Scopes as Types

A scope graph captures the binding structure of a program. A scope is a region in a program that behaves uniformly with respect to name resolution. Declarations of names and references are associated with scopes. Visibility is modeled by edges between scopes. A generic, language-independent resolution algorithm interprets a scope graph to resolve references to declarations by finding the most specific well-formed path in a scope graph. To express the binding rules of a programming language, one defines a mapping from abstract syntax trees to scope graphs. Scope graphs cover a wide range of binding structures, including lexical bindings1 such as let bindings, function parameters, and local variables in blocks; and non-lexical bindings such as (potentially cyclic) module imports and class inheritance. The framework enables language- independent definitions of alpha equivalence and safe variable renaming.

The paper makes the following technical contributions (os que me interessam):
• We show that viewing scopes as types enables modeling the internal structure of types in a
range of interesting type systems, including structural records and generic classes, using the
generic representation of scopes.
• We extend the scope graph framework of Néron et al. [2015] and Van Antwerpen et al. [2016]
with scoped relations to model the association of types with declarations and the representation of explicit substitutions in the instantiation of parameterized types. We generalize name resolution from resolution of references to general queries for scoped relations. Furthermore, visibility policies, which were global (per language), can be defined per query, enabling namespace-specific visibility policies. We simplify the framework by not including imports as a primitive, since these can be encoded using the scopes-as-types approach.
• We extend the visual notation of scope graph diagrams with scoped relations, which provides a useful language to explain patterns of names and types in programming languages.

A scope graph consists of scopes, connected by edges, containing data. A labeled edge s1 l s2 between scopes s1 and s2 determines that the declarations in scope s2 are reachable from scope s1. The label can be used to regulate visibility. A scoped datum s r d associates a data term d with a scope s under relation r . For example, we will use s : (x , T), to represent a declaration of name x in the scope s with type T , and use x : T to denote the pair. There may be multiple data items associated with a scope under the same relation.
Given this structure, we can now precisely characterize name resolution for a reference as finding a path from its scope to a scope with a matching declaration. This intuition is formally captured by the resolution calculus in the third part of Fig. 1, which is parameterized by well-formedness and visibility

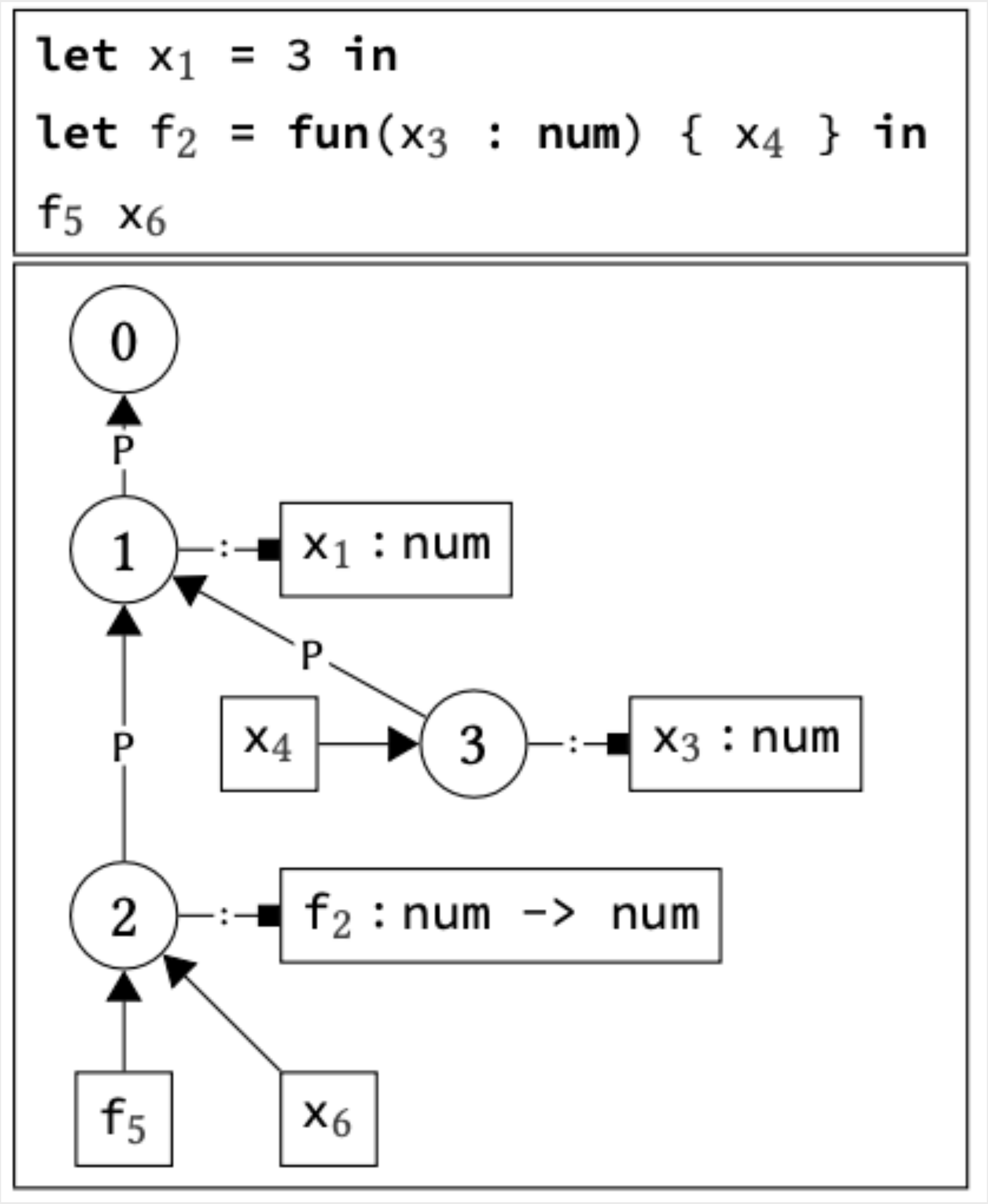
parameters defined in the second part of Fig. 1. We discuss the judgments of the calculus.

