# A Theory of Name Resolution

Descreve uma teoria independente de linguagem para name binding and resolution (Resumidamente é a teoria que servirá de pilar para a minha tese uma vez que a metodologia por eles usada vai servir como base para a minha implementação).

Eles dividem o problema em 2 fases:

- Construção de um grafo de scope usando as regras da linguagem de uma árvore de sintaxe
- Depois são feitas as verificações do grafo usando um processo de resolução independente de linguagem

Esta abordagem colmata problemas da maior parte das linguagens que usam processos repetitivos e por vezes ambíguos para as suas resoluções de nomes (manipulation of a symbol table in a compiler, a use-to-definition display in an IDE, or a substitution function in a mechanized soundness proof).

## SCOPE GRAPH

**Node :**

- Name references
- Declarations
- Scopes

**Edges :**

- References to scopes
- Declarations to scopes
- Scopes to “parent” scopes

**Capacidades da teoria :**

- O cálculo de resolução especifica como construir um caminho através do gráfico a partir de uma referência a uma declaração, o que corresponde a uma possível resolução da referência. Referências ambíguas correspondem naturalmente a vários caminhos de resolução a partir do mesmo nó de referência; referências não resolvidas correspondem à ausência de caminhos de resolução.
- A teoria desenvolvida suporta também “imports”.

  “The calculus supports complex import patterns including transitive and cyclic import of scopes.“

- Para qualquer linguagem, a construção pode ser especificada por uma definição convencional dirigida por sintaxe sobre a gramática da linguagem.
- Um algoritmo prático para calcular ambientes estáticos convencionais que mapeiam identificadores de limite para os locais AST das

declarações correspondentes, que podem ser usados para implementar uma função de resolução determinística e de terminação que é consistente com o cálculo.

O paper divide-se em 6 capítulos que se complementam para desenvolver e provar a teoria em questão:

- **Scope Graph and Resolution Calculus**: We introduce a language-independent framework to capture the relations among references, declarations, scopes, and imports in a program. We give a declarative specification of the resolution of references to declarations by means of a calculus that defines resolution paths in a scope graph .
- **Variants**: We illustrate the modularity of our core framework design by decribing several variants that support more complex binding schemes.
- **Coverage**: We show that the framework covers interesting name binding patterns in existing languages, including various flavors of let bindings, qualified names, and inheritance in Java.
- **Scope graph construction**: We show how scope graphs can be constructed for arbitrary programs in a simple example language via straightforward syntax-directed traversal.
- **Resolution algorithm**: We define a deterministic and terminating resolution algorithm based on the construction of binding environments, and prove that it is sound and complete with respect to the calculus.
- **α-equivalence and renaming**: We define a language-independent characteriation of α-equivalence of programs, and use it to define a notion of valid renaming.

## Scope Graphs and resolution paths

A scope graph is obtained by a language-specific mapping from the abstract syntax tree of a program.The mapping col- lapses all abstract syntax tree nodes that behave uniformly with respect to name resolution into a single ‘scope’ node in the scope graph.

**Toy language used:**

- Lambda and mu: The functional abstractions **fun** and **fix** represent lambda and mu terms, respectively; both have basic unary lexically scoped bindings.

- Let: The various flavors of let bindings(**sequentiallet**, **letrec** and **letpar**) challenge the unary lexical binding model.

- Definition: A definition (**def**) declares a variable and binds it to the

value of an initializing expression. The definitions in a module are not ordered (no requirement for ‘def-before-use’), giving rise to mutually recursive definitions.

- Qualified names: Elements of modules can be addressed by means of a qualified name using conventional dot notation.

- Imports: All declarations in an imported module are made visible without the need for qualification.

- Transitive imports: The definitions imported into an imported module are themselves visible in the importing module.

- Cyclic imports: Modules can (indirectly) mutually import each other, leading to cyclic import chains.

- Nested modules: Modules may have sub-modules, which can be accessed using dot notation or by importing the containing module.

