Scopes as Types

(under submission; feedback welcome)

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
CASPER BACH POULSEN, Delft University of Technology, Netherlands
EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework for modeling the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

1 INTRODUCTION

Language workbenches aim to support easy construction of programming tools such as language-aware editors, type checkers, and compilers from high-level language definitions [Erdweg et al. 2015]. A longer term goal in the development of language workbenches is to integrate support for mechanized meta theory in order to (automatically) verify properties of language definitions [Visser et al. 2014]. That requires meta-languages for language definitions that are amenable to verification as well as implementation. Scope graphs were designed to provide a central component in such a framework.

Scope graphs were introduced by Néron et al. [2015a] as a general model for name resolution in programming languages that is suitable for formalization as well as implementation. A scope graph captures the binding structure of a program. A generic, language-independent resolution algorithm interprets a scope graph to resolve references to declarations of names. Thus, to express the binding rules of a programming language, one defines the mapping from abstract syntax tree to scope graph. A generic resolution engine takes care of resolving names. Scope graphs cover a wide range of binding structures, including *lexical binding structures* such as variations of let bindings, function parameters, and local variables in blocks, and *non-lexical binding structures* such as (cyclic) module imports and class inheritance. The framework allows language-independent definition of operations such as alpha equivalence and safe renaming of programs.

Van Antwerpen et al. [2016] embed scope graphs in a constraint language. The static semantics of a programming language is defined as a mapping from abstract syntax trees to a set of name and type constraints, which are solved by a language-independent constraint solver. The combination with type constraints allows declarative definition of *type-dependent name resolution* in cases where type analysis and name resolution are mutually dependent, such as in resolving a field access to an object instantiating a record or class.

Authors' addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

In addition to serving as a model for name resolution, scope graphs provide a language-independent model for run-time memory. Poulsen et al. [2016] define a framework in which *scopes describe frames*. A frame is a run-time unit of memory allocation that corresponds one-to-one to a scope in a scope graph. This provides the basis for language-independent invariants about the well-typedness of memory (the store), which are used by Poulsen et al. [2018] in a technique for automating the type safety proofs of definitional interpreters by incorporating object-language types and bindings in the types of abstract syntax trees.

Thus, scope graphs are the basis for a promising approach to the definition of the static semantics of programming languages that serves the implementation of tools such as type checkers, as well as the verification of language properties such as type safety. However, the adoption of scope graphs is inhibited by its limitation to simple type systems. As a model that ties information to *names*, scope graphs appear to be limited in expressiveness. The works above cover languages with simple, nominal type systems, and their future work calls for extension to more sophisticated type systems. In particular, it is not clear how scope graphs can be used to describe *structural types*, in which types are not identified by names, and *generic types*, in which types are parameterized by types.

In this paper, we demonstrate how scope graphs can be used to model type systems with more sophisticated forms of type equality, such as for structural and parametric types, by using *scopes as types*. Scopes provide a uniform representation for types with rich structure that cannot easily be captured in syntactic terms. Similarly to how relations over syntactic terms are defined in terms of pattern matching, do we define relations over these rich scope types using resolution queries. Furthermore, we present *Statix*, a new constraint-based language for static semantics specification based on this approach. We make the following technical contributions:

- We show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries.
- We extend the scope graph model with *scoped relations* to model the association of types with declarations and explicit substitutions in the instantiation of parameterized types. We generalize name resolution in scope graph from resolution from references to declarations to general *queries* for scoped relations. This enables flexible definition of queries for reachable or visible declarations and other properties, such as the visible record fields in the definition of subtyping of structural record types.
- We extend the *visual notation* of scope graph diagrams with scoped relations, which provides a useful language for explaining patterns of names and types in programming languages. We also extend the visual notation with unresolved (constraint) nodes for illustrating the resolution process.
- We introduce *Statix*, a declarative, language for specifying type systems. The language provides simple guarded rules for definition of user-defined constraints with unification and scope graph construction and resolution as built-in theories. We provide a declarative and an operational semantics of Statix.
- We *simplify* the resolution calculus and algorithm of Néron et al. [2015a] and Van Antwerpen et al. [2016] by not including imports as a primitive. We demonstrate how imports (and other name- and type-dependent name resolution schemas) can be encoded using the scopes as types approach. We discuss how these patterns depend on *resolution in incomplete scope graphs*, and how the algorithm guarantees soundness of resolution in incomplete graphs. We further generalize resolution by namespace/query-specific parameterization with visibility policies instead of global policies.

- We have evaluated the Statix language in three case studies: the simply-typed lambda calculus with records [Pierce 2002] (STLC-REC), System F [Girard 1972; Reynolds 1974], and Featherweight Generic Java [Igarashi et al. 2001].

Outline. We proceed as follows. In the next section we review scope graphs, extend scope graphs with scoped relations, and illustrate the concepts using lexical bindings, modules with imports, and type-dependent name resolution in record types. In Section 3 we demonstrate that using scopes as types, we can model sophisticated type systems with structural types and parameterized types. In Section 4 we introduce the Statix language by means of examples. In Section 5 we discuss geneneralized and simplified name resolution calculus and algorithm. In Section 6 we define a the syntax and semantics of Statix. We discuss related work in Section 7. In the appendices we provide further technical discussion, including the operational semantics of Statix (Appendix B), and the specifications in Statix of STLC-REC, System F, and FGJ.

2 SCOPES WITH TYPES

In this section we revisit scope graphs as introduced by Néron et al. [2015a] and Van Antwerpen et al. [2016] and illustrate their application to modeling simple nominal type systems. We extend scope graphs with scoped relations to include type annotations *within* scope graph models. We end with an analysis of the limitations of scope graphs to model more sophisticated type systems. In this section and the next we use the visual notation of scope graph diagrams to illustrate the concepts. In Section 5 and Section 6 we give a formal account.

2.1 Scope Graphs

Fig. 1 shows an example program with lexical binding and its corresponding scope graph. The program consists of a sequential let, binding identifiers x_1 and f_2, which are used in a function application. To distinguish different occurrences of the same name, names in programs are subscripted with their position in the program. Thus, x_1, x_3, x_4, and x_6 are different occurrences of the same name x.

```
let x1 = 14
    f2 = fun (x3: Int) { x4 + 1 }
in f5 x6
```

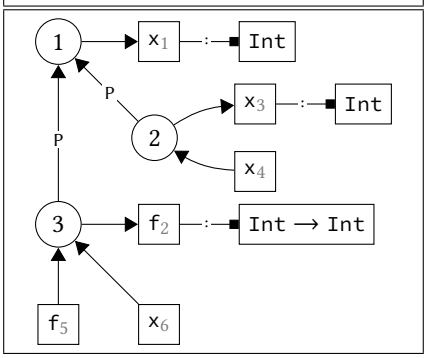


Fig. 1. Scope graph modeling lexical scoping of variables with types.

A *scope graph* is a directed graph, consisting of scopes, references and declarations, connected by labeled edges. A *scope* corresponds to a region in the program that behaves uniformly with respect to name resolution. Scopes are depicted by round nodes, labeled with a number, such as (1). For example, scope (3) is the scope of the body of the let and scope (2) is the scope of the function.

An *edge* between two scopes is depicted as (2) $\xrightarrow{P}$ (1) and denotes that scope (1) is *reachable* from scope (2). The edge label $\xrightarrow{P}$ denotes that scope (1) is a *lexical parent* of scope (2). Labels are used to control *visibility* of (declarations in) scopes. We illustrate visibility policy examples later.

An *occurrence* x_i corresponds to a name x occurring at position i in a program. Occurrences are depicted by rectangular nodes in the graph, labeled with the name and position, such as $\boxed{x_i}$.

A *declaration* is an occurrence that introduces a name. A declaration is depicted using an arrow from a scope to an occurrence, such as $(i) \longrightarrow \boxed{x_j}$. For example, $\boxed{x_1}$ is a declaration in scope (1), and $\boxed{f_2}$ is a declaration in scope (3). (Note that the declaration $\boxed{f_2}$ is not related to scope (2),

which is the scope of the body of the function bound to f_2; the parameters and local variables of a function are not accessible from outside.)

A *reference* is an occurrence that refers to a declared name. A reference is depicted by an arrow going from an occurrence to a scope, such as $x_j \longrightarrow (j)$. For example, x_4 is a reference in scope (2), and f_5 is a reference in scope (3).

2.2 Name Resolution

The name binding structure of a program is defined by a language-specific mapping from (the abstract syntax tree of) a program to a scope graph. A language-independent algorithm resolves the names in the scope graph. Name resolution is defined in terms of *reachable* and *visible* declarations.

A declaration is *reachable* from a reference if there is a *path* from the reference through scope-to-scope edges ending in a matching declaration. For example, in Fig. 1 x_1 is reachable from x_6 as witnessed by the path $x_6 \longrightarrow (3) \xrightarrow{P} (1) \longrightarrow x_1$. It is possible that there are multiple reachable declarations for a reference. For example, the reference x_4 reaches x_3 through path $x_4 \longrightarrow (2) \longrightarrow x_3$ and x_1 through path $x_4 \longrightarrow (2) \xrightarrow{P} (1) \longrightarrow x_1$.

A *visibility* policy determines which path to choose in case of an ambiguity. First, the set of reachable paths can be reduced by means of a path *well-formedness* predicate that defines *valid* paths. Typically such a predicate is expressed as a regular expression over path labels. For example, the regular expression `P*I*` expresses that paths may first follow zero or more lexical parent edges and then zero or more import edges. That is, following parent edges after import edges is not allowed. Second, from the remaining set of reachable paths, the most specific path is chosen using a partial order on edge labels and the end-of-path label \$. *Lexical shadowing*, where the closest identifier is visible, is achieved using the order \$ < P. Thus, path $x_4 \longrightarrow (2) \longrightarrow x_3$ is more specific than path $x_4 \longrightarrow (2) \xrightarrow{P} (1) \longrightarrow x_1$, which entails that x_4 resolves to x_3.

Thus, *name resolution* is defined as finding all paths to *visible declarations* that match the reference.

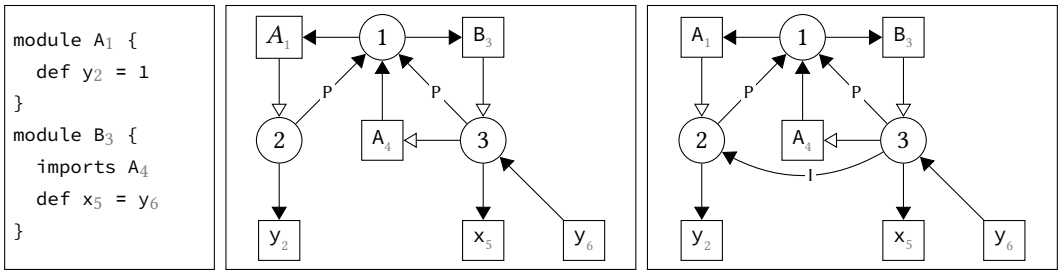
2.3 Scoped Relations

Van Antwerpen et al. [2016] define a constraint language for static semantics including constraints for declaring scope graphs and name resolution. The types of declarations are defined using constraints of the form $d : t$. These constraints are not integrated in the scope graph model. As a result declaration typings are a global property of a constraint set, and lack a scoping discipline. Furthermore, typings are not integrated in the *visual notation* of scope graphs, making awkward presentations combining scope graph diagrams with textual constraints.

In this paper we extend the representation of scope graphs with scoped relations and introduce a visual syntax for these relations. A *scoped relation* is a relation $(i) \xrightarrow{l} t$ associating a scope with a tuple t of data items labeled with the name l of the relation. A special case is a *declaration property* $(i) \longrightarrow x_j \xrightarrow{r} \ldots$, which is a short-hand for a declaration $(i) \longrightarrow x_j$ and a relation $(i) \xrightarrow{r} (x_j, \ldots)$. For example, we use the ":" label for relation mapping declarations to types. Thus, in Fig. 1 we have $(2) \longrightarrow x_3 \xrightarrow{:} \texttt{Int}$ to indicate that the type of x_3 is `Int`.

2.4 Non-Lexical Bindings

So far in this section we used scope graphs to model *lexical bindings*, which are modeled by scope edges from inner scopes to enclosing scopes, and path specificity to handle shadowing. This covers a large class of binding patterns including functional abstraction, local variable declarations, block structure, loop iterator variables, and flavors of let bindings such as sequential let, recursive let, and parallel let. [Néron et al. 2015a,b] show encodings of such binding patterns using scope graphs.



(a) Program (b) Scope graph (unresolved) (c) Scope graph (resolved)

Fig. 2. Modules with imports

However, names cannot always be resolved in the lexical scope. Another class of bindings can be characterized as 'name-dependent name resolution'. That is, to resolve a name, we first need to resolve another name. Typical examples are modules with imports, access to modules or packages through qualified names, and super class declarations in object-oriented languages. Scope graphs support this kind of binding pattern using associated scopes and imports.

An *associated scope* edge $\boxed{d_i} \rightarrow (s)$ associates a scope (s) with a declaration $\boxed{d_i}$. For example, in Fig. 2 scope 2 representing the body of module `A` is associated with its declaration: $(1) \rightarrow \boxed{\mathtt{A}_1} \rightarrow (2)$. An *import* edge $(s) \rightarrow \boxed{\mathtt{r}_i}$ entails the import into scope (s) of all declarations of the scope that is associated with the declaration to which the reference $\boxed{\mathtt{r}_i}$ resolves. For example, in Fig. 2, scope 3 representing the body of module `B` has an import edge $(3) \rightarrow \boxed{\mathtt{A}_4}$. The reference itself is a regular reference $\boxed{\mathtt{A}_4} \rightarrow (1)$, referring to a declaration with a matching name.

Resolving through an import is a two stage process. To resolve $\boxed{\mathtt{y}_6}$ in Fig. 2, first the import reference is resolved to a declaration with an associated scope: $(3) \rightarrow \boxed{\mathtt{A}_4} \rightarrow (1) \rightarrow \boxed{\mathtt{A}_1} \rightarrow (2)$. This gives rise to an edge $(3) \xrightarrow{\mathtt{I}} (2)$ from the importing scope to the associated scope of the declaration, which enables resolving the reference with the path $\boxed{\mathtt{y}_6} \rightarrow (3) \xrightarrow{\mathtt{I}} (2) \rightarrow \boxed{\mathtt{y}_2}$.

2.5 Type-Dependent Name Resolution

Another form of indirect name resolution is *type-dependent name resolution*, where it is necessary to perform type analysis before being able to resolve a name. The prototypical example is access to the named members of a class or record. Resolving the reference to field `f` in a field projection expression `e.f` requires resolving the type of the sub-expression `e`.

We illustrate the modeling of type-dependent name resolution using scope graphs with nominal record types for a language with record type declarations in Fig. 3. The example program in Fig. 3 defines a new record type `P`, creates an instance `p` of that record type, and accesses the field `y` of the instance. The unresolved scope graph in Fig. 3 includes the constraints (dashed boxes and arrows) that need to be resolved.

The record type is identified by the declaring occurrence $\boxed{\mathtt{P}_1}$. The declarations of fields of the record type are defined in a separate scope (2), which is associated with the record type. This scope is not reachable from other scopes, other than through the association of the record type.

The type of variable $\mathtt{p}_4$ is defined as $\mathtt{Rec}\ \mathtt{P}_5$, a record type with a reference to a record type declaration. In the unresolved scope graph this is represented with a constraint variable $\boxed{d_k}$, which is a placeholder for the type declaration, and used in the type of the declaration $\mathtt{p}_4$. In the graph, we use term graph notation, where the edge $\xrightarrow{i}$ denotes the ith child of the constructor.