The judgment $G \vdash p:s1 \twoheadrightarrow s2$ states that there is a path p from scopes1 to scopes2, if there is a sequence of labeled scope edges starting at s1 and leading to s2. Cyclic paths are not admitted: the $s1 < scopes(p)$ premise of (NR-Cons) asserts that scope s1 does not occur in path p. The path p records the scopes and edge labels that it passes through.

Scoping is modeled by extension of an environment with a new pair, which shadows any earlier declarations of the same name (either by removing a matching pair or through definition of the lookup function). The extended environment is only used for those sub-expressions where the binding is in scope. Scope graphs make the shadowing rules explicit by separating the construction of the binding structure and the definition of resolution.

Scopes are depicted by circles labeled with a number, and edges between scopes are depicted as labeled edges l .

Scope #0 in the scope graph in Fig. 2 is the scope of the context of the outer let. Scopes #1 and #2 are the scopes of the first and second let, respectively. Scope #3 is the scope of the function literal. The scopes are connected via P-labeled edges to their lexical parent scope (thus P is for parent). Declarations are depicted as boxes associated with scopes via an : edge going from a scope to a declaration. Lastly, references are depicted as boxes connected to scopes by edges going from the reference to the scope.



```
let x1 = 3 in
let f2 = fun(x3 : num) { x4 } in
f5 x6
```

Identifying Record Types

In nominal type systems, types are identified by name. Information about the type is associated with that name.For example, with scope graphs we can states td (Point,rPoint), which associates with the type name Point some representation rPoint of the record type. A record type can then be represented as REC(Point) referring to the declaration of the type by its name. Such a representation is efficient since copying the type entails copying a reference to its representation. Furthermore, a type is directly related to its origin in a program.