**References and declarations**

- $x_i^{\mathsf{D}}{:}S$: declaration with name $x$ at position $i$ and optional associated named scope $S$
- $x_i^{\mathsf{R}}$: reference with name $x$ at position $i$

**Scope graph**

- $\mathcal{G}$: scope graph
- $\mathcal{S}(\mathcal{G})$: scopes $S$ in $\mathcal{G}$
- $\mathcal{D}(S)$: declarations $x_i^{\mathsf{D}}{:}S'$ in $S$
- $\mathcal{R}(S)$: references $x_i^{\mathsf{R}}$ in $S$
- $\mathcal{I}(S)$: imports $x_i^{\mathsf{R}}$ in $S$
- $\mathcal{P}(S)$: parent scope of $S$

**Well-formedness properties**

- $\mathcal{P}(S)$ is a partial function
- The parent relation is well-founded
- Each $x_i^{\mathsf{R}}$ and $x_i^{\mathsf{D}}$ appears in exactly one scope $S$

**Fig. 1.** Scope graphs

**Resolution paths**

$$s := \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{I}(x_i^{\mathsf{R}}, x_j^{\mathsf{D}}{:}S) \mid \mathbf{P}$$
$$p := [] \mid s \mid p \cdot p$$
(inductively generated)
$$[] \cdot p = p \cdot [] = p$$
$$(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$$

**Well-formed paths**

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_, \_)^*$$

**Specificity ordering on paths**

$$\frac{}{\mathbf{D}(\_) < \mathbf{I}(\_, \_)} \quad (DI)$$

$$\frac{}{\mathbf{I}(\_, \_) < \mathbf{P}} \quad (IP)$$

$$\frac{}{\mathbf{D}(\_) < \mathbf{P}} \quad (DP)$$

$$\frac{s_1 < s_2}{s_1 \cdot p_1 < s_2 \cdot p_2} \quad (Lex1)$$

$$\frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2} \quad (Lex2)$$

**Fig. 2.** Resolution paths, well-formedness predicate, and specificity ordering

**Edges in scope graph**

$$\frac{\mathcal{P}(S_1) = S_2}{\mathbb{I} \vdash \mathbf{P} : S_1 \longrightarrow S_2} \quad (P)$$

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2} \quad (I)$$

**Transitive closure**

$$\frac{}{\mathbb{I} \vdash [] : A \twoheadrightarrow A} \quad (N)$$

$$\frac{\mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I} \vdash p : B \twoheadrightarrow C}{\mathbb{I} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T)$$

**Reachable declarations**

$$\frac{x_i^{\mathsf{D}} \in \mathcal{D}(S') \quad \mathbb{I} \vdash p : S \twoheadrightarrow S' \quad WF(p)}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad (R)$$

**Visible declarations**

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \qquad \forall j, p'(\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}} \quad (V)$$

**Reference resolution**

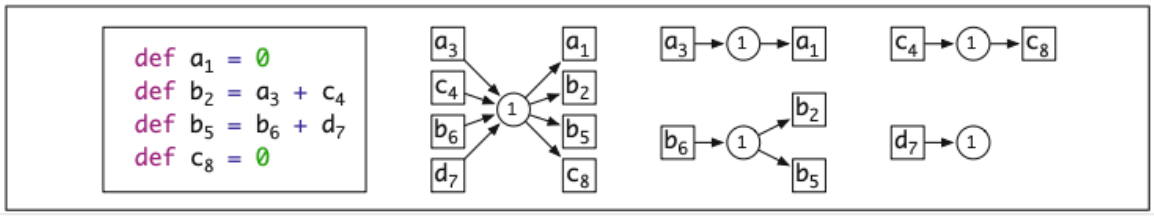
$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X)$$