Resolving the field access y_8 requires resolving the type of p_7, which requires resolving p_7 to some declaration p_i, and retrieving its type, which should be a record type $\texttt{REC}(d_j)$ identified by some declaration d_j, which should have some associated scope s in which the field access identifier y_8 should be resolved. These requirements are expressed in the scope graph as constraints (dashed boxes and edges). The arguments of these constraints are constraint variables, which need to be substituted with ground values. Name resolution resolves p_7 to p_4, which leads to unification of p_i with p_4, which leads to unification of d_j with P_1, which leads to unification of scope variable s with scope 2. In the graph unifications are indicated by the double dashed lines. Thus, y_8 becomes a reference in scope 2, where it can be resolved to field declaration y_3 to find out that its type is `Int`. The result is shown in the resolved scope graph in Fig. 3.

```
record P1 { x2: Int, y3: Int }
def p4 : Rec P5 = ...
def z6 : Int = p7.y8
```

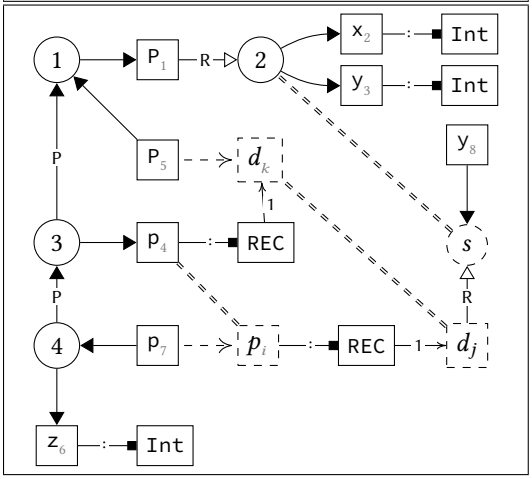


Fig. 3. Type-dependent name resolution for field access to a nominal record type.

2.6 Beyond Simple Type Systems

Scope graphs provide a language-independent model of name binding and resolution. This framework has many applications. Van Antwerpen et al. [2016] define a constraint language for declaratively defining type checkers based on scope graphs type unification . Poulsen et al. [2016] define the *scopes and frames* framework as a language-independent model of run-time memory with a systematic definition of type safety. Poulsen et al. [2018] incorporate this framework in the definition of intrinsically-typed abstract syntax, which facilitates automatic type safety checking of definitional interpreters.

However, these applications are limited to languages with *simple nominal type systems*. The key to this limitation lies in *type identity*. In the examples we have shown above and in the published work about scope graphs, types are either primitive types or user-defined types *identified by name*, and types are compared through their identity. More sophisticated type systems are complicated exactly through a more sophisticated notion of type equality or subtyping. In structural type systems, types emerging from different parts of a program may be in a subtype relation based on their structure. In type systems with type parameterization, types need to be considered modulo the instantiation of type parameters. In general, type comparison may involve some kind of *type normalization*. Parametric polymorphism (as in System F [Girard 1972; Reynolds 1974]) requires substitution. Path-dependent types [Amin et al. 2016] also require substitution, but also projection from records. And (traditional) dependent types may involve arbitrary normalization. In the next section, we show how to overcome this apparent limitation of scope graphs by considering *scopes as types*.

3 SCOPES AS TYPES

In this section we show how to characterize sophisticated notions of type equality and polymorphism in terms of scope graphs. The key to this characterization is to use *scopes as types*. That is, to use scopes to represent the structure of user-defined types such as records and classes.

```
let f1 = fun(r2: {p3: {x4: Int}}) { r5.x6 }
    q7 = { p8 = { x9 = -7, y10 = 10 } }
in f11 q12
```

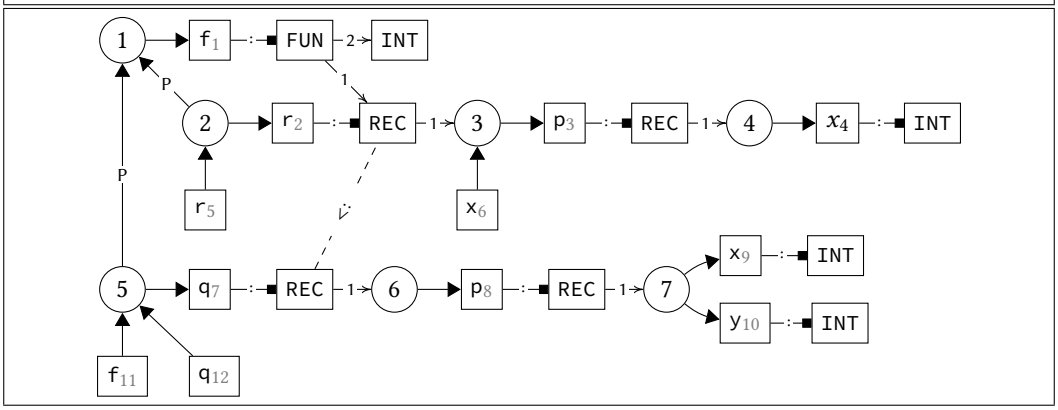


Fig. 4. Structural subtyping

Following Cardelli and Wegner [1985], there are two approaches to universal polymorphism: *inclusion polymorphism* (also called *subtype polymorphism*) characterizes and identifies different structures as being inter-changeable and inter-convertible; and *parametric polymorphism* characterizes structures that are parametric in other types or structures. Scope graphs expose the structure needed to define both kinds of polymorphic type disciplines. In contrast to traditional approaches to modeling polymorphism, the approach that we present in this section is based on a language-independent representation: the scope graph. This makes the framework well-suited for characterizing, studying, specifying, and implementing type systems for programming languages at large.

3.1 Structural Records with Structural Subtyping

We illustrate how to define a simple language with structural records by using a scope to define the type of a record, and how this definition of types affords a generic characterization of structural subtyping.

Fig. 4 defines an example program with structural record subtyping. The first line defines a function `f` whose parameter is typed by a nested record type where the innermost record has a single `Int` typed field `x`. The second line defines a nested record `q` whose innermost record has two fields, `x` and `y`. The third line applies `f` to `q`. In the scope graph in Fig. 4 we use a scope to define the type of records. For example, the $\boxed{r_2}$ declaration is typed as `REC(`(3)`)`, i.e., a record type that references the scope (3) that defines the structure of the record type in the parameter of the `f` function. Similarly, the $\boxed{q_7}$ declaration is typed as a record whose structure is defined by scope (6).

Structural subtyping is an approach to inclusion polymorphism based on structural comparison. A record type `{x: A}` (for some type `A`) is the super type of all record types that provide *at least* a field `x:B` where `B` is a subtype of `A`. By using scopes to identify types we can formulate structural subtyping in terms of the scope graph. For example, the record type given by scope (6) is a structural subtype of the record type given by scope (3) because there is a similarly-named and compatibly-typed declaration in scope (6) for every declaration in scope (3). In other words, a

structural subtyping discipline is characterized by a query over the scope graph of a program: for each declaration in the scope of the super type there is a compatibly-named and compatibly-typed declaration in the sub type. This characterization closely matches the traditional notion of structural record subtyping [Pierce 2002]; but using scopes as types makes the definition independent of the particular (abstract) syntax of languages.

3.2 Classes and Nominal Subtyping

Nominal subtyping is an alternative approach to inclusion polymorphism. Nominal subtyping identifies and relates types via a subtype ordering between names. The nominal subtype ordering for a given program is usually assumed to have been computed a priori by an ad hoc pre-processor. Using scope graphs, the subtype ordering between names is apparent from the scope graph itself. Thus scope graphs alleviate the need for ad hoc pre-processing by exposing the essential structure of programs and types. To illustrate, let us consider a Java-like example language with classes.

Fig. 5 shows a program with three classes, and the scope graph of this program. Each class has a name that is typed as a `CLASS(`s`)` where s is the scope of (the body of) the class. Class scopes have a declaration for each member of the class. For example, A_1 is associated with the class scope that has a single declaration f_2 of type `T` for the single class member of `A`. Class scopes are connected to the scope of their super class via an edge labeled S (for super) which makes the class members in super classes reachable via name resolution. S edges are the result of resolving the **`extends`** clauses of classes. For example, the class scope for `B` is connected to the class scope of `A` because A_4 in the program resolves to A_1. (For brevity we have omitted the extends clause references from the scope graph.) Thus scopes directly represent and expose the inheritance structure of classes.

```
class A1 {
    T f2;
}
class B3 extends A4 { ... }
class C5 extends B6 { ... }
... new C7().f8 ...
```

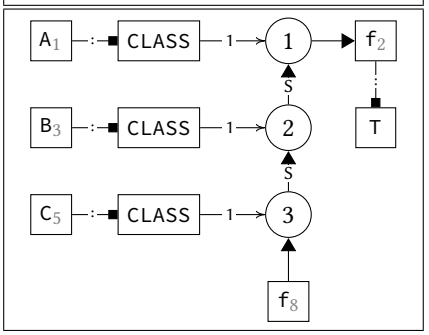


Fig. 5. Subtyping of classes

Nominal subtyping for Java entails that we can use a named sub-class where one of its super classes is expected. For example, the Java cast expression `((A)new C())` is well-typed when `A` is a super type of `C`. The scope graph affords a straightforward characterization of this subtype relationship: any class member declaration that is reachable from the class scope (1) of `A` is also reachable from the inheriting class scope of `C` (3), because (1) is reachable from (3). In other words, a nominal subtyping discipline is characterized by (1) type identity, which corresponds to scope identity, and (2) a query over the scope graph of a program: a type that references a scope (s) is the super type of all types that reference a scope from which (s) can be reached through a chain of S edges in the scope graph.

Thus, using scopes as types exposes the structure needed to define and characterize inclusion polymorphism. Next, we consider how to model parametric polymorphism.

3.3 Generic Classes and Parametric Polymorphism

We present an approach to parametric polymorphism that uses the scope graph to record delayed explicit substitutions. To illustrate, let us again consider the Java-like example language but with *generics*. Fig. 6 shows a program with a class definition `A` with a type parameter `X` and with a single field `f` typed with the type parameter `X`. The program also contains two instantiations

```
1  class A1<X2> {
2      X3 f4;
3      ...
4  }
5  ...
6  m5 = new A6<T>();
7  m7.f8;
8  n9 = new A10<S>();
9  n11.f12;
10 ...
```

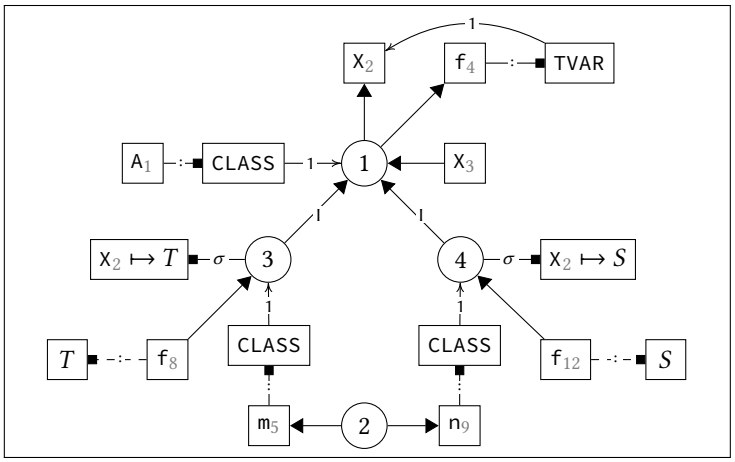


Fig. 6. Generic class with two instantiations. Resolution of field projections on instantiations applies the substitution in their instantiation scope.

of A: `m = new A<T>()` and `n = new A<S>()`. The field accesses `m.f` and `n.f` both resolve to the declaration of the field in A. However, their type should be considered relative to the specific instantiation of the type parameter. That is, `m.f` has type T and `n.f` has type S.

The scope graph in Fig. 6 illustrates how generic class instantiation (i.e., parametric polymorphism) is modeled using scope graphs: each generic class instantiation is modeled as an *instantiation scope*. For example, the instantiation `m = new A<T>()` gives rise to the instantiation scope (3) which contains a delayed and explicit substitution as a σ-labeled edge between (3) and $\texttt{X} \mapsto T$. Delayed substitutions are only applied to field types once a *field is accessed*, as opposed to eagerly when the class is initialized. By delaying the substitution as an instantiation scope we save having to duplicate the entire class scope when we instantiate the generic class A with a different generic type argument S.

The type of a generic class instance is represented by an instantiation scope, which refines the original class scope with a substitution. For example, the generic class instantiation $\texttt{m}_5$ `= new` $\texttt{A}_6\texttt{<}T\texttt{>()}$ has type `CLASS(`(3)`)`. The class members of the class scope (1) for A are reachable through the I-labeled instantiation edge between (3) and (1). Generic type parameter substitution works by enforcing the following reference resolution policy: after resolving a reference to a declaration and its type, we *normalize* the yielded type by applying each explicit substitution from each of the instantiation scopes that we encountered in the resolution path. This resolution policy is enforced for the field access expression $\texttt{m}_7\texttt{.f}_8$ as follows: resolving $\boxed{\texttt{f}_8}$ to $\boxed{\texttt{f}_4}$ yields the type $\texttt{TVAR(X}_2\texttt{)}$ which is then normalized by applying the delayed substitution $\texttt{X}_2 \mapsto T$ which makes T the result type of the field access expression.

The example in Fig. 6 illustrates how parametric polymorphism can be characterized, specified, and implemented using scope graphs. In general, parametric polymorphism allows program fragments to be typed by using variables in place of actual types, and then instantiating with actual types as needed. The way to instantiate parametric types in general is (some form of) substitution. To model parametric polymorphism in scope graphs, we make such substitutions explicit and delayed, reminiscent of the *explicit substitutions* approach due to Abadi et al. [1991], but here adapted to scope graphs. There are, of course, other ways to model such substitutions. A naive alternative is to use a substitution function that traverses a scope graph and applies substitution, similarly to

how substitution functions for the λ-calculus are often defined. Naive substitution is unattractive because: (i) substitution functions usually reside on a different meta-level from the scope graph itself which is conceptually unsatisfying; and (ii) substitution would *duplicate* parts of the scope graph, causing a scope graph size explosion proportional to the number of uses of parametric types. By making substitutions explicit, we avoid these problems. Scope graphs with explicit substitutions provide a model for parametric polymorphism in a way that the scope graph provides a uniform model and data structure for representing and inspecting programs after resolution.

4 STATIX: SPECIFICATION WITH SCOPES AND CONSTRAINTS

In the previous section we saw that viewing scopes as types allows us to model sophisticated type systems using a generic framework. In this section we describe the design of *Statix*, a new language for the specification of the static semantics of programming languages that incorporates these ideas.

4.1 Requirements

Viewing *scopes as types* relies on a generalized notion of scope graphs. In this section we analyze the consequences of this generalization on the constraint language of Van Antwerpen et al. [2016], which we will refer to as CLA, by lack of an official name. First, we highlight the key points of constraint-based type checking. Then, we discuss the main aspects of CLA, and discuss its limitations with respect to the patterns introduced in Section 3.

Constraint-Based Type Checking. Type checkers verify the well-formedness of programs with respect to a type system. This involves resolving references to declarations, checking type equality or subtyping, and possibly inferring implicit types. If the program is well-formed, the type checker assigns types to all expressions in the program. If the program is incorrect, the type checker should report any inconsistencies it found.

Specifying and implementing type checkers using constraints is a well established technique [e.g., Odersky et al. 1999; Pottier and Rémy 2005; Simonet and Pottier 2007; Sulzmann and Stuckey 2008; Vytiniotis et al. 2011]. The key point of constraint-based type checking is to reduce the problem of program well-formedness to a constraint problem. Satisfiability of the constraint problem implies well-formedness of the program. This approach has many benefits. It separates the object language from the type language, by splitting object language dependent constraint generation from type language dependent constraint solving. The constraint language can be simpler than the object language, which benefits reasoning about soundness and completeness of the solver. There is also the possibility of reuse of the constraint language and solver for different object languages. An important aspect of constraints is that they are generally order independent. The constraints specify the relations between different types, and the solver determines resolution order.

One aspect that is under developed in many approaches is the treatment of name resolution. Mostly it is considered part of the constraint generation phase. When it is part of the constraint language [e.g., Pottier and Rémy 2005], the constraints mimic the (lexical) binding structure from the object language. This is problematic if the object language contains type-dependent names, such as record fields or class methods. Name resolution is now type-dependent and non-lexical, and stratification does not work anymore.

Name Resolution Constraints. CLA addresses this problem by making name resolution part of the constraint language. A standard constraint language of type equalities is extended with constraints to resolve names, as well as assumptions about the binding structure. The binding structure is represented with scope graphs [Néron et al. 2015a], which are object language independent and come with a formal resolution principle.

The key idea underlying CLA is that the order independence of constraints enables interaction between the name resolution and type constraints. The crucial insight is that some name resolution is possible in an incomplete graph, without losing soundness of resolution. That is, any resolved reference resolves as it would in the (possibly extended) final scope graph.

Specifically, an incomplete scope graph contains edges whose targets are represented by unification variables, and thus unknown until a substitution is found. A constraint generator produces both the scope graph and the type constraints for a program, which allows sharing of unification variables between them. During constraint solving, the graph is instantiated when equalities are resolved, which gradually makes more references resolvable. The approach is declarative since it does not require the language designer to stratify resolution; the resolution engine finds an order.

