# Towards a Theory of Type-Safe Renaming and Refactoring (Draft Paper)

Casper Bach[1] [0000-0003-0622-7639], Luka Miljak[2] [0009-0000-3882-7722], and Rosilde Corvino[3] [0000-0003-1311-8027]

[1] University of Southern Denmark, Odense, Denmark
[2] Delft University of Technology, Delft, The Netherlands
[3] TNO-ESI, Eindhoven, The Netherlands

## 1 Introduction

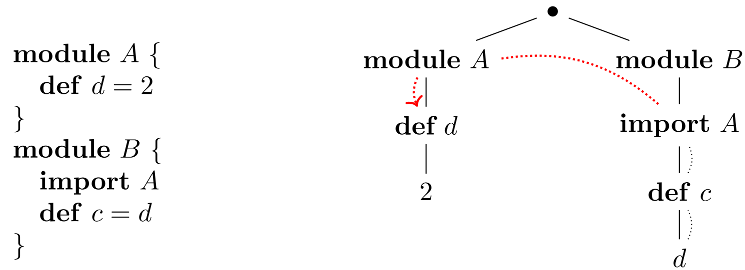
Modern software is continuously maintained and refactored. But as code bases grow, manually maintaining and refactoring code becomes challenging. An attractive solution to this problem is to automatically refactor code, using automated transformation tools. For example, modern IDEs provide built-in support for common refactorings; and dedicated transformation languages offer *semantic pattern matching* capabilities [5, 6, 19].

Such automated transformation tools should ideally provide correctness guarantees, such as guaranteeing program well-boundness and well-typedness. However, such guarantees are rare. The binding model used by automated transformation tools is rarely specified, and many tools are known to not preserve program well-boundness and well-typedness. For code bases where it is crucial that transformations only alter code in well-defined ways, and where fixing type errors throughout a large code base is untenable, this is unsatisfactory.

It is theoretically well-understood how to verify that transformations preserve well-typedness in languages with *lexical scoping* (e.g., lambda- and let-binding). The problem is that modern languages generally provide features for "programming in the large" that have a *non-lexical scoping*; e.g., modules, classes, traits, etc. The binding model behind such features is less commonly formalized.

By lexical scoping, we understand a name resolution semantics where names resolve to bindings that are lexical parents of the name in the AST; e.g.:

Modules or classes have non-lexical scoping since they typically resolve names to bindings that may not be lexical parents of the name in the AST; e.g.:




In this extended abstract, we propose a type-theoretic framework for defining and verifying type-safe automated transformations for languages with non-lexical binding. Our approach is based on the observation that type- and binding-preserving transformations require fine-grained information about the scoping structure that each AST node constructs, and about which other nodes rely on that scoping structure. Based on this observation, we present an approach to defining typing rules that precisely characterize the scoping structure that each AST node in a program constructs and relies on.

This extended abstract gives an overview of the approach, the type system of a core calculus for a language with modules and a functional core that uses it, and describes a type-safe *renaming transformation* that avoids name capture by synthesizing *minimal qualifiers* for affected names. For example, consider the program with two modules, $X$ and $Y$, in fig. 1. Say we want to rename the definition of $r$ to $s$ in module $X$ (left). Naively applying this renaming in module $Y$ (right) would rename $r$ to $s$ in the right-hand side of **def** $t$. However, this causes name capture: the **def** $s$ in $Y$ shadows the renamed definition in $X$. A better solution is to rename and *qualify* named references to avoid name capture, as shown in fig. 2. We call $X.s$ a *qualified name*, with $X$ being the *qualifier* of the name $s$. The qualifier $X$ is *minimal* since it is the *least* qualifier that preserves the name binding semantics of the original program. A selling point of our framework is that it provides a semantics of such *least qualification*.

The name capture discussed above could also be avoided by renaming the **def** $s$ in $Y$ to some other name. This approach is taken by, e.g., *name-fix* [2]. However, since names communicate intent, it is desirable that renaming transformations preserve existing names insofar as possible. The renaming transformation we consider renames and resolves name capture conflicts by synthesizing minimal qualifiers, when possible.