**Fig. 3.** Resolution calculus

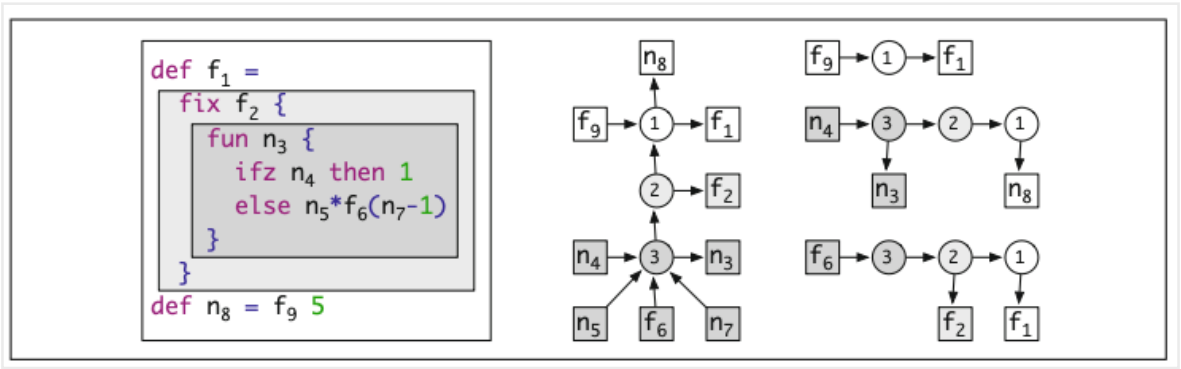
Exemplo de uso:

```
program = decl*
   decl = module id { decl* } | import qid | def id = exp
    exp = qid | fun id { exp } | fix id { exp }
        | let bind* in exp | letrec bind* in exp | letpar bind* in exp
        | exp exp | exp ⊕ exp | int
    qid = id | id . qid
   bind = id = exp
```

Name resolution is specified by a relation –→ ⊆ R(G) × D(G) between references and corresponding declarations in G.
A scope is an abstraction over a group of nodes in the abstract syntax tree that behave uniformly with respect to name resolution. Each program has a scope graph G, whose nodes are a finite set of scopes S(G). Every program has at least one scope, the global or root scope. Each scope S has an associated finite set D(S) of declarations and finite set R(S) of references (at particular program positions), and each declaration and reference in a program belongs to a unique scope. A scope is the atomic grouping for name resolution: roughly speaking, each reference xRi in a scope resolves to a declaration of the same variable xDj in the scope, if one exists. Intuitively, a single scope corresponds to a group of mutually recursive definitions, e.g., a **letrec** block, the declarations in a module, or the set of top-level bindings in a program. Below we will see that edges between nodes in a scope graph determine visibility of declarations in one scope from references in another scope.




We model lexical scope by means of the parent relation on scopes. In a well-formed scope graph, each scope has at most one parent and the parent relation is well-founded. Formally, the partial function P(_) maps a scope S to its parent scope P(S). Given a scope graph with parent relation we can define the notion of reachable and visible declarations in a scope.



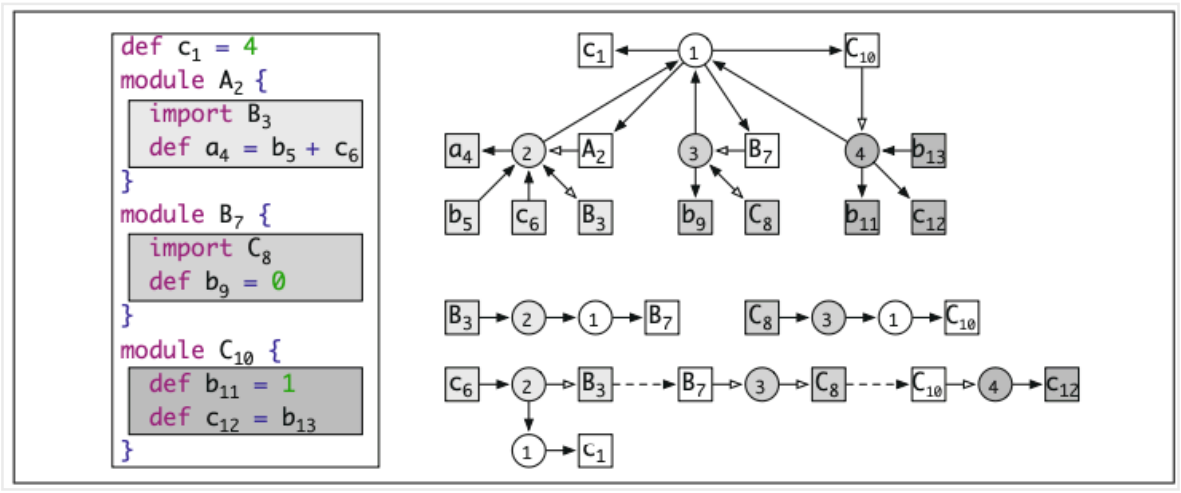

**Reachability:** That is, xRi in scope S1 can be resolved to xDj in scope S2, if S2 is reachable from S1, i.e. if S1  S2.
**Visibility:** Under lexical scoping, multiple possible resolutions are not problematic, as long as the declarations reached are not declared in the same scope. A declaration is visible unless it is shadowed by a declaration that is 'closer by'.