Limitations. The patterns discussed in Section 3 rely on a generalized model of scope graphs, as well as on scoped relations, which were not expressible in CLA. We highlight some important limitations of CLA and illustrate them with examples.

- The resolution policy (path well-formedness and ordering) is a global property, which applies to all namespaces. However, in Java-like languages different kinds of identifiers have different policies. For example, type variables are only visible in the lexical scope of a class (path matches P^*), but fields are also visible in subclasses (path matches P^*S^*).
- The only way to query the graph is by resolving a reference to a single declaration. A few built-in queries are available, such as all visible declarations, but there is no general mechanism to describe such queries. A query to get all methods in super classes (path has at least one S-step) – useful to check overloading or overriding – cannot be defined.
- The scope graph can only contain scopes and declarations, and declarations can exist in one scope only. Types can be associated with declarations only as a global property, i.e. without the option of local (scoped) refinements. Storing a substitution for our generic class example is not possible.
- The user-defined constraint generation function defines a relation between programs and types. It is not possible to define relations between types. Type comparisons that are not syntactic equality, such as structural subtyping cannot be defined.
- The full edge structure of the scope graph is required before resolution in an incomplete graph is possible. When using scopes as types, in a language where type normalization requires the creation of new scopes as we saw in Section 3, this is too prohibitive.

Statix. We have designed *Statix*, a new language for static semantics specification, to generalize over CLA, retaining its benefits, while addressing the limitations outlined above. In the rest of this section we introduce Statix by means of examples. In subsequent sections we formalize the design.

4.2 Declarations and References

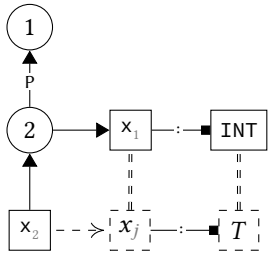
We introduce constraint rules and simple name resolution using the example in Fig. 7, which shows the program, its scope graph, and the corresponding Statix specification. The program consists of a let binding x_1 to an integer literal, and a variable reference x_2 in the body. The scope graph consists of a program scope 1, and a scope 2 introduced by the let. The let scope contains the declaration and the reference. The reference is expected to resolve to some declaration x_j, which in turn is expected to have a type T. The double dashed lines indicate the nodes that are unified during resolution. The Statix specification on the right defines how the scope graph and constraint patterns are derived from the program. The abstract syntax trees of programs are represented using term notation.

Constraint Rules. In Statix, typing rules are defined by means of constraint rules for user-defined constraints. For example, the typing rule for integer literals is defined as

Program

```
let x1 : Int = 49 in x2
```

Scope graph




```
relation   type : occurrence -> TYPE
labels     P
namespace  Var : string
resolve    Var filter pathMatch[P*] min pathLt[$ < P]

constraint typeOfExpr : scope * Expr -> TYPE

typeOfExpr(s, Int(_), INT()).

typeOfExpr(s1, Let(x, T, e1, e2), U) :- {s2} new s2,
  s2 -P-> s1,
  s2 -> Var{x@x} with type T,
  typeOfExpr(s1, e1, T),
  typeOfExpr(s2, e2, U).

typeOfExpr(s, Var(x), T) :- {p d}
  Var{x@x} in s |-> [(p, d)],
  ?type[d, T] in dst(p).
```

Fig. 7. Declarations and references

```
typeOfExpr(s, Int(i), INT()).
```

This rule is one of multiple rules for the `typeOfExpr` constraint, which is defined with signature

```
constraint typeOfExpr : scope * Expr -> TYPE
```

The signature specifies that a `typeOfExpr` constraint takes three arguments: a scope, an expression, and a type. The arrow `->` separates input arguments on the left from output arguments on the right. Input and output arguments are treated differently when inline patterns are used in the head of the rule. The normal form of the rule above is

```
typeOfExpr(s, e, T) | e == Int(i) :- T == INT().
```

The head `typeOfExpr(s, e, T)` matches on variables only. Patterns for input arguments become equalities in the guard (between the bar and the turnstile `| C :-`), while patterns for output arguments become equalities in the body (after the turnstile `:- C`).

Statix constraint rules are interpreted as simplification rules. Constraints are resolved by simplifying them to the basic built-in constraints of the language. These constraint are then solved using built-in solvers. For example, the equality constraint `T == INT()` is resolved using unification. If multiple rules are defined for a constraint, such as is the case for `typeOfExpr`, the rule guards are used to determine which rule is used for simplification. If a guard holds, the constraint is replaced by the body of that rule. If the constraints in the body are not satisfiable, no other rules are tried (no back-tracking). If none of the guards are satisfiable, the constraint is considered unsatisfiable. In general, a constraint is considered satisfiable if it can be completely reduced to built-in constraints, and the built-in constraints are satisfiable according to their specific solvers.

The rule for let introduces a new scope for the body (`new s2`), and connects it to the surrounding lexical scope by declaring a scope edge (`s2 -P-> s1`). The declaration and type edges for the let

variable are introduced by the constraint `s2 -> Var{x@x} with type T`, where `Var{x@x}` represents the occurrence of the variable in the program with its name, its position (which is derived from the token), and its namespace.

The type is associated with the scope through the `type` relation, which is a functional relation, indicated by the arrow `->` in its signature. This means that at most one type can be associated with an occurrence, and that the type may contain unification variables. For non-functional relations, all elements in the tuples of the relation must be ground. The constraint that introduces the declaration and its type is a short form that can be used for any functional relation of type `occurrence -> ...`. It can also be written as two more basic constraints `s2 -> Var{x@x}, !type[Var{x@x}, T] in s2`. A constraint `s -> Ns{...}` introduces a declaration. A constraint `!rel[...] in s2` adds a value to the relation, and can be used for relations of any type.

Resolution. The rule for variables uses the constraint `Var{x@x} in s |-> [(p, d)]` to resolve the occurrence `Var{x@x}` in scope `s` to a single pair of a path and a declaration. The variables `p` and `d` are local variables, introduced in the `{p d}` block after the turnstile. Local variables are turned into fresh constraint variables during simplification. The policy for resolving the reference is defined as

```
resolve Var filter pathMatch[P*] min pathLt[$ < P]
```

It specifies a well-formedness predicate using the **`pathMatch`** constraint, which matches path labels against a regular expression. Shadowing is specified using a **`pathLt`** constraint that compares two paths based on their labels and the given label order.

Finally, the type of the variable is related to the type of the resolved declaration using the relation constraint `?type[d, T] in p.dst`, which takes as argument a tuple of terms and the scope in which the relation is checked. This form only considers relation tuples that are declared in that exact scope. During simplification, the constraint can be resolved when all terms corresponding to types before the `->` are ground. The constraint is then simplified to equality constraints between the remaining terms and the terms found in the relation.

4.3 Queries and Resolution Policies

We illustrate rich queries on the scope graph using the structural subtyping example in Fig. 8. The example program defines a function x_2 over records with field `x`, and a record literal r_3 with fields `x` and `y`. The scope graph is a fragment showing the record types of the function parameter and the record literal. Record types are represented by scopes. The type of the function argument corresponds to scope 1, and the type of the literal to scope 2. The Statix specification on the right shows the constraints and rules that express structural subtyping over record types.

The subtype constraint is defined as a binary predicate with signature `subType: TYPE * TYPE`. Subtyping of integer types is defined by the reflective rule

```
subType(INT(), INT()).
```

The committed-choice nature of constraint resolution requires rules to have non-overlapping guards. Therefore we list all cases here, even though some would obviously fall under a general reflexivity rule, such as

```
subType(T1, T2) | T1 == T2. // problem: guard overlaps with REC rule
```

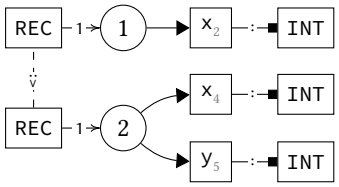
Scope Graph Queries. The rule for record subtyping is defined in terms of a query over record fields of the supertype

```
query decl filter pathMatch[e] and { Fld{_@_} }
           min pathLt[] and { Fld{x@_}, Fld{x@_} } in s2 |-> flds2
```

Program

```
let f1: {x2: Int} -> Int = ...
    r3: {x4: Int, y5: Int} = ...
in f6 r7
```

Scope graph




```
namespace Fld : string
resolve   Fld filter pathMatch[e] min pathLt[]

constraint subType : TYPE * TYPE

subType(INT(), INT()).

subType(REC(s1), REC(s2)) :- {flds2}
  query decl
        filter pathMatch[e] and { Fld{_@_} }
        min pathLt[] and { Fld{x@_}, Fld{x@_} }
        in s2 |-> flds2,
  subFields(s1, flds2).

constraint subField : scope * (path * occurrence)
constraint subFields maps subField(*, list(*))

subField(s1, (p2, d2)) :- {p1 d1 T1 T2}
  d2 in s1 |-> [(p1, d1)],
  ?type[d1, T1] in dst(p1),
  ?type[d2, T2] in dst(p2),
  subType(T1, T2).
```

Fig. 8. Scope graph queries

The query is a more general version of the resolution policies for namespaces. The query queries the relation **`decl`**, a built-in relation for declarations. The **`filter`** and **`min`** parameters specify constraints on the resolved declarations. The **`filter`** parameter takes two constraints as arguments. The first matches the path with a regular expression, while the second matches element from the relation. In this case, we are only interested in fields. The match `{ Fld{_@_} }` is an anonymous constraint matching field occurrences. It can also be written with an explicit body, as `{ d :- d == Fld{_@_} }`. The parameters for **`min`** define the ordering used for disambiguation and follow a similar pattern. The first parameter specifies the order with which paths are compared. The second specifies which elements from the relation shadow each other. In this case, fields with the same name `x` shadow each other. Disambiguation works on pairs of paths, where the path comparison determines shadowing if the two terms are comparable according to the given constraint.

Subtyping requires that all fields from the supertype are present in the subtype. This is expressed with the `subField` constraint. It resolves a field from the supertype in the subtype scope, and requires that the field types are subtypes. The types are looked up in the declaration scope using the projection `p.dst`, which projects the final scope from a resolution path.

Other Constraints as Queries. With this general syntax for queries, we can express some of the constraints in terms of queries. The resolution constraint `Var{x@x} in s |-> ps` can be expressed as

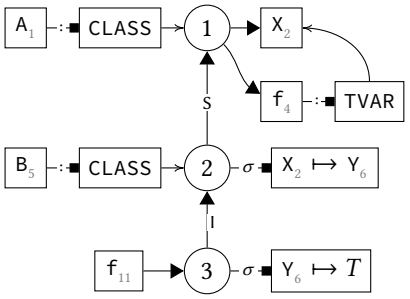
```
query decl filter pathMatch[P*] and { d :- d == Var{x@_} }
           min    pathLt[$ < P] and true in s |-> ps
```

The path match and comparison are taken from the namespace resolution policy in the signature. The matching constraints are standard for each namespace. The filter constraint matches all

Program

```
class A1<X2> { X3 f4; }
class B5<Y6> extends A7<Y8> { ... }
... new B9<T10>().f11 ...
```

Scope graph



Specification

```
constraint
  normS : list(scope) * TYPE * TYPE

normS([], T, U) :- T == U.

normS([s|ss], T@TVAR(d), U)
  |  noSubst(s, d)
  :- normS(ss, T, U).

normS([s|ss], TVAR(d), U)
  |  hasSubst(s, d, T)
  :- normS(ss, T, U).

normS([s|ss], CLASS(s), U)
  :- {es} new s', s' -I-> s,
     allSubsts(s, es),
     instWithList(s', es),
     normS(ss, CLASS(s'), U).
```

```
sort       TVAR = occurrence
constraint norm : path * TYPE * TYPE

norm(p, T, U) :-
  reverse(p.scopes, ss),
  norm(ss, T, U).

relation   subst     : TVAR -> TYPE
constraint hasSubst  : scope * TVAR * TYPE
constraint noSubst   : scope * TVAR
constraint allSubsts : scope * list(TVAR * TYPE)

hasSubst(s, d, T) :-
  query subst filter pathMatch[e] { d' :- d' == d }
        in s |-> [(_, (_, T))].

noSubst(s, d) :-
  query subst filter pathMatch[e] { d' :- d' == d }
        in s |-> [].

allSubsts(s, es) :-
  query subst filter pathMatch[e]
        in s |-> ps,
  resolvedEntries(ps, es).

constraint resolvedEntries :
    list(path * (TVAR * TYPE)) * list(TVAR * TYPE)

resolvedEntries(ps, es) :- // ... elided ...

constraint instWith : scope * (TVAR * TYPE)
constraint instWithList maps instWith(*, list(*))

instWith(s, (d, T)) :- !subst[d, T] in s.
```

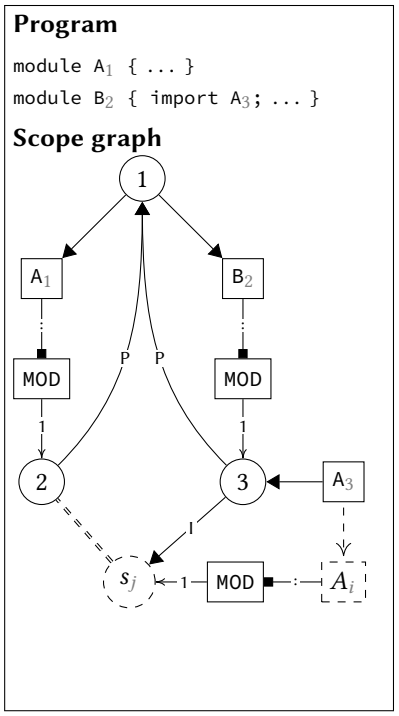
Fig. 9. Path driven type normalization

arguments except the position of the declaration with the arguments of the reference. The order constraint is always true, since we match only declarations with the same name to begin with.

Similarly, we can express the relation constraint `?type[d, T] in s` in terms of a query as follows

```
query type filter pathMatch[e] and { (d', T) :- d' == d } in s to [(_, T)]
```

The query only considers the given scope because of the empty regular expression `e`, and matches on the exact given declaration. The result should be one tuple, there the first component is the given declaration, and the second the type `T`.




```
constructors
  Module : ID * list(Declaration) -> Module
  Import : ID -> Declaration
name resolution
  resolve Mod filter pathMatch[P*] min pathLt[$ < P]
  resolve Var filter pathMatch[P*I*]
              min pathLt[$ < P, I < P]
constraints
  moduleOK : scope * Module
  declarationOK : scope * Declaration
relation
  type : occurrence -> TYPE
rules
  moduleOK(s, Module(m, decs)) :- new s_mod,
    s_mod -P-> m,
    s -> Mod{m@m} with type MOD(s_mod),
    declarationsOK(s_mod, decs).

  declarationOK(s, Import(m)) :-
    type of Mod{m@m} in s |-> [(_, MOD(s_mod))],
    s -I-> s_mod. // edge to imported module
```

Fig. 10. Encoding imports as name-dependent name resolution. The import edge from s_3 to s_j becomes an edge to s_2 after resolution of the import reference A_3 and its types are resolved. References in module B can now be resolved to declarations in module A.

4.4 Substitution and Type Normalization

Using scoped relations, we can define the type normalization required for generic classes. Fig. 9 shows an example with two generic classes, class A_1 with type parameter X_2, class B_5 with type parameter Y_6, which extends class A_1, and an instantiation of class B_5. The scope graph shows the class scopes, the inheritance structure using S-edges, and the instantiation using an I-edge. Scope 2 of class B_5 contains the substitution for X_2, scope 3 of the instantiation the substitution for Y_6. The reference f_{11} resolves to the field f_4. However, the correct type, is a normalized version of the type `TVAR(`X_2`)` of the field. Normalization corresponds to the application of all substitutions along the resolution path.

The constraint `norm(p, T, U)` relates a type T to its normalized type U relative to the resolution path p. Since the substitutions should be applied top-to-bottom, it is defined in terms of a helper constraint `normS(ss, T, U)` that normalizes over the reversed lists of scopes (projected with `p.scopes`). The base case, where the scope list is empty, simply equates the type with the normalized type. For the case that the scope list is not empty, different rules are defined for the different types.