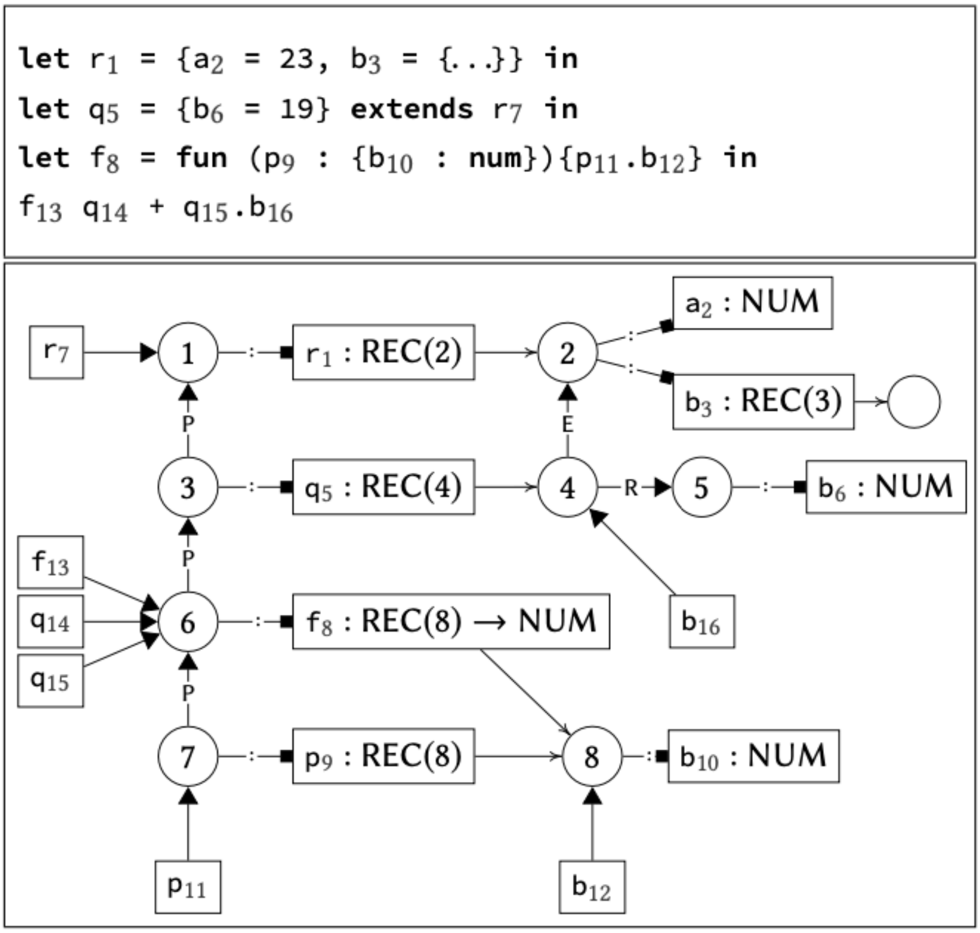
Representing Record Types

With scope graphs, we do not need a new representation: scopes provide a natural representation for record types.

To realize a structural type system, we use the scope reference itself as a type, and represent a record type as REC(sr). A difference with the traditional representation of structural types as association lists is that scopes have identity.

Since scopes are not part of the surface syntax of types, Fig. 4 defines two notions of types: **syntactic types and semantic types** for use in typing rules. Fig. 4 defines a relation ⊢ JtK ⇛ T that relates a syntactic type t to a corresponding semantic type T . In particular, the (T-Rec) rule defines how a syntactic record type is related to a scope with a declaration for each field in the record type. We use the vector notation x¯ to denote sequences and point-wise application. The mapping from syntactic to semantic types is used in the (ERS-Fun) rule (not shown) to convert the syntactic type annotation on the formal parameter. The (ERS-Rec) rule asserts that a record literal is typed by a scope that has a declaration for each field name in the list, inferring the type from the initialization expression. In the (T-Rec) and (ERS-Rec) rules we have omitted the assertion that field names of record types need to be unique. This can be expressed with a scope graph query that requires that a field name reference in the record scope resolves to a single declaration.

Composing Record Types



```
let r1 = {a2 = 23, b3 = {..}} in
let q5 = {b6 = 19} extends r7 in
let f8 = fun (p9 : {b10 : num}){p11.b12} in
f13 q14 + q15.b16
```

Fig. 6. A program with records and functions

Extensions are represented as edges that preserve the structure of the substitutions being merged.

Accessing Record Types

The first premise of rule (ERS-Access)requires the expression e to have a record type REC(sr).
The second premise resolves the field x relative to the scope sr of that type using the resolution query DECL(xi)
Path well-formedness is given by a regular expression stating that resolution may follow any path via R and E edges. Record fields can also be accessed using plain variables due to the **with** form.

Comparing Record Types

The type of the actual parameter is a subtype of the type of the formal parameter or that same type.