**Fig. 1.** An example program with two modules, $X$ (left) and $Y$ (right)

```
module X {                    module Y {
  def s = 3                     import X
}                               def s = 4
                                def t = X.s
                              }
```

**Fig. 2.** A renamed version of the program from fig. 1

The techniques we explore in this extended abstract are conceptually based on *scope graphs* [1, 10, 13], with some adaptations. Our contributions are:

- A type and scope discipline for non-lexical name binding, which lets us reason about and verify type- and binding-preserving transformations. We demonstrate the approach on a module language with a functional core.
- A safe-by-construction renaming transformation based on the approach.

The paper is structured as follows. Section 2 gives an overview of our approach, which we present formally in section 3. Then, in section 4 we present a type-safe renaming transformation. Section 5 describes related work.

# 2 Overview: Representing and Transforming Name Binding

Our approach relies on a binding model that precisely characterizes the scoping structure that each AST node constructs and relies on. For languages with non-lexical scoping, this structure typically forms a *graph*–i.e., a *scope graph* [1, 10, 13]. This section illustrates how, and illustrates their use for type-safe renaming. Our notion of scope graph is based on existing work, with some adaptations that we describe in this section.

## 2.1 Scope Graph of an Example Program with Modules

Figure 3 shows the scoping structure of the program from fig. 1. The nodes in the graph represent scopes: $\bullet$ is the root scope, 0 is the scope of module $X$, 1 of module $Y$, and $2, 3, 4$ are the scopes of each definition in the program.

Names are resolved by traversing (labeled) edges in the graph. For example, the dotted edges in fig. 3 represent a *resolution path*; i.e., a witness that the name $r$ in the right-hand side of **def** $t$ in fig. 1 resolves to the binding in module $Y$ along the edge sequence 4-1-0-2.

Each edge in the graph encodes a scoping reachability relation. Unlabeled edges represent lexical parent relationships: 0 and 1 are lexical children of the root scope, and the definition sub-scopes (2, 3, and 4) are lexical children of the module scopes they reside in. The direction of the unlabeled arrows encodes that bindings in lexical parent scopes can be reached from lexical child scopes (but not necessarily the other way around).

The **mod** $X$ and **mod** $Y$ edges in the graph represent the declarations of the two modules in the root scope. Each of these edges connect the root scope to a module scope, such that bindings in module scopes can be reached from the root scope via the declared name of the module.

The $r, s, t$ : **Int** edges represent bindings of names of type **Int**, corresponding to the members of each module scope.

### 2.2 Name Reachability and Visibility

In the scope graph in fig. 3, scopes can be reached reachable via multiple different paths. Only some of these paths are semantically well-formed. Furthermore, paths are ordered, and only the *least* paths are *visible*; other paths are *shadowed*.

**Reachability and Path Well-Formedness** Following existing work on scope graphs, we define well-formed paths using a regular expression over path labels. However, whereas existing work encodes bindings as data associated with nodes in the graph, our notion of scope graph encodes bindings in labels. For that reason, we augment our notion of regular expression to incorporate existential quantification. Using this, a well-formed path in our module language is a path that satisfies the following regular expression, for some name $x$ and type $T$:

$$\textsc{lex}^* \cdot \mathbf{imp}^? \cdot (\exists X.\, \mathbf{mod}\ X)^* \cdot (x : T)$$

Here LEX represents lexical (i.e., unlabeled) edges in the graph. Thus the regular expression says that a well-formed path may follow a sequence of lexical parent edges, followed by at most one import edge, followed by a series of module edges (representing *qualifiers*), followed by a binder. For example, the path 4-1-0-2 in fig. 3 is well-formed according to this regular expression, whereas the path 4-1-•-1-0-2 is not because it follows an **imp** edge after a **mod** edge.