Substitution of Variables. Two rules deal with normalization of variables. The rules are guarded to distinguish between the case where the head scope contains a substitution for the variable (`hasSubst(s, d, T)`), and the case where the scope does not substitute the variable (`noSubst(s, d)`). These guard constraints are defined in terms of queries over the `subst` relation, differing only in the expected result. In both cases the result type is the normalized type with respect to the remaining

substitution scopes. If the variable was substituted, the substituted type is used for the remaining normalization, otherwise the variable is used unchanged.

Lazy Substitution on Classes. While the rules for variables may look similar to traditional substitution, the case for classes is different. Classes are represented by their class scopes, and we cannot simply rely on term substitution. However, we can use the idea of lazy substitution as it is represented in the inheritance structure for normalized types. Instead of eagerly instantiating types in the class, requiring duplication of class definitions, we create a new instantiation scope (`new s'`), and connect it to the current class scope (`s' -I-> s`). All substitution entries defined in the current substitution scope (`allSubsts(s, es)`) are added to the new instantiation (`instWithList(s', es)`). These constraints are defined using the usual relation and query constraints. The normalized type `U` is related the new class type, relative to the remaining substitution scopes.

4.5 Imports

Using type-dependent name resolution, we can define nominal imports à la CLA. Fig. 10 shows an example with two modules, $\texttt{A}_1$ and $\texttt{B}_2$, where module $\texttt{B}_2$ imports module $\texttt{A}_1$. Module scope 3 has an edge to its lexical parent, scope 1, and an edge to the imported scope. The target of this edge is a unification variable. Resolving the import reference gives us the module declaration, its type, and thus the module scope.

Module declarations are introduced with a type that contains the (fresh) module scope, using the constraint `s -> Mod{m@m} with type MOD(s_mod)`. Module imports resolve the module reference to its type, and declare a scope edge from the importing module scope `s` to the scope `s_mod` of the imported module.

How does resolution proceed in this scenario? After all, the import reference is resolved in the scope where the import edge is added. This is taken care of using the resolution policy. Module references can only be resolved in the lexical context (via P edges). The resolution algorithm determines that the import edge with label I is inconsequential for the resolution of the module reference. Variable references inside the module may use a resolution policy like $\mathsf{P}^*\mathsf{I}^*$ to resolve in the lexical context and via imports.

4.6 Case Studies

We have defined Statix specifications for several known calculi: A structural record calculus with width and depth subtyping [Pierce 2002]; Featherweight Generic Java [Igarashi et al. 2001]; System F [Girard 1972; Reynolds 1974]. We include the specifications as appendices.

5 GENERALIZED AND SIMPLIFIED NAME RESOLUTION

In this section we present a calculus for name resolution in scope graphs and a corresponding name resolution algorithm. On the one hand, our definition simplifies previous definitions by omitting imports, which complicated the calculus and algorithm. In the previous section we saw that imports can be expressed using scoped relations and constraints. On the other hand, we generalize previous definitions by generalizing declarations to scoped relations.

5.1 Resolution Calculus

Fig. 11 presents our modified name resolution calculus, which defines paths, reachability, and visibility in scope graphs. The calculus is defined with respect to an implicit scope graph consisting of labeled edges between scopes ($s_1 \xrightarrow{l} s_2$) and edges between scopes and data ($s \xrightarrow{r} t$). There is a *path* p between scope s_1 and s_2 if $\vdash p : s_1 \rightarrow\!\!\!\!\!\rightarrow s_2$. A path is a sequence of the form $s_1 \cdot l_1 \cdot \ldots \cdot s_n$ marking the scopes and edge labels traversed. A datum t is *reachable* from scope s if $\vdash (p, t) : s \rightarrowtail t$,

which is the case according to rule (NR-Rel) if there is a path p from scope s to some scope s', t is a datum in s', and the path and datum are well-formed. Finally, a datum t is *visible* from scope s if $\vdash (p, t) : s \overset{r}{\mapsto} t$, which is the case if t is reachable through path p, and that path is the 'smallest' reachable path according to the ordering $<$.

The differences with respect to the resolution calculus of Néron et al. [2015a]; van Antwerpen et al. [2016] are: (1) The resolution judgement has been generalized to support queries in the scope graph. Queries may resolve to declarations, as before, but also to data items, and may produces sets of such results. (2) Imports and associated scopes (as illustrated in Fig. 2) have been removed as built-in features. As we saw in Section 4.5, those features can be encoded using scopes as types, type edges, and edges with constraint variables. Leaving out imports simplifies the calculus since there is no need to carry along 'seen imports' to avoid cyclic imports. (3) Well-formedness and ordering predicates are defined *per resolution query* instead of globally (for a language definition). This makes these predicates a run-time parameter of the algorithm.

5.2 Resolution in Incomplete Graphs

We introduced the use of constraint variables for endpoints of scope graph edges. These edges only become concrete after name and/or type resolutions. This is crucial to the approach since it allows modeling of name-dependent and type-dependent name resolution for binding patterns such as module imports, class inheritance, and record/object field access. However, the question is how to do sound resolution in such incomplete scope graphs.

Consider the concrete scenario depicted by the scope graph in Fig. 12. Reference $\mathtt{A}_3$ can be resolved to declaration $\mathtt{A}_1$ through path $\boxed{\mathtt{A}_3} \to (3) \xrightarrow{\mathtt{P}} (1) \to \boxed{\mathtt{A}_1}$, and this resolution can then be used to instantiate constraint variables A_i and s such that the Q edge from scope (3) gets scope (2) as its target. Now the scope graph has changed, and we can reconsider the resolution of reference $\mathtt{A}_3$. In addition to the path above we now also have the path $\boxed{\mathtt{A}_3} \to (3) \xrightarrow{\mathtt{Q}} (2) \to \boxed{\mathtt{A}_2}$. If we decide that this new resolution shadows the first (the path is more specific), then we have a new result for the resolution of $\mathtt{A}_3$. But then the Q edge targets scope (4), and $\mathtt{A}_2$ is no

Definitions

scope graph	$\mathcal{G}$		
projections			scopes, edges, labels
scopes	s	$\in$	$\mathcal{S}$
edge labels	l	$\in$	$\mathcal{L}$
relations	r	$\in$	$\mathcal{R}$
paths	p	$:=$	$s \mid s{\cdot}l{\cdot}p$
well-formed	$\mathrm{WF}(p)$	$:=$	$\mathrm{labels}(p) \in \mathcal{E}$
	$\mathrm{WF}(t)$		over terms in r
label regexp	$\mathcal{E}$		on $\mathcal{L}$
path order	$p_1 < p_2$	$:=$	$(p_1, p_2) \in \mathcal{O}$
label partial order	$\mathcal{O}$		on $\mathcal{L} \cup \{\$\}$
datum match	$t_1 \simeq t_2$		on terms in r

Resolution calculus

$$\text{(NR-Id)}\ \frac{}{\vdash s : s \twoheadrightarrow s}$$

$$\text{(NR-Trans)}\ \frac{s_1 \xrightarrow{l} s_2 \qquad \vdash p : s_2 \twoheadrightarrow s_3 \qquad s_1 \notin \mathrm{scopes}(p)}{\vdash s_1{\cdot}l{\cdot}p : s_1 \twoheadrightarrow s_3}$$

$$\text{(NR-Rel)}\ \frac{\vdash p : s \twoheadrightarrow s' \qquad \mathrm{WF}(p) \qquad s' \xrightarrow{r} t \qquad \mathrm{WF}(t)}{\vdash (p, t) : s \overset{r}{\rightarrowtail} t}$$

$$\text{(NR-Vis)}\ \frac{\vdash (p, t) : s \overset{r}{\rightarrowtail} t \qquad \nexists p' t'.\ \left(\vdash p' : s \overset{r}{\rightarrowtail} t' \wedge t \simeq t' \implies p' < p\right)}{\vdash (p, t) : s \overset{r}{\mapsto} t}$$

Fig. 11. Name resolution calculus

longer reachable, and therefore resolution flips back to A_1. The result is not only unsound, but also unstable!

Whether A_2 does indeed shadow A_1 depends on the visibility policy of the resolution. To see what may happen here, we consider the two ingredients used to resolve A_3: the well-formedness predicate and the path order. For each scenario, we consider wether we can safely resolve A_3. The following table summarizes all scenarios with in the rows the well-formedness predicate and in the columns the path order:

	$P < Q$	$P \not< Q$
$P.p$	Yes: cannot resolve via Q	Yes: cannot resolve via Q
$Q.p$	Yes: cannot resolve via P	Yes: cannot resolve via P
$(P\|Q).p$	Yes: resolve via Q cannot shadow via P	No: resolve via Q complements or shadows via P

The first column shows that if we prefer P paths over Q paths, we will always prefer the resolution to A_1, and the initial resolution is not affected by the discovery of the Q edge. The first two rows of the second column show that if we can only resolve via either a P edge or a Q edge, there is still no problem. (In the latter case A_3 cannot be resolved to A_1 and the Q edge does not emerge, A_3 cannot resolved at all. But that is fine.) Finally, the last row in the second column shows the problematic scenario, where we allow resolution through P and Q edges, and P paths are not preferred over Q paths. Then the unsound and unstable scenario arises. Note that there would be no problem if scope (2) would not have a declaration with name A.

Thus, incomplete scope graphs do not lead to unsound resolution in all scenarios. We should prevent attempting to resolve a reference when it is clear that this may result in unsound resolution. Considering the initial scope graph in Fig. 12 and the resolution policy for reference A_3, we should determine whether it is safe to attempt resolution, i.e. in all cases but the lower right cell in the table. This reduces to the question whether it is safe to do resolution from scope (3) through the P edge, since it also has an edge to an unresolved scope. That is, is the scope *complete for resolution through label P*? If we know from the resolution policy that completion of the incomplete edge cannot lead to shadowing of resolutions through the complete edge, then it is safe to do so. Thus, while (3) is incomplete, it is complete *enough*. This property is used by the resolution algorithm that we describe next to ensure sound name resolution. See Appendix A for a technical discussion of l-completeness.

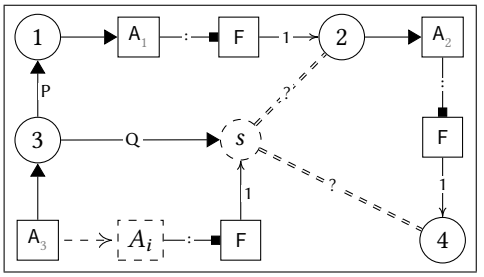


Fig. 12. Resolution in incomplete scope graph

5.3 Resolution Algorithm

Fig. 13 defines a resolution algorithm that implements the resolution calculus of Fig. 11. That is, $(p, t) \in Res^r(s)$ iff $\vdash (p, t) : s \overset{r}{\longmapsto} t$. The definition of the algorithm follows [Van Antwerpen et al. 2016], applying the generalization and simplification of the calculus. Following the generalization of the calculus, the resolution algorithm supports resolution of relations instead of just name declarations. Following the simplification of the calculus, the algorithm does not support imports. This simplifies the algorithm since it no longer needs to keep track of seen imports. Since paths now include the scopes traversed, checking for cycles is done by checking the path in the definition of $\textsc{Res}^r_{re}$. Thus, the algorithm also does not have to keep track of seen scopes separately.

Definitions

$$s \uparrow l \qquad s \text{ incomplete in } l$$

Resolution algorithm

$$\textsc{Res}^r(s) := \textsc{Res}^r_{\mathcal{E}}(s)$$

$$\textsc{Res}^r_{re}(p{\cdot}s) := \begin{cases} \emptyset & \text{if } s \in \text{scopes}(p) \text{ or } re = \emptyset \\ \textsc{Res}^r_{\mathcal{L}\cup\{\$\},re}(p{\cdot}s) & \text{otherwise} \end{cases}$$

$$\textsc{Res}^r_{L,re}(p{\cdot}s) := \biguplus_{l \in \max(L)} \left(\textsc{Res}^r_{\{l' \in L \mid l' < l\},re}(p{\cdot}s) \lhd \textsc{Res}^r_{l,re}(p{\cdot}s)\right)$$

$$\textsc{Res}^r_{\$,re}(p{\cdot}s) := \begin{cases} \emptyset & \text{if } \epsilon \notin re \\ \bot & \text{if } \epsilon \in re \wedge \left(s \uparrow r \vee \exists t.\, (s \xrightarrow{r}\!\!\blacksquare\ t \wedge \text{WF}(t) = \bot)\right) \\ \{(p{\cdot}s, t) \mid s \xrightarrow{r}\!\!\blacksquare\ t \wedge \text{WF}(t)\} & \text{otherwise} \end{cases}$$

$$\textsc{Res}^r_{l,re}(p{\cdot}s) := \begin{cases} \bot & \text{if } s \uparrow l \\ \biguplus_{p' \in \left\{p{\cdot}s{\cdot}l{\cdot}s' \,\middle|\, s \xrightarrow{l} s'\right\}} \textsc{Res}^r_{(\partial_l\, re)}(p') & \text{otherwise} \end{cases}$$

with the following auxiliary definitions:

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

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

and the shadowing and union operators on environments

$$E_1 \lhd E_2 := \begin{cases} E_1 & \text{if } E_1 \neq \bot \wedge E_1 \neq \emptyset \wedge (\simeq) \equiv \top \\ E_1 \cup (E_2 \ominus E_1) & \text{if } E_1 \neq \bot \wedge E_2 \neq \bot \wedge (\simeq) \not\equiv \top \\ \bot & \text{otherwise} \end{cases}$$

$$E_1 \ominus E_2 \triangleq \begin{cases} \bot & \text{if } \exists t_1, t_2.\, (t_1 \simeq t_2 = \bot) \\ \{(p_1, t_1) \in E_1 \mid \nexists (p_2, t_2) \in E_2.\, t_1 \simeq t_2\} & \text{otherwise} \end{cases}$$

$$\biguplus_{i \in I} E_i := \begin{cases} \bot & \text{if } \exists i \in I.\, E_i = \bot \\ \bigcup_{i \in I} E_i \end{cases}$$

Fig. 13. Name resolution algorithm

The algorithm returns $\bot$ in case of resolution through a scope s that is incomplete for a label l. This indicates to the constraint solver (Appendix B) that resolution is not (yet) possible.

6 SYNTAX AND SEMANTICS OF STATIX

In this section we describe the syntax and a declarative semantics for Statix, in terms of a constraint satisfaction relation. Fig. 14 defines the syntax of Statix in a mathematical notation that is convenient and concise for the definition of the semantics. The declarative semantics, shown in Fig. 15, defines a constraint satisfaction relation that specifies whether a constraint C is satisfied relative to a model,

user-defined sorts	u	$\in$	$\mathcal{U}$
term sorts	σ	$:=$	$u \mid \textsc{Scope} \mid \textsc{Occurrence} \mid \textsc{Path}$
term function symbols	f, g	$\in$	$\mathcal{T}$
type mapping	$type$	$:$	$\mathcal{T} \to \mathcal{P}(\sigma_1 \times \cdots \times \sigma_n \to u)$
unification variable atoms	v	$\in$	$\mathcal{V}$
occurrences	r, d	$:=$	x_i
term variables	α, β, γ		
terms	t	$:=$	$\alpha \mid f(\vec{t}) \mid v \mid s$
type mapping	$type$	$:$	$\mathcal{R} \to \mathcal{P}(\sigma_1 \times \cdots \times \sigma_i \to \sigma_j \times \cdots \times \sigma_n)$
constraint symbols	c	$\in$	$\mathcal{C}$
type mapping	$type$	$:$	$\mathcal{C} \to \mathcal{P}(\sigma_1 \times \cdots \times \sigma_n)$
constraints	C	$:=$	$\mathbf{t} \mid \mathbf{f} \mid C \wedge C \mid c(\vec{t}) \mid t \equiv t \mid t \not\equiv t$
		$\mid$	$t \xrightarrow{l} t \mid t \xrightarrow{r}\!\blacksquare\, t \mid \mathbf{q}(re, c, ord, c) \to s \mapsto ps$
program	P	$:=$	$rule^*$
rules	$rule$	$:=$	$c(\vec{\alpha}) \mid \exists \vec{\beta}.\, C_G \vdash \nabla \vec{\gamma}.\, C_B$

Fig. 14. Syntax of Statix

consisting of a scope graph $\mathcal{G}$ and a substitution φ. The semantics is defined in terms of a program P, syntactic equality and substitution, and the resolution calculus from Section 5. We explain how we deal with freshness of unification variables and scopes, completeness and minimality of the scope graph, and the definition of the name resolution parameters.

