $t' \in T_1$ and $t'' \in T_2$. By transitivity we have $t \in T_1$. Therefore $t <'_T t''$ by construction of $<'_T$.

4. $t \not<_T t' \wedge t' <_T t''$: By similar reasoning as case (2), we must have $t \in T_1$ and $t' \in T_2$. By transitivity we have $t'' \in T_2$. Therefore $t <'_T t''$ by construction of $<'_T$.

$\langle 3 \rangle 5.$ $\forall t. (\{t' \mid t <'_T t'\}$ is totally ordered by $<'_T)$
PROOF: We distinguish two cases:
1. $t \not<'_T t_1$: Since we only added pairs $t <'_T t'$ for $t \leq_T t_1$, the set of larger types for any $t \not\leq_T t_1$ is the same as in $<_T$, and therefore totally ordered.
2. $t <'_T t_1$: By assump. $\langle 1 \rangle 18$-3 and the total ordering of $<_T$, we know that t_1 is the largest type in $T = \{t' \mid t <_T t'\}$. Therefore, the set $T' = \{t' \mid t <'_T t'\} = T \cup T_2$. Let $t', t'' \in T'$. We distinguish two cases:
   a. t' and t'' are both in T, or in T_2, in which case the pair is ordered in $<_T$.
   b. $t' \in T$ and $t'' \in T_2$, in which case the construction of $<'_T$ ensures the pair is ordered.

$\langle 3 \rangle 6.$ Q.E.D.
PROOF: By $\langle 3 \rangle 4$, $\langle 3 \rangle 3$, and $\langle 3 \rangle 5$.

$\langle 2 \rangle 2.$ PROVE: $\Delta, <_T \subseteq \Delta, \{(t, t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T$
PROOF: By Equation (†) on $<_T$.

$\langle 2 \rangle 3.$ ASSUME: 1. Any $\Delta'', <''_T$
2. $\langle C; \Delta, \{(t, t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T \rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', <''_T \rangle$
3. $\mathrm{WF}(\langle C; \Delta, \{(t, t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T \rangle)$
4. $\Delta'', <''_T \models C$

PROVE: $\Delta'', <''_T \models t_1 <: t_2 \wedge C$

$\langle 3 \rangle 1.$ SUFFICES: $\Delta'', <''_T \models t_1 <: t_2$
PROOF: By application of rule C-CONJ, given that $\Delta'', <''_T \models C$ follows from assump. $\langle 2 \rangle 3$-4.

$\langle 3 \rangle 2.$ $\Delta, \{(t, t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T \subseteq \Delta'', <''_T$
PROOF: By assump. $\langle 2 \rangle 3$-2, and Corollary 4.9.

$\langle 3 \rangle 3.$ LET: $<'_T$ be $\{(t, t') \mid t \leq_T t_1 \wedge t_2 \leq_T t'\} \cup <_T$

$\langle 3 \rangle 4.$ $t_1 \varphi'' <''_T t_2 \varphi''$
PROOF: Since t_1 and t_2 are ground, it suffices to show $t_1 <''_T t_2$. By $\langle 3 \rangle 2$, given that $t_1 <'_T t_2$ in the construction of $<'_T$.

$\langle 3 \rangle 5.$ $\forall t. (t_1 \varphi'' <''_T t \implies t_2 \varphi'' \leq''_T t)$
PROOF: Since t_1 and t_2 our ground, it suffices to show that $\forall t. (t_1 <''_T t \implies t_2 \leq''_T t)$ holds. From the definition of $<'_T$, it follows that $\forall t. (t_1 <'_T t \implies t_2 \leq'_T t)$. In the rest of the reduction, assump. $\langle 1 \rangle 18$-3 ensures that no new supertypes of t_1 are added that are not also supertypes of t_2, therefore $\forall t. (t_1 <''_T t \implies t_2 \leq''_T t)$.

$\langle 3 \rangle 6.$ Q.E.D.
PROOF: By $\langle 3 \rangle 4$, $\langle 3 \rangle 5$, and rule C-SUPERTYPE.

$\langle 1\rangle$19. CASE: Rule S-SUBTYPE
ASSUME: 1. $\langle t_1 \leq: t_2 \wedge C; \Delta, <_T\rangle \longrightarrow \langle C; \Delta, <_T\rangle$
2. $\mathrm{vars}(t_1) \cup \mathrm{vars}(t_2) = \emptyset$
3. $t_1 \leq_T t_2$
$\langle 2\rangle$1. ASSUME: $\mathrm{WF}(\langle t_1 \leq: t_2 \wedge C; \Delta, <_T\rangle)$
PROVE: $\mathrm{WF}(\langle C; \Delta, <_T\rangle)$
PROOF: Trivial.
$\langle 2\rangle$2. PROVE: $\Delta, <_T \subseteq \Delta, <_T$
PROOF: By reflexivity of $\subseteq$.
$\langle 2\rangle$3. ASSUME: 1. Any $\Delta'', <''_T$
2. $\langle C; \Delta, <_T\rangle \longrightarrow^* \langle \mathsf{True}; \Delta'', <''_T\rangle$
3. $\mathrm{WF}(\langle C; \Delta, <_T\rangle)$
4. $\Delta'', <''_T \models C$
PROVE: $\Delta'', <''_T \models t_1 \leq: t_2 \wedge C$
$\langle 3\rangle$1. SUFFICES: $\Delta'', <''_T \models t_1 \leq: t_2$
PROOF: By application of rule C-CONJ, given that $\Delta'', <''_T \models C$ follows from assump. $\langle 2\rangle$3-4.
$\langle 3\rangle$2. SUFFICES: $t_1 \leq''_T t_2$
PROOF: By inversion of rule C-SUBTYPE and that fact that t_1, t_2 are ground, so $t_i\varphi'' = t_i$.
$\langle 3\rangle$3. $\Delta, <_T \subseteq \Delta'', <''_T$
PROOF: By assump. $\langle 2\rangle$3-2, and Corollary 4.9.
$\langle 3\rangle$4. Q.E.D.
PROOF: By assump. $\langle 1\rangle$19-3 and $\langle 3\rangle$3.