While the examples discussed so far have not contained nested modules, our module language allows modules to be nested. For example, in the program with

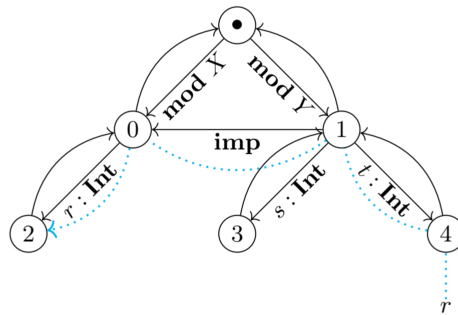



**Fig. 3.** Scoping structure of example program from fig. 1

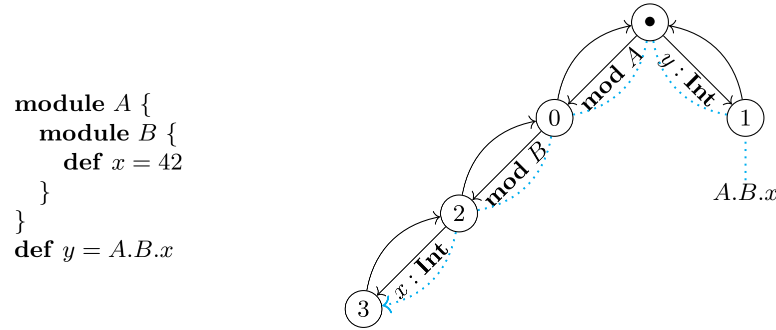



**Fig. 4.** An example program with nested modules and its scope graph

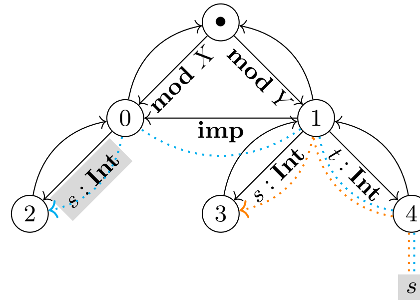



**Fig. 5.** Scoping structure of example program from fig. 1 with $r$ naively renamed to $s$

nested modules in fig. 4 we can reach $x$ from the root scope via the qualifier $A.B$ and the dotted path shown in the scope graph.

**Path Ordering and Visibility** Programs may have multiple well-formed paths that resolve to different binders. For example, fig. 5 shows the scope graph of the program from fig. 1 where we have renamed **def** $r$ to $s$ and naively substituted the name $r$ by $s$ in the right-hand side of **def** $t$. In this scope graph, there are multiple valid resolution paths for $s$, such as the paths shown in the figure: 4-1-0-2 and 4-1-3. In this case, we must decide which path is preferred; i.e., which path is *visible*. To this end, we use the following partial path order. We prefer names with fewer qualifiers. In case of names with the same number of qualifiers, we prefer names from imports over names from the lexical context, as given by the following path ordering. Let $p$ and $p'$ be the path prefixes of the paths that lead to two (qualified) names with an equal number of qualifiers. The path $p$ is shorter than $p'$ if (a) the labels of $p$ is a prefix of the labels of $p'$; or (b) the labels of $p$ and $p'$ have a common prefix up until a step where $p$ follows an import edge and $p'$ follows a lexical parent edge.

Comparing the two paths shown in fig. 2 using this path order, the path 4-1-3 is shorter, since the path prefix 4-1 is a prefix of 4-1-0.

### 2.3 Renaming Transformation

Let us now turn to the question of how to specify and implement the renaming transformation illustrated in fig. 2. Informally, for a transformation that renames a definition $x : T$ to $y : T$, we approach this question as follows:

(I) Relabel the $x : T$ edge in the graph with $y : T$ instead.
(II) Identify potentially affected paths in the original program; e.g., paths whose final step is an $x : T$ edge or a $y : T'$ edge, for some $T'$.
(III) For each potentially affected name identified in step (II), check if there exists a new least path in the relabeled graph from the source to the target scope. These recomputed least paths correspond to the least qualified names that preserve the binding semantics of the original program.

Applying these steps to fig. 5, we identify the path 4-1-3 in step (II) as potentially-affected. Then, in step (III), we find the new least path that leads to scope 2, namely 4-1-•-0-2. This path corresponds to the qualified name $X.s$.