Freshness. Constraint rules introduce fresh unification variables ($\exists\beta$) and fresh scopes ($\nabla\vec{\gamma}$). The problem of freshness is that this is a global property, stating that the fresh value is not used as a fresh value anywhere else. This property is modeled in the semantics using the notion of support [Gabbay and Pitts 2002]. The satisfaction relation is parametrized by a set of variables and a set of scopes, which indicate which values are consumed. We look at the two important cases, user constraints and conjunction. According to rule (DS-C), a user constraint is satisfied if the program contains a rule for the constraint, for which the guard and body of the rule are satisfied. The fresh values picked are added to the support, while we ensure that they are not consumed for the recursive check. Satisfaction of conjunction, defined in rule (DS-Conj) is expressed in terms of satisfaction of the left and right conjuncts. Freshness is enforced by disallowing the conjuncts to consume the same values. The support of the conjunction itself is the union of the disjoint supports of the conjuncts.

Completeness and Minimality. We are often interested in establishing whether the scope graph in the model is minimal with respect to the given constraints. Therefore, the satisfaction relation is also parametrized by a set E of edges. This set is simply the union of all edges that are specified in the constraint. Using the sets S and E, the criterion for a minimal model is as follows:

$$\mathcal{G}, \varphi \models_{\langle V,S,E\rangle} C \implies S \subseteq \text{scopes}(\mathcal{G}) \wedge E \subseteq \text{edges}(\mathcal{G})$$

Entailment. Name resolution parameters, such as the well-formedness predicate on data, are defined in terms of constraint entailment. Constraint entailment states that the constraint is satisfiable in some model that is an extension of the given model. This extended model cannot

Definitions

$$\begin{array}{rrcl} \text{substitution} & \varphi, \theta & : & \mathcal{V} \to t \\ \text{variable support} & V & \subseteq & \mathcal{V} \\ \text{scope support} & S & \subseteq & \mathcal{S} \\ \text{edge support} & E & \subseteq & \mathcal{S} \times \mathcal{L} \times \mathcal{S} \end{array}$$

Constraint satisfaction $\boxed{\mathcal{G}, \varphi \models_{\langle V,S,E \rangle} C}$

$$\text{(DS-True)}\ \frac{}{\mathcal{G}, \varphi \models_{\langle V,S,E \rangle} \mathbf{t}}$$

$$\text{(DS-Conj)}\ \frac{\mathcal{G}, \varphi \models_{\langle V_1,S_1,E_1 \rangle} C_1 \quad \mathcal{G}, \varphi \models_{\langle V_2,S_2,E_2 \rangle} C_2 \quad V_1 \cap V_2 = \emptyset \quad S_1 \cap S_2 = \emptyset}{\mathcal{G}, \varphi \models_{\langle V_1 \cup V_2, S_1 \cup S_2, E_1 \cup E_2 \rangle} C_1 \wedge C_2}$$

$$\text{(DS-C)}\ \frac{\begin{array}{c} \text{“}c(\vec{\alpha}) \mid \exists \vec{\beta}.\, C_G \vdash \nabla \vec{\gamma}.\, C_B\text{”} \in P \quad \vec{v} \subseteq \mathcal{V} \setminus V \quad \vec{n} \subseteq \mathcal{S} \setminus S \quad \theta = [\vec{t}/\vec{\alpha}, \vec{v}/\vec{\beta}, \vec{n}/\vec{\gamma}] \\ \mathcal{G}, \varphi \Vdash_{\langle V,S \rangle} C_G\theta \qquad\qquad \mathcal{G}, \varphi \models_{\langle V,S,E \rangle} (C_G \wedge C_B)\theta \end{array}}{\mathcal{G}, \varphi \models_{\langle V \cup \{\vec{v}\}, S \cup \{\vec{n}\}, E \rangle} c(\vec{t})}$$

$$\text{(DS-Eq)}\ \frac{t_1\varphi = t_2\varphi}{\mathcal{G}, \varphi \models_{\langle V,S,E \rangle} t_1 \equiv t_2} \qquad \text{(DS-Ineq)}\ \frac{t_1\varphi \neq t_2\varphi}{\mathcal{G}, \varphi \models_{\langle V,S,E \rangle} t_1 \not\equiv t_2}$$

$$\text{(DS-Edge)}\ \frac{s_1 = t_1\varphi \quad s_2 = t_2\varphi \quad (s_1 \xrightarrow{l} s_2) \in \text{edges}(\mathcal{G})}{\mathcal{G}, \varphi \models_{\langle V,S,E \cup \{(s_1,l,s_2)\} \rangle} t_1 \xrightarrow{l} t_2}$$

$$\text{(DS-Rel)}\ \frac{n = t\varphi \quad (n \xrightarrow{r} (t_1, \ldots, t_n)\varphi) \in \text{edges}(\mathcal{G})}{\mathcal{G}, \varphi \models_{\langle V,S,E \cup \{(s_1,r,s_2)\} \rangle} t \xrightarrow{r} (t_1, \ldots, t_n)}$$

$$\text{(DS-Resolve)}\ \frac{\begin{array}{c} s_1 = t_1\varphi \qquad\qquad [(p_1, t_1'), \ldots, (p_n, t_n')] = t_2\varphi \\ \forall i \in \{1..n\}.\, \left(\left[\!\!\left[\begin{array}{ll} \mathcal{E} \triangleq re & \text{WF}(t) \triangleq \mathcal{G}, \varphi \Vdash_{\langle V,S \rangle} c_1(t) \\ \mathcal{O} \triangleq ord & t_1 < t_2 \triangleq \mathcal{G}, \varphi \Vdash_{\langle V,S \rangle} c_2(t_1, t_2) \end{array} \right]\!\!\right] \mathcal{G} \vdash p_i' : s_1 \overset{r}{\mapsto} t_i' \right) \end{array}}{\mathcal{G}, \varphi \models_{\langle V,S,E \rangle} \mathbf{q}(re, c_1, ord, c_2) \rightarrow t_1 \overset{r}{\mapsto} t_2}$$

Entailment

$$\begin{array}{rcl} \mathcal{G}, \varphi \Vdash C & \triangleq & \exists \mathcal{G}', \varphi'.\, (\mathcal{G}', \varphi' \models_{\langle V',S',E' \rangle} C \wedge \mathcal{G} \Vdash \mathcal{G}' \wedge \varphi \Vdash \varphi' \\ & & \qquad \wedge \text{vars}(\varphi) \cap V' = \emptyset \wedge \text{scopes}(\mathcal{G}) \cap S' = \emptyset) \\ \mathcal{G} \Vdash \mathcal{G}' & \triangleq & \text{edges}^+(\mathcal{G})|_S = \text{edges}^+(\mathcal{G}', s)|_S \quad \text{where } S = \text{scopes}(\mathcal{G}) \\ \varphi \Vdash \varphi' & \triangleq & \varphi =_\alpha \varphi'|_{\text{dom}(\varphi)} \end{array}$$

Fig. 15. Declarative Semantics of Statix

restrict the given model in any way, which is captured in the entailment definitions for substitutions and scope graphs. The substitution necessary for entailment must be equal to the given substitution on its domain, modulo renaming of variables. The extended scope graph must match outgoing edges on all scopes that are in the given scope graph.

Operational Semantics. In Appendix B we define an operational semantics for Statix.

7 RELATED WORK

Statically-typed programming languages let us reason about how programs are structured, and how programs pass around (structured) data. Previous work on scope graphs has shown that scope graphs are a conceptually attractive approach to defining the structure of programs. The observation that we make in this paper is that scopes are also the *types* of the structured data that programs pass around. We have argued that this observation facilitates the characterization, study, specification, and implementation of type systems for programming languages. The facilitation of these goals is an activity that has been widely pursued in the literature, and remains an active and important area of research. We have already discussed how our work relates to previous work throughout the paper. Here we discuss other related work.

Type systems, declaratively. Ott [Sewell et al. 2010] is a tool for formally specifying programming languages with name binding. Ott can automatically generate data types and substitution functions for different proof assistant back-ends. Lem [Mulligan et al. 2014] provides similar support. Needle and Knot [Keuchel et al. 2016] is inspired by Ott, but focuses specifically on name binding, and provides more flexible and extensive support than Ott for declaratively specifying name binding and generating substitution functions *and* lemmas about these for proof assistant back-ends. Statix does not (yet) support generating infrastructure in proof assistants, although scope graphs have been formalized in both Coq [Poulsen et al. 2016] and Agda [Poulsen et al. 2018]. Ott, Lem, and Needle and Knot are specification languages that focus on *defining* a language. In contrast, Statix is a language designed to support execution of declarative type checkers. There are many other languages for semantic specification, including PLT Redex [Klein et al. 2012], the K Framework [Rosu and Serbanuta 2010], and funcons [Churchill et al. 2014]. These frameworks also provide ways of declaratively specifying (static) semantics and obtaining prototype implementations from the declarative specification, but they do not provide integrated support for name binding, and specifications typically rely on language specific encodings and representations of name binding, structured types, and polymorphism.

Type systems, algorithmically. Type systems are often specified using inference rules that abstract from the details of how to implement the type system. There are many approaches to bridging the divide between declarative theory and practical implementation. Pierce [2002] uses *algorithmic typing rules*. A popular approach for specifying a type system in a way that the specification devises an implementation strategy is *bidirectional type checking* [Pierce and Turner 2000] which mixes *type synthesis* (inference) and *type checking*. Type inference is classically based on unification [Damas and Milner 1982; Hindley 1969; Milner 1978]. Logic programming languages in the Prolog and Datalog family provide built-in support for unification, but controlling the order in which unification happens can become unwieldy; not least due to back-tracking and the untyped nature of many logic programming languages. Approaches using *Constraint handling rules* (CHR) [Frühwirth 2009] translate a type checking problem into a constraint satisfaction problem, often solved by using unification. In the presence of type system features like parametric polymorphism, care must be taken that constraints are solved in an order that constraint variables are safely generalized and specialized (substituted). Statix is a language that borrows and combines ideas from many of these approaches: it is a typed logic programming languages (akin to Mercury [Somogyi et al. 1996]) with intentionally-limited support for back-tracking; it is based on constraint solving like CHR; and, being based on constraint solving and unification, Statix supports both type synthesis and type checking, like bidirectional typing.

Type systems with structured types. A recent example of an expressive type discipline which inspired the work that we describe in this paper, is dependent object types (DOT) [Amin et al. 2016; Amin and Rompf 2017]. The DOT calculus is an idealized version of Scala, and the type system of DOT is far removed from how one could implement a type system for Scala. There appears to have been made recent progress [Nieto 2017] on formulating algorithmic typing rules for DOT, but those rules still use substitution to model type parameter instantiation, just like Featherweight Generic Java [Igarashi et al. 2001] models generic type parameter instantiation using substitution. As we argued earlier in Section 3.3, substitution causes duplication which is both conceptually and practically unsatisfactory. We conjecture that scope graphs provide a good model for specifying and implementing type systems for languages with structured types like DOT.

REFERENCES

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. https://doi.org/citation.cfm?id=3009866

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/db/journals/jacm/Brzozowski64.html

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (1985), 471–522.

Martin Churchill, Peter D. Mosses, and Paolo Torrini. 2014. Reusable components of semantic specifications. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 145–156. https://doi.org/10.1145/2577080.2577099

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. 207–212.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël 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. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

Thom Frühwirth. 2009. *Constraint Handling Rules*. Cambridge University Press.

Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.* 13, 3-5 (2002), 341–363. https://doi.org/link/service/journals/00165/bibs/2013003/20130341.htm

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris 7.

Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc* 146 (December 1969).

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

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 419–445. https://doi.org/10.1007/978-3-662-49498-1_17

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 285–296. https://doi.org/10.1145/2103656.2103691

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 175–188. https://doi.org/10.1145/2628136.2628143

Abel Nieto. 2017. Towards algorithmic typing for DOT (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, 2–7. https://doi.org/10.1145/3136000.3136003

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015a. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015b. *A Theory of Name Resolution with Extended Coverage and Proofs*. Technical Report TUD-SERG-2015-001. Delft University of Technology, Software Engineering Research Group. https://doi.org/twiki/pub/Main/TechnicalReports/TUD-SERG-2015-001.pdf

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *TAPOS* 5, 1 (1999), 35–55.

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

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

François Pottier and Diddier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, 389–489.

Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.20

Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158104

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science)*, Bernard Robinet (Ed.), Vol. 19. Springer, 408–423.

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems* 29, 1 (2007), 1. https://doi.org/10.1145/1180475.1180476

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic and Algebraic Programming* 29, 1-3 (1996), 17–64.

Martin Sulzmann and Peter J. Stuckey. 2008. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming* 18, 2 (2008), 251–283. https://doi.org/10.1017/S0956796807006569

Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. https://doi.org/10.1145/2661136.2661149

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

A L-COMPLETENESS

In Section 5 we discussed resolution in an incomplete scope graph. Here we give formal definitions related to l-completeness and sound name resolution.

We say a graph $\mathcal{G}'$ extends $\mathcal{G}$, written as $\mathcal{G} \sqsubseteq \mathcal{G}'$, if

$$\text{scopes}(\mathcal{G}) \subseteq \text{scopes}(\mathcal{G}') \wedge \text{edges}(\mathcal{G}) \subseteq \text{edges}(\mathcal{G}')$$

Given scope graphs $\mathcal{G}$ and $\mathcal{G}'$ such that $\mathcal{G} \sqsubseteq \mathcal{G}'$, we say scope s is l-complete in $\mathcal{G}$ with respect to $\mathcal{G}'$, written as $\mathcal{G} \downarrow^{l}_{s} \mathcal{G}'$ if

$$\forall s'.\, (s \xrightarrow{l} s') \in \text{edges}(\mathcal{G}) \iff (s \xrightarrow{l} s') \in \text{edges}(\mathcal{G}')$$

and similarly for relations $\mathcal{G} \downarrow^{r}_{s} \mathcal{G}'$ if

$$\forall t.\, (s \xrightarrow{r} \blacksquare\, t) \in \text{edges}(\mathcal{G}) \iff (s \xrightarrow{r} \blacksquare\, t) \in \text{edges}(\mathcal{G}')$$

Given scope graphs $\mathcal{G}$ and $\mathcal{G}'$ such that $\mathcal{G} \sqsubseteq \mathcal{G}'$, then resolution of r in s in $\mathcal{G}$ is sound with respect to $\mathcal{G}'$

$$\mathcal{G} \vdash p : s \overset{r}{\longmapsto} t \implies \mathcal{G}' \vdash p : s \overset{r}{\longmapsto} t$$

and it is complete if

$$\forall t.\, \left(\mathcal{G}' \vdash p : s \overset{r}{\longmapsto} t \implies \mathcal{G} \vdash p : s \overset{r}{\longmapsto} t\right)$$

We conjecture that the resolution algorithm is sound and complete

$$\forall t.\, \left((p, t) \in \textsc{Res}^{r}(s) \iff \mathcal{G} \vdash p : s \overset{r}{\longmapsto} t\right)$$

Given scope graphs $\mathcal{G}$ and $\mathcal{G}'$ such that $\mathcal{G} \sqsubseteq \mathcal{G}'$, we conjecture that the resolution algorithm is sound up to l-completeness

$$\mathcal{G} \vdash \textsc{Res}^{r}_{re}(s) = \bot \vee \mathcal{G} \vdash \textsc{Res}^{r}_{re}(s) = \mathcal{G}' \vdash \textsc{Res}^{r}_{re}(s)$$

B OPERATIONAL SEMANTICS OF STATIX

In Section 6 we defined a declarative semantics of Statix, which specifies whether a solution to a constraint problem is correct. In this appendix we describe an operational semantics of Statix, which defines how to compute solutions to constraint problems, We define the semantics as a rewrite relation on configurations of constraints and partial solutions. The rewrite relations takes care of generating fresh unification variables and scopes; entailment checking and constraint simplification; unification; scope graph construction and resolution. We also define a relation that specifies if scopes are complete with respect to a constraint, which is used to implement the safety predicate for scope graph resolution.

Rewrite Rules. The operational semantics in Fig. 16 and Fig. 17 is defined as a rewrite relation on configurations of the form $\langle C \mid D; \mathcal{G}; \varphi; V, S\rangle$, consisting of a constraint stack $C \mid D$, a scope graph $\mathcal{G}$, a unifier φ, a set of unification variables V, and a set of scope variables S. When type checking a program we start with a constraint and empty components. However, we can start from any well-formed state, which is a state where $\text{vars}(\varphi, \mathcal{G}) \subseteq V \wedge \text{scopes}(\mathcal{G}) \subseteq S$. This is useful for evaluating queries in an existing scope graph.