Now that we have defined the notions of reachability and visibility, we can give a more precise description of the sense in which scopes "behave uniformly" with respect to resolution. For every scope S:
– Each declaration in the program is either visible at every reference in R(S) or not visible at any reference in R(S).

– For each reference in the program, either every declaration in D(S) is reachable from that reference, or no declaration in D(S) is reachable from that reference.

– Every declaration in D(S) is visible at every reference in R(S).

**Imports:** We model an import by means of a reference xRi in the set of imports I(S) of a scope S.This associated named scope represents the declarations introduced by, and encapsulated in, the module.




In other words when importing a module, we import not just its declarations, but all declarations in its lexical context. This behavior seems undesirable; to our knowledge, no real languages exhibit it. To rule out such resolutions, we define a well-formedness predicate WF(p) that requires paths p to be of the form P∗ ·I(_, _)∗, i.e. forbidding the use of parent steps after one or more import steps. We use this predicate to restrict the reachable declarations relation by only considering scopes reach- able through a well-formed path.

To rule out this kind of behavior we extend the calculus to **keep track of the set of seen imports** <- !impedir rever imports!

An import can only be used once at the end of the chain of scopes.
To handle both **include** and ordinary imports, we can once again differentiate the references, and define different ordering rules depending on the reference used in the import step.

## Scope Graph construction

The traversal is specified as a collection of (potentially) mutually recursive functions, one or more for each syntactic class of LM. Each function f is defined by a set of clauses [pattern]fargs. When f is invoked on a term, the clause whose pattern matches the term is executed. Functions may also take additional arguments args. Each clause body consists of a sequence of statements separated by semicolons. Functions can optionally return a value using ret(). The let statement binds a metavariable in the remainder of the clause body. An empty clause body is written ().
The algorithm is initiated by invoking [_]prog on an entire LM program. Its net effect is to produce a scope graph via a sequence of imperative operations. The construct newP creates a new scope S with parent P (or no parent if p =⊥) and empty sets D(S), R(S), and I(S). These sets are subsequently populated using the += operator, which extends a set imperatively. The program scope graph is simply the set of scopes that have been created and populated when

the traversal terminates.

## Resolution Algorithm

For us, an environment is just a set of declarations xDi . This can be thought of as a function from identifiers to (possible empty) sets of declaration positions. We construct an atomic environment corresponding to the declarations in each scope, and then combine atomic environments to describe the sets of reachable and visible declarations resulting from the parent and import relations. The key operator for combining environments is shadowing, which returns the union of the declarations in two environments restricted so that if a variable x has any declarations in the first environment, no declarations of x are included from the second envionment.
The algorithm computes sets of declarations rather than full derivation paths, so it does not maintain enough information to delay the visibility computation.

**Termination:** The algorithm is terminating using the well-founded lexicographic measure (|R(G) \ I|, |S(G) \ S|). Termination is straightforward by unfolding the calls to Res in EnvI and then inlining the definitions of EnvV and EnvL: this gives an equivalent algorithm in which the measure strictly decreases at every recursive call.

**Correction:**

- Cannot have cycles
- All names must have a scope path (reachable and visible)

Free variables. The ~ equivalence classes corresponding to free variables x also contain the artificial position $x^{-}$. Since the equivalence classes of two equivalent programs P1 and P2 have to be exactly the same, every element equivalent to $x^{-}$ (i.e. a free reference) in P1 is also equivalent to $x^{-}$ in P2. Therefore the free references of α-equivalent programs have to be identical.

Duplicate declarations. The definition allows us to also capture α-equivalence of programs with duplicate declarations. Assume that a reference xRi1 resolves to two definitions xDi2 and xDi3 ; then i1, i2 and i3 belong to the same equivalence class. Thus all α-equivalent programs will have the same ambiguities.