This illustrates our approach to modeling name binding and typing for both lexical and non-lexical scoping. That leaves the question of how this model arises and relates to programs. We tackle this question next.

## 3 A Module Calculus and its Type and Scope Discipline

This section presents a type and scope discipline for a module calculus.

### 3.1 Scope Graphs and Fragments

Scope graphs and fragments are triples $\langle \zeta; \epsilon; \psi \rangle$ where $\zeta$ is a set of scopes, $\epsilon$ a set of labeled edges between scopes in $\zeta$, and $\psi$ a set of *paths* that follow labeled edges in $\epsilon$. A *scope fragment* is a subset of the nodes, edges, and paths in a *scope graph*. The scope graph of a program is given by the (disjoint) union of the scope fragments each AST node constructs. The typing rules of the module calculus we discuss in section 3.2 witness what scope fragments each AST node and its children construct, using the following notion of typing:

$$\underbrace{G}_{\substack{\text{Scope graph of} \\ \text{entire program}}} \vdash \underbrace{e}_{\substack{\text{AST node (e.g.,} \\ \text{expression)} \\ \text{being typed}}} : \underbrace{T}_{\text{Type}} \dashv \underbrace{F}_{\substack{\text{Scope fragment} \\ \text{that } e \text{ constructs} \\ \text{and relies on}}}$$

Figure 6 defines the syntax of scope graphs and fragments. The figure uses the notational conventions discussed below. We consider scope graphs with only integer- and function types, and with the labels we discussed in section 2. Besides the meta-variables $\zeta$, $\epsilon$, and $\psi$ discussed already, $\ell$ ranges over labels, $w$ over

$$T : \textit{Type} ::= \textbf{Int} \mid T \to T \qquad s : \textit{Scope} \qquad \zeta : \textit{Scopes} ::= \overline{s}$$

$$\ell : \textit{Label} ::= x : T \mid \textbf{mod}\ X \mid \textbf{imp} \mid \textsc{lex} \qquad \epsilon : \textit{Edges} ::= \overline{s \xrightarrow{\ell} s} \qquad w : \textit{Word} ::= \overline{\ell}$$

$$R : \textit{RegEx} ::= \ell \mid R^* \mid R \cdot R \mid R + R \mid \exists x.\, R \qquad p : \textit{Path} ::= \epsilon \cdot s \qquad \psi : \textit{Paths} ::= \overline{p}$$

$$F, G : \textit{Fragment} ::= \langle \zeta ; \epsilon ; \psi \rangle$$

**Fig. 6.** Syntax of scope graphs

$$\textsc{WF-Path-Empty}\quad \frac{}{\epsilon \vdash [] \cdot s : s \xrightarrow{[]} s}$$

$$\textsc{WF-Path-Step}\quad \frac{s_1 \xrightarrow{\ell} s_2 \in \epsilon \qquad \epsilon \vdash \epsilon_p \cdot s_3 : s_2 \xrightarrow{w} s_3}{\epsilon \vdash (s_1 \xrightarrow{\ell} s_2, \epsilon_p) \cdot s_3 : s_1 \xrightarrow{\ell, w} s_3}$$

$$\textsc{WF-Fragment}\quad \frac{\begin{array}{c}\zeta' \subseteq \zeta \qquad \epsilon' \subseteq \epsilon \qquad \psi' \subseteq \psi \\ \left(\forall (s_1 \xrightarrow{\ell} s_2) \in \epsilon'.\ s_1 \in \zeta \wedge s_2 \in \zeta\right) \qquad \left(\forall p \in \psi'.\ \epsilon \vdash p : \mathsf{src}\ p \xrightarrow{\mathsf{word}\ p} \mathsf{tgt}\ p\right)\end{array}}{\langle \zeta ; \epsilon ; \psi \rangle \vDash \langle \zeta' ; \epsilon' ; \psi' \rangle}$$

**Fig. 7.** Well-formed paths and well-formed fragments

words given by a sequence of labels, $R$ over regular expressions augmented with existential quantification, $p$ over paths given by a target scope and a sequence of edges, and $F$ and $G$ over fragments. By convention, we use $G$ for scope *graphs* (i.e., the disjoint union of all fragments that a program constructs), and $F$ for scope *fragments*.