Freshness. The algorithm needs to pick fresh values for scopes and unification variables, when applying constraint rules. Two sets V and S represent unification variables and scopes that are already used. Fresh values are selected from the domain, excluding the values that are already present in these sets.

Entailment. Simplifying a rule, as specified in rule (RS-C-Simp), requires that its guard constraints C_G are satisfied before the constraint is replaced by the body constraints C_B. Entailment checking $\langle D; \mathcal{G}; \varphi; V, S\rangle \Vdash C$ is implemented by reducing the guard constraint. If the constraint reduces to $\mathbf{t}$, we check if the resulting components do not restrict the initial configuration that we started with. This means that scopes were not extended with new outgoing edges (although new ones may have been introduced) and that unification variables were not instantiated (although they may have been renamed). If the constraint reduces to $\mathbf{f}$, entailment does not hold. If the reduction got stuck, we must delay this check. This can be the case if a scope is not complete yet, so a resolution constraint cannot be resolved. Because of entailment, the rewrite relation is defined over a constraint stack, instead of a simple constraint. Only the first component of the stack is reduced. The rest of the stack represents the context in which the constraint is reduced. The safety of name resolution depends on the constraint as well as the context. We check for the whole stack to see if a scope is complete.

Unification. Unification uses a standard unification algorithm.

Scope Extension. Fig. 18 defines the relation $C \overset{l}{\hookrightarrow} s$ to check if a scope may be extended with an l-edge by a constraint C, which is passed to the resolution algorithm to determine whether resolution is safe in a scope.

We also use this relation to impose some static restrictions on specifications, to rule out situations that would prevent any progress to be made. For any rule we require

$$\forall(\text{“}c(\vec{\alpha}) \mid \exists\vec{\beta}.\, C_G \vdash \nabla\vec{\gamma}.\, C_B\text{”} \in P)\nexists l.\, \left(C_G \overset{l}{\hookrightarrow} \alpha \vee (C_G \wedge C_B) \overset{l}{\hookrightarrow} \beta\right)$$

The definition of the scope extension relation here is simple. It is not defined for scopes wrapped in other terms. This technique can be extended to work for these cases as well, by using projections into terms instead of top-level positions (i.e., the i in α_i). However, to keep the presentation compact, and because it does not add anything fundamental to the principle, we omit those details here.

Rewrite relation $\boxed{\langle C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle}$

$$\text{(RS-True)}\ \frac{}{\langle \mathbf{t} \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

$$\text{(RS-False)}\ \frac{}{\langle \mathbf{f} \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \bot}$$

$$\text{(RS-Eq)}\ \frac{\mathsf{unify}(t_1, t_2) = \theta}{\langle t_1 \equiv t_2 \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C\theta \mid D\theta; \mathcal{G}\theta; \varphi\theta; V, S\rangle}$$

$$\text{(RS-NotEq)}\ \frac{\mathsf{unify}(t_1, t_2) = \bot}{\langle t_1 \equiv t_2 \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle \mathbf{f} \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

$$\text{(RS-Ineq)}\ \frac{\mathsf{unify}(t_1, t_2) = \bot}{\langle t_1 \not\equiv t_2 \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

$$\text{(RS-C-Simp)}\ \frac{\begin{array}{c}\text{“}c(\vec{\alpha}) \mid \exists\vec{\beta}.\, C_G \vdash \nabla\vec{\gamma}.\, C_B\text{”} \in P \\ \vec{v} \subseteq \mathcal{V} \setminus V \qquad \vec{s} \subseteq \mathcal{S} \setminus S \qquad \theta = [\vec{t}/\vec{\alpha}, \vec{v}/\vec{\beta}, \vec{s}/\vec{\gamma}] \\ V' = V \cup \{\vec{v}\} \qquad\qquad S' = S \cup \{\vec{s}\} \\ \langle C \mid D; \mathcal{G}; \varphi; V', S'\rangle \Vdash C_G\theta\end{array}}{\langle c(\vec{t}) \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle (C_G \wedge C_B)\theta \wedge C \mid D; \mathcal{G}; \varphi; V', S'\rangle}$$

$$\text{(RS-C-NoSimp)}\ \frac{\begin{array}{c}\forall(\text{“}c(\vec{\alpha}) \mid \exists\vec{\beta}.\, C_G \vdash \nabla\vec{\gamma}.\, C_B\text{”} \in P). \\ \Big(\vec{v} \subseteq \mathcal{V} \setminus V \qquad \vec{s} \subseteq \mathcal{S} \setminus S \qquad \theta = [\vec{t}/\vec{\alpha}, \vec{v}/\vec{\beta}, \vec{s}/\vec{\gamma}] \\ V' = V \cup \{\vec{v}\} \qquad\qquad S' = S \cup \{\vec{s}\} \\ \langle C \mid D; \mathcal{G}; \varphi; V', S'\rangle \nVdash C_G\theta\Big)\end{array}}{\langle c(\vec{t}) \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle \mathbf{f} \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

Entailment

$$\longrightarrow^* \ \triangleq\ \text{transitive closure over } \longrightarrow$$

$$\langle D; \mathcal{G}; \varphi; V, S\rangle \Vdash C \ \triangleq\ \begin{cases} \mathcal{G} \Vdash \mathcal{G}' \wedge \varphi' \Vdash \varphi & \text{if } C' = \mathbf{t} \\ \bot & \text{if } C' \neq \mathbf{f} \end{cases}$$
$$\langle C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow^* \langle C' \mid D'; \mathcal{G}'; \varphi'; V', S'\rangle$$

Fig. 16. Operational Semantics of Statix (Constraints & Unification)

We conjecture that the $C \overset{l}{\hookrightarrow} s$ relation is sound, that is, given

$$\langle C|D; \mathcal{G}; \varphi; S, V\rangle \longrightarrow^* \langle C'|D'; \mathcal{G}'; \varphi'; S', V'\rangle \implies \mathcal{G} \sqsubseteq \mathcal{G}'$$

and

$$s \uparrow l \triangleq C \wedge D \overset{l}{\hookrightarrow} s \vee \exists v.\, C \wedge D \overset{l}{\hookrightarrow} v$$

Rewrite relation $\boxed{\langle C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle}$

$$\text{(RS-Edge)}\ \frac{\mathrm{fv}(s_1) \cup \mathrm{fv}(s_2) = \emptyset}{\langle s_1 \xrightarrow{l} s_2 \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G} \cup \{s_1 \xrightarrow{l} s_2\}; \varphi; V, S\rangle}$$

$$\text{(RS-Rel-1)}\ \frac{\begin{array}{c} r : \sigma_1 \times \cdots \times \sigma_i \to \sigma_j \times \cdots \times \sigma_n \qquad \mathrm{fv}(s) = \emptyset \qquad \mathrm{fv}(t_1) \cup \cdots \cup \mathrm{fv}(t_i) = \emptyset \\ s \xrightarrow{r}\!\blacksquare\ (t_1, \ldots, t_i, t'_j, \ldots, t'_n) \notin \mathcal{G} \qquad \mathcal{G}' = \mathcal{G} \cup \{s \xrightarrow{r}\!\blacksquare\ (t_1, \ldots, t_n)\} \end{array}}{\langle s \xrightarrow{r}\!\blacksquare\ (t_1, \ldots, t_n) \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle C \mid D; \mathcal{G}'; \varphi; V, S\rangle}$$

$$\text{(RS-Rel-N)}\ \frac{\begin{array}{c} r : \sigma_1 \times \cdots \times \sigma_i \to \sigma_j \times \cdots \times \sigma_n \qquad \mathrm{fv}(s) = \emptyset \qquad \mathrm{fv}(t_1) \cup \cdots \cup \mathrm{fv}(t_i) = \emptyset \\ s \xrightarrow{r}\!\blacksquare\ (t_1, \ldots, t_i, t'_j, \ldots, t'_n) \in \mathcal{G} \end{array}}{\langle s \xrightarrow{r}\!\blacksquare\ (t_1, \ldots, t_n) \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle t_j \equiv t'_j \wedge \cdots \wedge t_n \equiv t'_n \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

$$\text{(RS-Query)}\ \frac{\mathrm{fv}(s) = \emptyset \qquad \left[\!\!\left[\begin{array}{l} \mathcal{E} \triangleq re \qquad \mathrm{WF}(t) \triangleq \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle \Vdash c_1(t) \\ \mathcal{O} \triangleq ord \quad t_1 < t_2 \triangleq \langle C \mid D; \mathcal{G}; \varphi; V, S\rangle \Vdash c_2(t_1, t_2) \\ l \uparrow s \triangleq (C \wedge D \overset{l}{\hookrightarrow} s) \vee \exists v \in V.\, \left(C \wedge D \overset{l}{\hookrightarrow} v\right) \end{array} \right]\!\!\right] \textsc{Res}^r(s) = E}{\langle \mathbf{q}(re, c_1, ord, c_2) \twoheadrightarrow s \overset{r}{\mapsto} t \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle \longrightarrow \langle t \equiv E \wedge C \mid D; \mathcal{G}; \varphi; V, S\rangle}$$

Fig. 17. Operational Semantics of Statix (Scope Graph & Queries)

Scope extension relation

$$\text{(SE-Conj-1)}\ \frac{C_1 \overset{l}{\hookrightarrow} s}{C_1 \wedge C_2 \overset{l}{\hookrightarrow} s} \qquad \text{(SE-Conj-2)}\ \frac{C_2 \overset{l}{\hookrightarrow} s}{C_1 \wedge C_2 \overset{l}{\hookrightarrow} s}$$

$$\text{(SE-C-Head)}\ \frac{\text{“}c(\vec{\alpha}) \mid \exists \vec{\beta}.\, C_G \vdash \nabla \vec{\gamma}.\, C_B\text{”} \in P \qquad C_B[\vec{t}/\vec{\alpha}] \overset{l}{\hookrightarrow} s}{c(\vec{t}) \overset{l}{\hookrightarrow} s}$$

$$\text{(SE-C-Var)}\ \frac{\text{“}c(\vec{\alpha}) \mid \exists \vec{\beta}.\, C_G \vdash \nabla \vec{\gamma}.\, C_B\text{”} \in P \qquad C_B[\vec{t}/\vec{\alpha}] \overset{l}{\hookrightarrow} \beta_i}{c(\vec{t}) \overset{l}{\hookrightarrow} s}$$

$$\text{(SE-Edge)}\ \frac{}{s \xrightarrow{l} t \overset{l}{\hookrightarrow} s} \qquad \text{(SE-Rel)}\ \frac{}{s \xrightarrow{r}\!\blacksquare\ t \overset{r}{\hookrightarrow} s}$$

Fig. 18. Scope Extension

it holds that

$$\forall l, s.\, \neg(s \uparrow l) \implies \mathcal{G} \downarrow^l_s \mathcal{G}'$$

C SIMPLY-TYPED LAMBDA CALCULUS WITH RECORDS IN STATIX

In this appendix we define the simply-typed lambda calculus with structural records [Pierce 2002] in Statix.

```
module statics // Static semantics of STLCrec using scopes-as-types

signature
  sorts ID = string                          // $x$
  sorts BinOp                                // $op$

  sorts Exp constructors                     // $e$ :=
    Num     : int -> Exp                     //      | $n$
    Plus    : Exp * Exp -> Exp          //      | $e$ + $e$
    Fun     : ID * TypeExp * Exp -> Exp      //      | fun ($x$ : $te$) { $e$ }
    Var     : ID -> Exp                      //      | $x$
    App     : Exp * Exp -> Exp               //      | $e$ $e$
    Rec     : list(FldInit) -> Exp           //      | { $finit^\ast$ }
    FAccess : Exp * ID -> Exp                //      | $e$.$x$
    FExtend : Exp * list(FldInit) -> Exp     //      | { $e$ with $finit^\ast$ }
    TypeLet : ID * TypeExp * Exp -> Exp      //      | type $x$ = $te$ in $e$
    Let     : ID * Exp * Exp -> Exp          //      | type $x$ = $e$ in $e$
    TAS     : Exp * TypeExp -> Exp           //      | $e$ : $te$

  sorts FldInit constructors                 // $finit$ :=
    FldInit : ID * Exp -> FldInit            //          | $x$ = $e$

  sorts TypeExp constructors                 // $te$ :=
    NumType  : TypeExp                        //       | num
    FunType  : TypeExp * TypeExp -> TypeExp   //       | $te$ -> $te$
    RecType  : list(FldType) -> TypeExp       //       | { $ftype^\ast$ }
    ERecType : TypeExp * list(FldType) -> TypeExp      //       | $te$ with { $ftype^\ast$ }
    TypeRef  : ID -> TypeExp                  //       | $x$

  sorts FldType constructors                 // $ftype$ :=
    FldType : ID * TypeExp -> FldType        //          | $x$ : $te$

  sorts Type constructors
    NUM   : Type
    FUN   : Type * Type -> Type
    REC   : scope -> Type

  relations
    typeOfDecl : occurrence -> Type

  namespaces
    Var  : string
    Fld  : string
    Type : string
```

```
  name-resolution
    labels P R
    resolve Var filter pathMatch[P*] min pathLt[$ < P]
    resolve Fld filter pathMatch[R*] min pathLt[$ < R]

rules

  programOK : Exp
  programOK(e) :- {s T}
    new s,
    typeOfExp(s, e) == T.

rules

  typeOfExp : scope * Exp ->  Type

  typeOfExp(s, Num(_)) = NUM().

  typeOfExp(s, Plus(e1, e2)) = NUM() :-
    typeOfExp(s, e1) == NUM(),
    typeOfExp(s, e2) == NUM().

  typeOfExp(s, Fun(x, te, e)) = FUN(S, T) :- {s_fun}
    typeOfTypeExp(s, te) == S,
    new s_fun, s_fun -P-> s,
    s_fun -> Var{x@x} with typeOfDecl S,
    typeOfExp(s_fun, e) == T.

  typeOfExp(s, Var(x)) = T :-
    typeOfDecl of Var{x@x} in s |-> [(_, (_, T))].

  typeOfExp(s, App(e1, e2)) = T :- {S U}
    typeOfExp(s, e1) == FUN(S, T),
    typeOfExp(s, e2) == U,
    subType(U, S).

  typeOfExp(s, Rec(finits)) = REC(rs) :-
    new rs, fieldInitsOK(s, finits, rs).

  typeOfExp(s, FExtend(e, finits)) = REC(rs2) :- {rs1}
    typeOfExp(s, e) == REC(rs1),
    new rs2, rs2 -R-> rs1,
    fieldInitsOK(s, finits, rs2).

  typeOfExp(s, FAccess(e, x)) = T :- {rs d}
    typeOfExp(s, e) == REC(rs),
    typeOfDecl of Fld{x@x} in rs |-> [(_, (_, T))].
```

```
  typeOfExp(s, TypeLet(x, te, e)) = S :- {s_let}
    new s_let, s_let -P-> s,
    s_let -> Type{x@x} with typeOfDecl typeOfTypeExp(s, te),
    typeOfExp(s_let, e) == S.

  typeOfExp(s, Let(x, e1, e2)) = S :- {s_let}
    new s_let, s_let -P-> s,
    s_let -> Var{x@x} with typeOfDecl typeOfExp(s, e1),
    typeOfExp(s_let, e2) == S.

  typeOfExp(s, TAS(e, te)) = T :- {S}
    typeOfExp(s, e) == S,
    typeOfTypeExp(s, te) == T,
    subType(S, T).

rules

  fieldInitOK : scope * FldInit * scope
  fieldInitsOK maps fieldInitOK(*, list(*), *)

  fieldInitOK(s, FldInit(x, e), rs) :- {T}
    rs -> Fld{x@x} with typeOfDecl typeOfExp(s, e).

rules

  typeOfTypeExp : scope * TypeExp ->  Type

  typeOfTypeExp(s, NumType()) = NUM().

  typeOfTypeExp(s, FunType(te1, te2)) = FUN(typeOfTypeExp(s, te1), typeOfTypeExp(s, te2)).

  typeOfTypeExp(s, RecType(ftypes)) = REC(rs) :-
    new rs, fieldTypesOK(s, ftypes, rs).

  typeOfTypeExp(s, ERecType(te, ftypes)) = REC(rs2) :- {rs1}
    typeOfTypeExp(s, te) == REC(rs1),
    new rs2, rs2 -R-> rs1,
    fieldTypesOK(s, ftypes, rs2).

  typeOfTypeExp(s, TypeRef(x)) = T :-
    typeOfDecl of Type{x@x} in s |-> [(_, (_, T))].

rules

  fieldTypeOK : scope * FldType * scope
  fieldTypesOK maps fieldTypeOK(*, list(*), *)
```

```
  fieldTypeOK(s, FldType(x, te), rs) :-
    rs -> Fld{x@x} with typeOfDecl typeOfTypeExp(s, te).

rules

  subType  : Type * Type
  subField : scope * (path * occurrence)
  subFields maps subField(*, list(*))

  subType(NUM(), NUM()).

  subType(FUN(S1, T1), FUN(S2, T2)) :- subType(S2, S1), subType(T1, T2).

  subType(REC(s_sub), REC(s_sup)) :- subFields(s_sub, allFields(s_sup)).

  subField(s_sub, (p_sup, d_sup)) :- {S T}
    ?typeOfDecl[d_sup, S] in dst(p_sup),
    typeOfDecl of d_sup in s_sub |-> [(_, (_, T))],
    subType(T, S).

  allFields: scope -> list((path * occurrence))
  allFields(s) = ps :-
    query decl filter pathMatch[R*] and { Fld{_@_} }
                min pathLt[$ < R] and { Fld{x@_}, Fld{x@_} }
                in s |-> ps.
```