Summary. A crucial difference between scope graphs and association lists is that association lists represent an eager name shadowing policy (applied before doing name resolution), while scope graphs support a lazy name shadowing policy (applied during name resolution). The scopes as types approach scales to type systems with binding patterns that go beyond lambdas and records, including type systems for languages with classes; association lists alone do not.

Class Tables

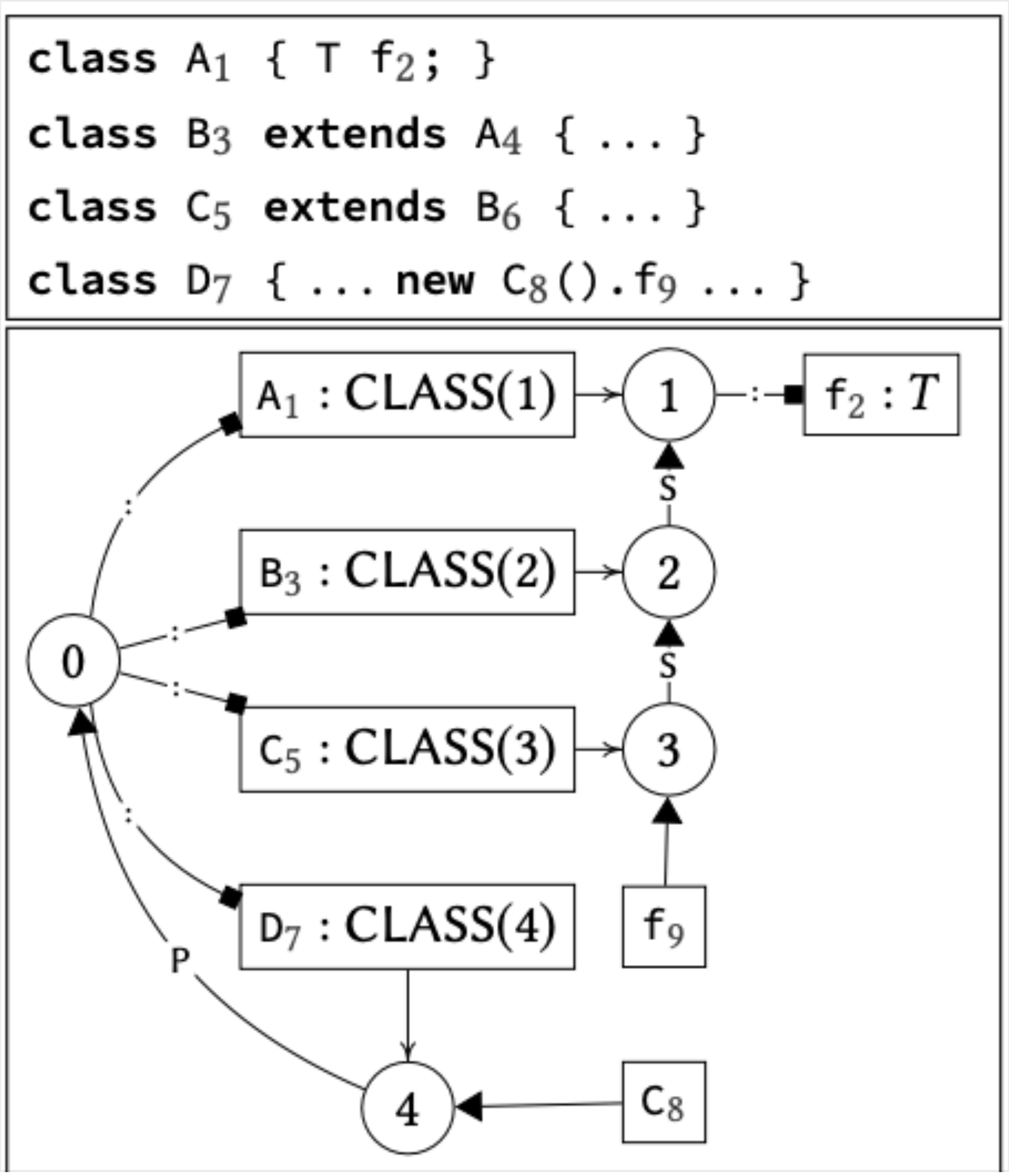
The original presentation of FJ relies on various data structures for name resolution, notably class tables, type contexts, and the AST of classes themselves. Names are mapped to class definitions via the class table. In turn, the class table is used in auxiliary relations that define how to retrieve association lists of names and types for class members, by traversing the AST of classes. Thus, classes are used as a data structure since they are not reducible to a simple association list representation. But the AST of FJ programs is not an ideal data structure for reuse to define name resolution for other languages with nominal subtyping. For such languages we would have to re-specify similar auxiliary relations to do name resolution using a different AST. We show how the definition of a class table data structure is subsumed by the use of scope graphs.