Figure 7 characterizes well-formed paths and fragments. A well-formed path is a sequence of connected edges. A well-formed fragment is a sub-part of the scope graph where each fragment edge connects scopes in the graph and each fragment path is well-formed. The projection functions $\mathsf{src}$, $\mathsf{tgt}$, and $\mathsf{word}$ in WF-Fragment are for the source scope, target scope, and label sequence of a path. Path ordering is defined in appendix B.

*Notational Conventions.* Here and throughout, we use $\overline{x}$ to represent a sequence of $x$es. ',' appends sequences and [] is the empty sequence. We overload notation and write $x_1, \overline{x}_2$ for the sequence with $x_1$ prepended to $\overline{x}_2$, and $\overline{x}_1, x_2$ for $x_2$ appended to $\overline{x}_1$. $[x]$ is the singleton sequence, $[x, y]$ two-element sequences, etc. $x \sqcup y$ represents a *disjoint union* of two sequences. That is, $\overline{x} \sqcup \overline{y}$ is a permutation of $\overline{x}, \overline{y}$ where no two elements in $\overline{x}$ and $\overline{y}$ are identical.

### 3.2 Syntax and Typing Rules of a Module Calculus

The syntax of our module calculus is:

$$\begin{aligned}
a : AExpr &::= e^s \\
e : Expr &::= n \mid \rho \mid a\ a \mid \lambda x.\, a \\
m : ADecl &::= d^s \\
d : Decl &::= \mathbf{def}\ x : T = a \mid \mathbf{mod}\ X\ \{\ \overline{m}\ \} \mid \mathbf{import}\ \rho \\
\rho : ARef &::= r^p \\
r : Ref &::= r.x \mid x \\
P : Prog &::= m
\end{aligned}$$

AST nodes are annotated with scope information such that we can distinguish and match on specific AST nodes. The transformation in section 4 exploits this.

The typing rules of our module calculus are provided in appendix A, fig. 8. We assume the existence of a type checker that produces the annotated ASTs in a way that respects these typing rules.

## 4 Type-Safe Renaming and Qualification

The type and scope discipline from section 3 lets us characterize safe renamings.

**Definition 1 (Least path).** *A path is* least *when no other paths are lesser.*

$$\frac{(\forall p'.\ \epsilon \vdash p' : \mathsf{src}\ p' \xrightarrow{\mathsf{word}\ p'} \mathsf{tgt}\ p' \implies p \leq p')}{\epsilon \vdash \mathsf{least}\ p}$$

**Definition 2 (Safe renaming).** *Renaming a binder $x : T$ to $y : T$ is safe for a program with scope graph $G = \langle \zeta; \epsilon; \psi \rangle$ when there exists a scope $s$, a function $f : Path \to Path$, and a graph $G' = \langle \zeta; \epsilon'; \psi' \rangle$ for which the following holds:*

$$\frac{\begin{gathered}\epsilon = \epsilon_1 \sqcup s' \overset{x:T}{\twoheadrightarrow} s \qquad \epsilon' = \epsilon_1 \sqcup s' \overset{y:T}{\twoheadrightarrow} s \\ \left(\begin{aligned}&\forall p \in \psi. \\ &\quad \mathsf{src}\ p = \mathsf{src}\ (f\,p) \wedge \mathsf{tgt}\ p = \mathsf{tgt}\ (f\,p) \wedge \\ &\quad \epsilon' \vdash f\,p : \mathsf{src}\ (f\,p) \xrightarrow{\mathsf{word}\ (f\,p)} \mathsf{tgt}\ (f\,p) \wedge \epsilon' \vdash \mathsf{least}\ (f\,p)\end{aligned}\right) \\ \psi' = \mathsf{map}\ f\ \psi\end{gathered}}{\langle \zeta; \epsilon; \psi \rangle; f; x^s \rightsquigarrow y : T; \langle \zeta; \epsilon'; \psi' \rangle}$$

The first two premises in definition 2 say that the edges in the graph before and after are the same, except for the renamed binder edge. The second premise says that the mapping $f$ preserves source- and target scopes and yields least paths in the new graph. The final premise says that $\psi'$ contains the same paths as $\psi$ but with $f$ applied to each.