D SYSTEM F IN STATIX

In this appendix we define System F [Girard 1972; Reynolds 1974] in Statix.

```
module statics // Static semantics of System F using scopes-as-types

signature
  sorts ID = string

  sorts Exp constructors                    // $e$ :=
    Num     : ID -> Exp                     //      | $n$
    Fun     : ID * TypeExp * Exp -> Exp     //      | \($x$ : $te$) { $e$ }
    Var     : ID -> Exp                     //      | $x$
    App     : Exp * Exp -> Exp              //      | $e$ $e$
    TFun    : ID  * Exp -> Exp              //      | /\($X$) { $e$ }
    TApp    : Exp * TypeExp -> Exp          //      | $e$[$te$]
    Let     : ID  * Exp * Exp -> Exp        //      | let $x$ = $e$ in $e$
    LetT    : ID * TypeExp * Exp -> Exp     //      | let $x$ : $te$ = $e$ in $e$
    TAS     : Exp * TypeExp -> Exp          //      | $e$ : $t$

  sorts TypeExp constructors                // $te$ :=
    NumType : TypeExp                       //       | num
    FunType : TypeExp * TypeExp -> TypeExp  //       | $te$ -> $te$
    AllType : ID * TypeExp -> TypeExp       //       | $X$ => $te$
    VarType : ID -> TypeExp                 //       | $te$

  namespaces
    Var   : string
    TVar  : string

  sorts Type constructors // variables: T, S, U
    NUM   : Type
    FUN   : Type * Type -> Type
    ALL   : scope -> Type
    TVAR  : occurrence -> Type

    PROJ  : scope * occurrence -> Type
    PL    : scope -> Type

  relations
    typeOfDecl : occurrence -> Type

  name-resolution
    labels
      P // Parent (lexical)
      I // Instantiation

    resolve TVar  filter lexicalPathMatch min lexicalPathOrd
    resolve Var   filter lexicalPathMatch min lexicalPathOrd
```

```
rules // Reference resolution for type variables, variables

  lexicalPathMatch : list(label)
  lexicalPathOrd   : label * label

  lexicalPathMatch(p)    :- pathMatch[P*](p).
  lexicalPathOrd(p1, p2) :- pathLt[$ < P](p1, p2).

rules

  programOK : Exp
  programOK(e) :- {s T}
    new s,
    typeOfExp(s, e, T).

rules

  typeOfExp: scope * Exp -> Type

  typeOfExp(s, Num(_)) = NUM().

  typeOfExp(s, Fun(x, t_arg, e)) = FUN(U, T) :- {s_fun}
    typeOfTypeExp(s, t_arg) == U,
    new s_fun, s_fun -P-> s,
    s_fun -> Var{x@x} with typeOfDecl U,
    typeOfExp(s_fun, e) == T.

  typeOfExp(s, Var(x)) = T :-
    typeOfDecl of Var{x@x} in s |-> [(_, (_, T))].

  typeOfExp(s, App(e1, e2)) = T :- {S U}
    strict(typeOfExp(s, e1)) == FUN(S, T),
    typeOfExp(s, e2) == U,
    typeEq(S, U).

  typeOfExp(s, TFun(x, e)) = T :- {s_all d}
    new s_all, s_all -P-> s,
    d == TVar{x@x},
    s_all -> d with typeOfDecl TVAR(d),
    all(s_all, x, typeOfExp(s_all, e)) == T.

  typeOfExp(s, TApp(e, t)) = T :- {s_all}
    strict(typeOfExp(s, e)) == ALL(s_all),
    body(instWith(s_all, paramOf(s_all), typeOfTypeExp(s, t))) == T.

  typeOfExp(s, Let(x, e1, e2)) = S :- {s_let}
    new s_let, s_let -P-> s,
```

```
    s_let -> Var{x@x} with typeOfDecl typeOfExp(s, e1),
    typeOfExp(s_let, e2) == S.

  typeOfExp(s, LetT(x, t, e)) = S :- {s_let}
    new s_let, s_let -P-> s,
    s_let -> TVar{x@x} with typeOfDecl typeOfTypeExp(s, t),
    typeOfExp(s_let, e) == S.

  typeOfExp(s, TAS(e, t)) = T :- {S}
    typeOfExp(s, e) == S,
    typeOfTypeExp(s, t) == T,
    typeEq(S, T).

rules

  typeOfTypeExp: scope * TypeExp -> Type

  typeOfTypeExp(s, NumType()) = NUM().

  typeOfTypeExp(s, FunType(t1, t2)) = FUN(typeOfTypeExp(s, t1), typeOfTypeExp(s, t2)).

  typeOfTypeExp(s, AllType(x, t)) = T :- {s_all d}
    new s_all, s_all -P-> s,
    d == TVar{x@x},
    s_all -> d with typeOfDecl TVAR(d),
    T == all(s_all, x, typeOfTypeExp(s_all, t)).

  typeOfTypeExp(s, VarType(x)) = T :-
    typeOfDecl of TVar{x@x} in s |-> [(_, (_, T))].

rules

  typeEq: Type * Type

  typeEq(NUM(), NUM()).

  typeEq(FUN(T_arg, T_ret), FUN(S_arg, S_ret)) :-
    typeEq(T_arg, S_arg),
    typeEq(T_ret, S_ret).

  typeEq(S@PROJ(_, _), T) :- typeEq(strict(S), T).

  typeEq(S, T@PROJ(_, _)) | S != PROJ(_, _) :- typeEq(S, strict(T)).

  typeEq(PL(s1), PL(s2)) | s1 == s2.

  typeEq(ALL(s_all1), ALL(s_all2)) :- {d_fresh}
    new d_fresh,
```

```
    typeEq(
      body(instWith(s_all1, paramOf(s_all1), PL(d_fresh)))
    , body(instWith(s_all2, paramOf(s_all2), PL(d_fresh)))
    ).

  typeEq(TVAR(d1), TVAR(d2)) | d1 == d2. // Should not occur in programs with no free type variables


rules // Abbreviation for resolving a parameter via a reference

  paramOf: scope -> occurrence
  paramOf(s) = d :-
    query typeOfDecl filter instantiationPathMatch and { TVar{_@_} }
          in s |-> [(_, (_, TVAR(d)))].

rules

  namespace TBody :
  resolve TBody filter instantiationPathMatch min instantiationPathOrd

  body: scope -> Type
  body(s) = PROJ(s, TBody{@-}).

  all: scope * ID * Type -> Type
  all(s_all, x, T) = ALL(s_all) :-
    !typeOfDecl[TBody{@x}, T] in s_all.

rules // Reference resolution for body

  instantiationPathMatch : list(label)
  instantiationPathOrd   : label * label

  instantiationPathMatch(p)    :- pathMatch[I*](p).
  instantiationPathOrd(p1, p2) :- pathLt[I < $](p1, p1). // longest path

////////////////////////////////
// SUBSTITUTION & NORMALIZATION //
////////////////////////////////

rules

  relation subst : occurrence -> Type

  instWith: scope * occurrence * Type -> scope
  instWith(s, d, T) = s_inst :-
    new s_inst, s_inst -I-> s, !subst[d, T] in s_inst.

rules // forces normalization of a postponed projection
```

```
  strict : Type -> Type

  strict(PROJ(s, r)) = T :- {p S}
    typeOfDecl of r in s |-> [(p, (_, S))],
    norm(scopes(p), S) == T.

  strict(T) = T' | T != PROJ(_, _) :- T == T'.

rules // type normalization

  norm: list(scope) * Type -> Type
  norm([], T) = T.
  norm(ss@[_|_], T) = normR(reverseScopes(ss), T).

  normR: list(scope) * Type -> Type

  normR([], T) = T.

  normR([_|_], NUM()) = NUM().

  normR(ss@[_|_], FUN(S, T)) = FUN(normR(ss, S), normR(ss, T)).

  normR([s|ss], TVAR(d_tvar)) = normVar(s, ss, d_tvar).

  normR(ss@[_|_], S@PROJ(_, _)) = normR(ss, strict(S)).

  normR([s|ss], ALL(s_all)) = normAll(s, ss, s_all).

  normR([_|_], T@PL(_)) = T.


  normAll: scope * list(scope) * scope -> Type
  normAll2: list((path * (occurrence * Type))) * list(scope) * scope -> Type

  normAll(s, ss, s_all) = U :- {PDTs}
    query subst filter { p :- pathMatch[e](p) }
                in s |-> PDTs,
    normAll2(PDTs, ss, s_all, U).

  normAll2([(_, (d, S))], ss, s_all) = normR(ss, ALL(instWith(s_all, d, S))).
  normAll2([], ss, s_all) = normR(ss, ALL(s_all)).


    // assumes that there is only one substitution in each scope;
    // safe for single-argument type binders in System F
    // add an instantiation scope; i.e., delay the substitution; no expansion
```

```
  normVar: scope * list(scope) * occurrence -> Type
  normVar2: list(Type) * list(scope) * occurrence -> Type

  normVar(s, ss, d_tvar) = U :- {Ts}
    substV(s, d_tvar, Ts),
    normVar2(Ts, ss, d_tvar, U).

  normVar2([T], ss, d_tvar) = normR(ss, T).
  normVar2([], ss, d_tvar) = normR(ss, TVAR(d_tvar)).

rules // reverse list of scopes

  reverseScopes : list(scope) -> list(scope)
  reverseScopes(ss) = reverseScopesR(ss, []).

  reverseScopesR : list(scope) * list(scope) -> list(scope)
  reverseScopesR([], ss) = ss.
  reverseScopesR([s|ss], ss') = reverseScopesR(ss, [s|ss']).

rules // substitution

  substV : scope * occurrence -> list(Type)

  substV(s, d) = substPs2Vs(ps) :-
    query subst filter { p :- pathMatch[e](p) } and { d' :- d' == d }
                 in s |-> ps.

  substP2V : (path * (occurrence * Type)) -> Type
  substPs2Vs maps substP2V(list(*)) = list(*)

  substP2V((_, (_, T))) = T' :- T == T'.
```