Syntactic and Semantic Types

FJ has a single kind of syntactic type, namely class names ranged over by C. The corresponding semantic type of a class name C is an INST(s) type where s is the scope of the class declared as C. The (T-Class) rule in Fig. 8 translates a syntactic type to a semantic type by resolving the name in the lexical context by following a sequence of P edges. The łrootž scope is similar to a class table: it binds all class declarations that a program defines and is a dominating lexical context for all classes in a program. Whereas INST(s) represents an instance of the class identified by scope s in the scope graph, the class type CLASS(s) represents the definition of the class s, and is the type of declarations in the łrootž scope.

Class Typing

The structure of a class is reflected in the scope graph. The (FJ-Class) rule declares the name of a class (C) as being typed by the scope that defines it (sc) in the łrootž scope of a program (s). The rule omits the assertion that field and record names are unique in a class.
The (FJ-FldK) rule asserts that fields and constructors are associated with class scopes, where the constructor parameter types are recorded using the relation.
The (FJ-Method) rule asserts that well-typed methods are associated with the class scope, and that overriding methods have the same type signature as the overridden methods in super classes.



```
class A1 { T f2; }
class B3 extends A4 { ... }
class C5 extends B6 { ... }
class D7 { ... new C8().f9 ... }
```

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 (FJ-Class).

Expression Typing

The (FJ-Var) rule matches paths that either traverse a sequence of lexical parent edges, which makes formal parameters of methods as well as local fields reachable, or traverse a sequence of super edges which makes fields in

super classes reachable.
The (FJ-New) rule for **new** expressions dereferences the constructor method of a class by resolving the K relation in the class scope sc;ε denotes the empty regular expression, which matches a 0-step path.

Subtyping

Nominal subtyping allows the use of a sub-class in the place of any of its super classes: if A is a super type of B, then B can be used anywhere an A is expected. The scope graph affords a straightforward characterization of this subtype relationship: any class member declaration that is reachable from the class scope of A is also reachable from the inheriting class scope of B, because their scopes are connected via an S edge. In other words, using scopes as types lets us define nominal subtyping as path connectedness in a scope graph.

Parametric Polymorphism (isto já é de mais pra minha cabecinha, não vou fazer)

Characterizes types that are parameterized by other types and that can be instantiated by substitution.

Name Resolution Algorithm

The algorithm essentially implements an ordered depth-first search in the scope graph. The well-formedness predicate WFL is used to control depth, and the label order <l is used to control breadth and cut-off of the search. Cyclic paths are also disallowed so the algorithm is terminating.

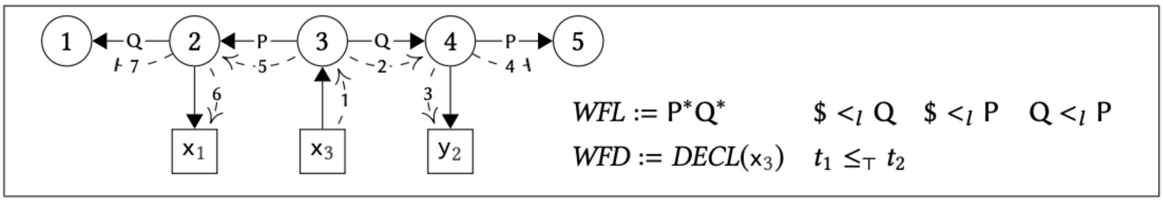


Fig. 16. Name resolution example. The dashed arrows visualize the search in the graph, and the label numbers indicate the search order.

The parameters we show are for resolving the reference x3. Edges have labels P and Q. Path well-formedness WFL states that well-formed paths cannot follow Q edges after P edges. Data well-formedness WFD matches declarations with the same name x as the reference. The label order <l prefers Q over P, and local declarations over both. Finally, the data order ≤⊤ states that declarations via more specific paths always shadow declarations reachable via less specific paths.