$\langle 1\rangle$20. Q.E.D.
PROOF: By the principle of exhaustion. □

A.2 Reduction Termination

Recall the lemma for termination of multi-step reductions:

LEMMA 4.12 (Termination of $\longrightarrow$). $\forall C\Delta.\,(\langle C; \Delta\rangle \longrightarrow^* \ldots$ *terminates*).

Proof. Take the lexicographical tuple on natural numbers (n_1, n_2, n_3, n_4), where
- n_1 is the number of non-equality constraints,
- n_2 is the number of distinct variables in eq. constraints,
- n_3 is the number of symbols in eq. constraints,
- n_4 is the number of equality constraints of the form $t \overset{?}{=} \alpha$, where $t \notin Var$.

By case analysis on the reduction rules, we show that this tuple decreases for every reduction rule.

$\langle 1\rangle$1. CASE: S-TRUE
PROOF: The removal of the constraint True decreases n_1 by 1.

$\langle 1\rangle$2. CASE: S-TRIVIAL
PROOF: The removal of the constraint $t \overset{?}{=} t$ decreases n_3 by two times the number of symbols in t.

⟨1⟩3. CASE: S-ORIENT
PROOF: The removal of the constraint $t \overset{?}{=} \alpha$ decreases n_4 by 1. The condition $t \notin \mathit{Var}$ ensures that the newly introduced constraint does not count towards n_4.

⟨1⟩4. CASE: S-ELIMINATE
PROOF: The application of the substitution $\alpha \mapsto t$, together with the condition that $\alpha \notin \mathrm{vars}(t)$ ensures that α is eliminated, decreasing n_2 by 1.

⟨1⟩5. CASE: S-DECOMPOSE
PROOF: The elimination of the two f symbols decreases n_3 decreases by 2. The newly introduced constraints may increase n_4.

⟨1⟩6. CASE: S-TYPEOF1
PROOF: The removal of the constraint $x^{\mathsf{D}}_i : t$ decreases n_1 by 1.

⟨1⟩7. CASE: S-TYPEOFN
PROOF: The removal of the constraint $x^{\mathsf{D}}_i : t$ decreases n_1 by 1. The introduced equality constraint may increase n_2, n_3 and n_4.

⟨1⟩8. CASE: S-GDECL
PROOF: The removal of the constraint $j \mathrel{-\!\!\blacktriangleright} x_i$ decreases n_1 by 1.

⟨1⟩9. CASE: S-GREF
PROOF: The removal of the constraint $x_i \mathrel{-\!\!\blacktriangleright} j$ decreases n_1 by 1.

⟨1⟩10. CASE: S-GNEDGE
PROOF: The removal of the constraint $j \overset{l}{\mathrel{-\!\!\triangleright}} x_i$ decreases n_1 by 1.

⟨1⟩11. CASE: S-GEDGE
PROOF: The removal of the constraint $j \overset{l}{\mathrel{-\!\!\blacktriangleright}} s$ decreases n_1 by 1.

⟨1⟩12. CASE: S-GASSOC
PROOF: The removal of the constraint $x_i \mathrel{-\!\!\triangleright} j$ decreases n_1 by 1.

⟨1⟩13. CASE: S-ASSOC
PROOF: The removal of the constraint $x^{\mathsf{D}}_i \rightsquigarrow s$ decreases n_1 by 1. The introduced equality constraint may increase n_2, n_3 and n_4.

⟨1⟩14. CASE: S-RESOLVE1
PROOF: The removal of the constraint $x^{\mathsf{R}}_i \mapsto d$ decreases n_1 by 1. The introduced equality constraint may increase n_2, n_3 and n_4. The resolution algorithm terminates (van Antwerpen et al., 2016, Section 4.3).

⟨1⟩15. CASE: S-RESOLVEN
PROOF: The removal of the constraint $x^{\mathsf{R}}_i \mapsto d$ decreases n_1 by 1. The introduced equality constraint may increase n_2, n_3 and n_4.

⟨1⟩16. CASE: S-DISTINCT
PROOF: The removal of the constraint $!N$ decreases n_1 by 1. Calculating the name set terminates, because the sets of references and declarations are finite, and the resolution algorithm terminates.

⟨1⟩17. CASE: S-SUBSET
PROOF: The removal of the constraint $N_1 \underset{\sim}{\subset} N_2$ decreases n_1 by 1. Cal-

culating the name sets terminates, because the sets of references and declarations are finite, and the resolution algorithm terminates.

$\langle 1 \rangle$18. CASE: S-SUPERTYPE
PROOF: The removal of the constraint $t_1 <: t_2$ decreases n_1 by 1. Building the subtyping relation terminates, since $<_T$ is finite.

$\langle 1 \rangle$19. CASE: S-SUBTYPE
PROOF: The removal of the constraint $t_1 \leq: t_2$ decreases n_1 by 1. Checking if the types are in the subtyping relation terminates, since $<_T$ is finite.

□