E FEITHERWEIGHT GENERIC JAVA IN STATIX

In this appendix we define Featherweight Generic Java [Igarashi et al. 2001] in Statix.

```
module statics // Static semantics of Featherweight Generic Java

signature

  sorts ID = string

  sorts Program constructors                                                    // prog :=
    Program : list(ClassDecl) -> Program                                        //       | L*
    Let     : list(ClassDecl) * Exp -> Program                                  //       | let L* in e

  sorts ClassDecl constructors                                                  // L :=
    ClassDecl : ID * list(TVarDecl) * /*N*/TypeExp *                            //    | class C<V*> <
                list(FieldDecl) * CtorDecl * list(MethodDecl) -> ClassDecl

  sorts TVarDecl constructors                                                   // V :=
    TVarDecl : ID * /*N*/TypeExp -> TVarDecl                                    //    | X <: N

  sorts FieldDecl constructors                                                  // F :=
    FieldDecl : TypeExp * string -> FieldDecl                                   //    | T f;

  sorts CtorDecl constructors                                                   // K :=
    CtorDecl : ID * list(Param) * list(Exp) * list(FieldInit) -> CtorDecl       //    | C(P*) { super

  sorts FieldInit constructors                                                  // finit :=
    FieldInit : ID * Exp -> FieldInit                                           //        | this.f = e

  sorts MethodDecl constructors                                                 // M :=
    MethodDecl : list(TVarDecl) * TypeExp * ID *                                //    | <V*> T m(P*)
                 list(Param) * Exp -> MethodDecl

  sorts Param constructors                                                      // P :=
    Param : TypeExp * ID -> Param                                               //    | T x

  sorts Exp constructors                                                        // e :=
    Var     : ID -> Exp                                                       //    | x
    Fld     : Exp * ID -> Exp                                                 //    | e.f
    Call    : Exp * list(TypeExp) * ID * list(Exp) -> Exp                     //    | e.<T*>m(e*)
    New     : /*N*/TypeExp * list(Exp) -> Exp                                 //    | new N(e*)
    Cast    : TypeExp * Exp -> Exp                                            //    | (T)e
    Ascribe : Exp * TypeExp -> Exp                                            //    | e:T

  sorts TypeExp constructors                                                    // T :=
    ClassT : ID * list(TypeExp) -> /*N*/TypeExp                                 //    | C<T*>
    TVar   : ID -> TypeExp                                                      //    | X
```

```
  sorts TYPE constructors
    TVAR   : occurrence -> TYPE
    CLASS  : scope -> TYPE
    CTOR   : list(TYPE) -> TYPE
    METHOD : scope -> TYPE
    MTY    : TYPE * list(TYPE) -> TYPE

    PROJ   : scope * occurrence -> TYPE // delayed projection

  namespaces
    Var        : string
    TVar       : string
    Field      : string
    Method     : string
    MType      :
    Ctor       :
    Class      : string

  relations
    typeOfDecl     : occurrence -> TYPE
    tparamsOfClass : -> list(occurrence)

  name-resolution
    labels P    // lexical parent
           S    // super class
           I    // instantiation

    resolve Class  filter lexicalPathMatch min lexicalPathOrd
    resolve TVar   filter lexicalPathMatch min lexicalPathOrd
    resolve Var    filter lexicalPathMatch min lexicalPathOrd
    resolve Ctor   filter pathMatch[I*]    min subtypePathOrd
    resolve MType  filter pathMatch[I*]    min subtypePathOrd
    resolve Field  filter subtypePathMatch min subtypePathOrd
    resolve Method filter subtypePathMatch min subtypePathOrd

rules

  lexicalPathMatch : list(label)
  lexicalPathOrd   : label * label

  lexicalPathMatch(p)    :- pathMatch[I*P*](p).
  lexicalPathOrd(p1, p2) :- pathLt[$ < P, $ < I](p1, p2).

rules

  programOK : Program

  programOK(Program(cdecls)) :- {s}
```

```
    new s,
    object(s),
    classesOK(s, cdecls).

  programOK(Let(cdecls, e)) :- {s}
    new s,
    object(s),
    classesOK(s, cdecls),
    strict(typeOfExp(s, e)) == _.


  object : scope

  object(s) :- {d_class s_class s_ctor}
    // class
    d_class == Class{"Object"@-},
    new s_class, s_class -P-> s,
    s -> d_class with typeOfDecl CLASS(s_class),
    // type parameters
    !tparamsOfClass[[]] in s_class,
    // constructor
    new s_ctor, s_ctor -P-> s_class,
    s_class -> Ctor{@-} with typeOfDecl CTOR([]).

rules

  classOK : scope * ClassDecl
  classesOK maps classOK(*, list(*))

  classOK(s, ClassDecl(x, tvars, te_super, fields, ctor, methods)) :- {d_class s_class Xs T_super}
    d_class == Class{x@x},
    new s_class, s_class -P-> s,                  // class scope
    s -> d_class with typeOfDecl CLASS(s_class),  // class declaration and type
    classTParamsOK(s_class, tvars) == Xs,
    classSuperOK(s_class, te_super),
    fieldDeclsOK(s_class, fields),                // assumed to be distinct
    ctorOK(s_class, d_class, ctor),
    methodsOK(s_class, methods).                  // assumed to be distinct

  classTParamsOK: scope * list(TVarDecl) -> list(occurrence)

  classTParamsOK(s_class, tvars) = Xs :-
    typesOfTParams(s_class, tvars) == Xs,
    !tparamsOfClass[Xs] in s_class.

  classSuperOK : scope * TypeExp

  classSuperOK(s_class, ClassT(x, tes)) :- {r p d s_super Xs Ts Us}
```

```
    typeOfDecl of Class{x@x} in s_class |-> [(_, (_, CLASS(s_super)))], // type of super
    typesOfTypeExps(s_class, tes) == Ts,        // type arguments
    s_class -S-> s_super,                       // connect to super scope
    ?tparamsOfClass[Xs] in s_super,             // type parameters
    instWithA(s_class, Xs, Ts),                 // super class type variable instantiation
    boundsOfTParams(s_super, Xs) == Us,         // super class type variable bounds
    subTypes(s_class, Ts, Us),                  // bounds subtype validity
    notExtends(s_super, s_class).               // prevent cyclic inheritance

rules

  fieldDeclOK : scope * FieldDecl
  fieldDeclsOK maps fieldDeclOK(*, list(*))

  fieldDeclOK(s_class, FieldDecl(ty, x)) :- {T}
    typeOfTypeExp(s_class, ty) == T,            // field type
    s_class -> Field{x@x} with typeOfDecl T.    // field declaration

rules

  ctorOK : scope * occurrence * CtorDecl

  ctorOK(s_class, Class{x@_}, CtorDecl(y, params, es, finits)) :- {s_ctor s_super d_super CT Us Ts Ss}
    x == y,                                     // constructor name corresponds to class name
    new s_ctor, s_ctor -P-> s_class,            // ctor body scope
    typesOfParams(s_ctor, params) == Ts,        // ctor param types
    s_class -> Ctor{@x} with typeOfDecl CTOR(Ts),    // ctor declaration
    superClassCtorType(s_class) == CTOR(Ss),    // constructor type with instantiated type variab
    typesOfExps(s_ctor, es) == Us,              // argument types
    subTypes(s_ctor, Us, Ss),                   // valid arguments
    initsOK(s_ctor, finits).

rules

  initOK : scope * FieldInit
  initsOK maps initOK(*, list(*))

  initOK(s, FieldInit(x, e)) :- {T U}
    query typeOfDecl filter lexicalPathMatch and { Field{x'@_} :- x' == x }
          in s |-> [(_, (_, T))],               // field type
    typeOfExp(s, e) == U,                       // expr type
    subType(s, U, T).

rules

  methodOK : scope * MethodDecl
  methodsOK maps methodOK(*, list(*))
```

```
relation tparamsOfMethod : -> list(occurrence)

methodOK(s_class, MethodDecl(tvars, ty, x, params, e)) :- {d s_method Xs T Ts U s_mtype}
  d == Method{x@x},                          // method declaration
  !decl[d] in s_class,

  new s_method, s_method -P-> s_class,       // scope of method body

  typesOfTParams(s_method, tvars) == Xs,     // type parameters

  typesOfParams(s_method, params) == Ts,     // method parameters
  methodThisOK(s_method, s_class, d),        // declare this
  typeOfTypeExp(s_method, ty) == T,          // return type
  typeOfExp(s_method, e) == U,               // body type
  subType(s_method, U, T),                   // body type is subtype of return type

  new s_mtype,
  s_mtype -> MType{@x} with typeOfDecl MTY(T, Ts),    //
  !tparamsOfMethod[Xs] in s_mtype,                    // type parameters
  !typeOfDecl[d, METHOD(s_mtype)] in s_class,         // method type

  overrideOK(s_method, d, T, Ts).

methodThisOK : scope * scope * occurrence

methodThisOK(s_method, s_class, d@Method{_@x}) :- {s_inst}
  new s_inst, s_inst -I-> s_class,
  s_method -> Var{"this"@x} with typeOfDecl CLASS(s_inst).

overrideOK  : scope * occurrence * TYPE * list(TYPE)
overrideOK2 : scope * list((path * occurrence)) * TYPE * list(TYPE)

overrideOK(s, d@Method{x@_}, T, Ts) :- {ps}
  overrides(s, d, ps), overrideOK2(s, ps, T, Ts).

overrideOK2(s, [], T, Ts).

overrideOK2(s, [(_, d')], T, Ts) :- {U Us MT s_mtype}
  strict(PROJ(s, d')) == METHOD(s_mtype),
  strict(PROJ(s_mtype, MType{@-})) == MTY(T, Ts),
  subType(s, U, T),
  typesEq(Us, Ts).

overrides : scope * occurrence -> list((path * occurrence))
overrides(s_class, Method{x@_}) = ps :-
  query decl filter pathMatch[S S*] and { d :- d == Method{x@_} }
            min subtypePathOrd
            in s_class |-> ps.
```

```
rules

  typeOfParam:  scope * Param -> TYPE
  typesOfParams maps typeOfParam(*, list(*)) = list(*)

  typeOfParam(s, Param(ty, x)) = T :- {d}
    typeOfTypeExp(s, ty) == T,          // param type
    s -> Var{x@x} with typeOfDecl T.    // param declaration

rules

  typeOfTParam : scope * TVarDecl -> occurrence
  typesOfTParams maps typeOfTParam(*, list(*)) = list(*)

  typeOfTParam(s, TVarDecl(x, ty)) = d :-
    d == TVar{x@x},
    s -> d with typeOfDecl typeOfTypeExp(s, ty). // t is upper bound of type parameter

  boundsOfTParams: scope * list(occurrence) -> list(TYPE)
  boundsOfTParams(s, []) = [].
  boundsOfTParams(s, [X|Xs]) = [PROJ(s, X) | boundsOfTParams(s, Xs)].

  promoteType: scope * TYPE -> TYPE
  promoteTypes maps promoteType(*, list(*)) = list(*)

  promoteType(s, TVAR(d)) = strict(PROJ(s, d)).    // bound projection
  promoteType(s, T)       = T | T != TVAR(_).

rules

  typeOfTypeExp : scope * TypeExp -> TYPE
  typesOfTypeExps maps typeOfTypeExp(*, list(*)) = list(*)

  typeOfTypeExp(s, TVar(x)) = TVAR(d) :-
    TVar{x@x} in s |-> [(_, d)].            // type var reference

  typeOfTypeExp(s, ClassT(x, tes)) = CLASS(s_inst) :- {s_class Xs Ts Us}
    typeOfDecl of Class{x@x} in s |-> [(_, (_, CLASS(s_class)))],    // class type
    typesOfTypeExps(s, tes) == Ts,          // type arguments
    ?tparamsOfClass[Xs] in s_class,         // type parameters
    instWith(s_class, Xs, Ts) == s_inst,    // super class type variable instantiation
    boundsOfTParams(s_inst, Xs) == Us,      // super class type variable bounds
    subTypes(s, Ts, Us).                    // bounds subtype validity

rules

  typeOfExp : scope * Exp -> TYPE
```

```
  typesOfExps maps typeOfExp(*, list(*)) = list(*)

  typeOfExp(s, Var(x)) = T :- {r p d}
    typeOfDecl of Var{x@x} in s |-> [(p, (d, T))].

  typeOfExp(s, Fld(e, x)) = S :- {s_inst}
    promoteType(s, strict(typeOfExp(s, e))) == CLASS(s_inst),    // receiver type
    PROJ(s_inst, Field{x@x}) == S.                               // field reference

  typeOfExp(s, Call(e, tys, x, es)) = U :- {S s_cls Ts m_inst Xs s_mtype s_minst Us Ss p}
    promoteType(s, strict(typeOfExp(s, e))) == CLASS(s_cls),
    typeOfDecl of Method{x@x} in s_cls |-> [(p, (_, METHOD(s_mtype)))],
    typesOfTypeExps(s, tys) == Ts,
    ?tparamsOfMethod[Xs] in s_mtype,
    instWith(s_mtype, Xs, Ts) == s_minst,
    norm(scopes(p), PROJ(s_minst, MType{@-})) == MTY(U, Us),
    typesOfExps(s, es) == Ss,
    subTypes(s, Ss, Us).

  typeOfExp(s, New(te, es)) = T :- {s_inst Ts Us}
    strict(typeOfTypeExp(s, te)) == T@CLASS(s_inst),  // class type scope
    strict(PROJ(s_inst, Ctor{@-})) == CTOR(Ts),       // project constructor type
    typesOfExps(s, es) == Us,                         // argument types
    subTypes(s, Us, Ts).

  typeOfExp(s, Ascribe(e, te)) = T :- {S}
    strict(typeOfExp(s, e)) == S,
    strict(typeOfTypeExp(s, te)) == T,
    subType(s, S, T).

  typeOfExp(s, Cast(te, e)) = T_as :- {s_as s_act}
    promoteType(s, strict(typeOfTypeExp(s, te))) == CLASS(s_as),    // cast type
    promoteType(s, strict(typeOfExp(s, e))) == CLASS(s_act),        // expression type
    castOK(s_act, s_as).

  castOK : scope * scope
  castOK2 : list(path) * list(path)

  castOK(s_act, s_as) :- {p1 p2}
    extendsQ(s_act, s_as, p1),
    extendsQ(s_as, s_act, p2),
    castOK2(p1, p2).

  castOK2([_], [_]).
  castOK2([_], []).
  castOK2([], [_]).

rules
```

```
  typesEq maps typeEq(list(*), list(*))

  typeEq : TYPE * TYPE

  typeEq(TVAR(d), TVAR(d)).

  // two class types are equal if
  // - they are instantiations of the same class as identified by its scope
  // - the instantiations of the type parameter of the class are equal

  typeEq(CLASS(s1), CLASS(s2)) :- {p1 d1 p2 d2}
    classScope(s1) == p1,          // sub class scope
    classScope(s2) == p2,          // super class scope
    classInstEq(p1, p2).

  typeEq(T@PROJ(_, _), S) :- typeEq(strict(T), S).
  typeEq(T, S@PROJ(_, _)) | T != PROJ(_, _) :- typeEq(T, strict(S)).

rules

  // subType(s, S, T) : S is a subtype of T wrt scope s

  subType : scope * TYPE * TYPE
  subTypeA : scope * TYPE * TYPE
  subTypes maps subType(*, list(*), list(*))

  subType(s, S, T) :- {S' T'}
    strict(S, S'), strict(T, T'),
    subTypeA(s, S', T').

  subTypeA(s, T@TVAR(_), U) | U == T.

  subTypeA(s, T@TVAR(d), U) | U != T :-
    subType(s, PROJ(s, d), U).

  subTypeA(s, CLASS(s1), CLASS(s2)) :- {p1 p2 Xs Ts Us}
    classScope(s2) == p2,          // super class scope
    extends(s1, dst(p2)) == p1,    // subtype instance of super type
    classInstEq(p1, p2).

rules

  classScope : scope -> path // path to class corresponding to instance
  classScope(s_inst) = p :-
    query () filter pathMatch[I*]
              min pathLt[I < $] and true // longest path, shadow shorter paths
              in s_inst |-> [p].
```

```
  extends    : scope * scope -> path
  notExtends : scope * scope
  extendsQ   : scope * scope -> list(path)

  extends(s, s_class) = p :- extendsQ(s, s_class, [p]).
  notExtends(s, s_class)  :- extendsQ(s, s_class, []).

  extendsQ(s, s_class) = ps :-
    query () filter subtypePathMatch and { s :- s == s_class }
             min subtypePathOrd
             in s |-> ps.

  subtypePathMatch : list(label)
  subtypePathOrd   : label * label

  subtypePathMatch(p)    :- pathMatch[I* S*](p).
  subtypePathOrd(p1, p2) :- pathLt[$ < I, $ < S](p1, p2).

  classInstEq : path * path
  classInstEq(p1, p2) :- {s_class Xs TVs Ts Us}
    s_class@dst(p1) == dst(p2),
    ?tparamsOfClass[Xs] in s_class,        // type parameters of super type
    declsToTVARs(Xs, TVs),
    norms(scopes(p1), TVs) == Ts,          // normalize type params w.r.t. subtype
    norms(scopes(p2), TVs) == Us,          // normalize type params w.r.t. super type
    typesEq(Ts, Us).                       // equal argument types


rules // abbreviation

  superClassCtorType : scope -> TYPE

  superClassCtorType(s) = norm(scopes(p), T) :-
    query typeOfDecl filter pathMatch[S] and { d :- d == Ctor{@_} }
          in s |-> [(p, (_, T))].

  declToTVAR : occurrence ->  TYPE
  declsToTVARs maps declToTVAR(list(*)) = list(*)

  declToTVAR(d) = TVAR(d).

////////////////////////////////
// SUBSTITUTION & NORMALIZATION //
////////////////////////////////

rules // forces normalization of a postponed projection
```

```
  strict : TYPE -> TYPE
  strict(PROJ(s, r)) = T :- {p d S}
    typeOfDecl of r in s |-> [(p, (d, S))],
    norm(scopes(p), S) == T.

  strict(T) = T | T != PROJ(_, _).

rules // normalize types

  norm: list(scope) * TYPE -> TYPE
  norms maps norm(*, list(*)) = list(*)

  norm([], T) = T.
  norm(ss@[_|_], T) = normR(reverseScopes(ss), T).

  normR: list(scope) * TYPE ->  TYPE
  normsR maps normR(*, list(*)) = list(*)

  normR([], T) = T.

  normR([s|ss], METHOD(s_mtype)) = T :- {ps Xs Ts}
    query subst filter pathMatch[e] in s |-> ps,
    unzipPDTs(ps) == (Xs, Ts),
    normR(ss, METHOD(instWith(s_mtype, Xs, Ts))) == T.

  normR(ss@[_|_], MTY(T, Ts)) = MTY(normR(ss, T), normsR(ss, Ts)).

  normR([s|ss], TVAR(d_tvar)) = normVar(s, ss, d_tvar).

  normR(ss@[_|_], S@PROJ(_, _)) = normR(ss, strict(S)).

  normR([s|ss], CLASS(s1)) = T :- {ps Xs Ts}
    query subst filter pathMatch[e] in s |-> ps,
    unzipPDTs(ps) == (Xs, Ts),
    normR(ss, CLASS(instWith(s1, Xs, Ts))) == T.

  normR(ss@[_|_], CTOR(Ts)) = CTOR(normsR(ss, Ts)).

  normVar  : scope * list(scope) * occurrence ->  TYPE
  normVar2 : scope * list(scope) * occurrence * list(TYPE) ->  TYPE

  normVar(s, ss, d_tvar) = U :- {Ts}
    hasSubst(s, d_tvar) == Ts, normVar2(s, ss, d_tvar, Ts) == U.

  normVar2(s, ss, d_tvar, [T]) = normR(ss, T).
  normVar2(s, ss, d_tvar, [])  = normR(ss, TVAR(d_tvar)).

rules // reverse list of scopes
```

```
  reverseScopes: list(scope) -> list(scope)
  reverseScopes(ss) = reverseScopesR(ss, []).

  reverseScopesR: list(scope) * list(scope) -> list(scope)
  reverseScopesR([], ss)      = ss.
  reverseScopesR([s|ss], ss') = reverseScopesR(ss, [s|ss']).

rules // unzip path+declaration+type pairs

  unzipPDTs: list((path * (occurrence * TYPE))) -> list(occurrence) * list(TYPE)

  unzipPDTs([])                 = ([], []).
  unzipPDTs([(_, (X, T))|PDTs]) = ([X|Xs], [T|Ts]) :-
    unzipPDTs(PDTs) == (Xs, Ts).

rules

  relation subst : occurrence -> TYPE

  instWith : scope * list(occurrence) * list(TYPE) -> scope
  instWith(s, [], []) = s.
  instWith(s, Xs@[_|_], Ts@[_|_]) = s_inst :-
    new s_inst, s_inst -I-> s, instWithA(s_inst, Xs, Ts).

  instWithA: scope * list(occurrence) * list(TYPE)
  instWithA(_, [],     []).
  instWithA(s, [X|Xs], [T|Ts]) :- !subst[X, T] in s, instWithA(s, Xs, Ts).

rules // substitution

  hasSubst: scope * occurrence -> list(TYPE)

  hasSubst(s, d) = substValues(ps) :-
    query subst filter pathMatch[e] and { d' :- d' == d } in s |-> ps.

  substValue  : (path * (occurrence * TYPE)) -> TYPE
  substValues maps substValue(list(*)) = list(*)
  substValue((_, (_, T))) = T' :- T == T'.
```