Using this, we can define a renaming function, whose definition we elide for brevity, with the following type:

$$\textsf{rename} : \underbrace{(\mathit{Path} \to \mathit{Path})}_{\substack{\text{Mapping paths in old} \\ \text{to paths in new graph}}} \to \underbrace{\mathit{Name} \to \mathit{Scope}}_{\substack{\text{Identifies binder} \\ \text{to rename}}} \to \underbrace{\mathit{Name}}_{\substack{\text{New} \\ \text{name}}} \to \mathit{Prog} \to \mathit{Prog}$$

*Conjecture 1.* If a renaming is safe, then the rename transformation preserves program well-typedness.

$$G; f; x^s \rightsquigarrow y : T; G' \ \wedge \ G \vdash P \implies G' \vdash \textsf{rename}\ f\ x\ s\ y\ P$$

*Proof.* We expect the proof to be straightforward. Verifying this is future work.

A prerequisite for definition 2 is the existence of a function that maps paths in the graph before to least paths in the graph after. A naive approach to constructing this function could be to re-resolve (i.e., recompute) *all* paths in the graph. More precise approaches could only re-resolve those paths for which it is necessary, such that we do not re-resolve paths that are guaranteed to be unchanged. For example, paths that resolve to entirely different names from the ones affected by the renaming.

A benefit of our approach is that we can implement such heuristics and verify them w.r.t. the specification in definition 2, independently of the renaming transformation itself. It is a question for future work to explore and prove the safety of such heuristics. Another question for future work is to explore structural transformations for renaming modules, merge/split modules, move definitions, and the *extract method* refactoring [3].

## 5 Related Work

Refactoring has a long and distinguished history, going back to the pioneering works of Griswold [4] and Opdyke [11]. Here we focus on the most closely related lines of work on building and verifying safe and trustworthy refactorings.

Closely related to our work is the work of Schäfer et al. [15] and Schäfer and de Moor [17] who build and verify sound refactorings, and renamings in particular [15]. Their approach to sound renaming is closely related to our approach, albeit they do not connect or base their approach on a formalized type discipline. Instead, their refactorings are built using attribute grammars. In their approach, qualified names are resolved (or "looked up" in their terminology) by (non-lexically) traversing abstract syntax trees. Also similarly to us, their renaming transformation preserves names and resolves name capture conflicts by qualifying names. They guarantee [16] that their renamings produce soundly qualified names, in the following sense. Schäfer et al. [15] defines sound qualification by means of two functions, roughly typed as (we adjust their type notation to match the type notation used in this abstract) $\textsf{lookup} : \mathit{Pos} \to \mathit{Ref} \rightharpoonup \mathit{Decl}$ and $\textsf{access} : \mathit{Pos} \to \mathit{Decl} \rightharpoonup \mathit{Ref}$, where $\mathit{Pos}$ represents an AST position, $\mathit{Ref}$ is a

(possibly qualified) name, and *Decl* represents the AST position of a declaration. They verify in Coq that these functions satisfy the following equality, for any position $p$ and declaration position $d$: $\mathsf{lookup}\ p\ (\mathsf{access}\ p\ d) = d$. Our renaming transformation in section 4 satisfies a similar property. Specifically, our definition of safe renaming (definition 2) guarantees that each path resolves to the same declaration as before. Furthermore, unlike Schäfer et al. [15] and Schäfer et al. [16], conjecture 1 formally states that renaming is a type-preserving transformation. In ongoing work, we believe that similar formal statements can be made and proven for more ambitious refactorings, which (to the best of our knowledge) no one has attempted to formalize to date.

Rowe et al. [14] also define a sound renaming transformation. They provide a full formalization of the transformation and a type safety proof. Their approach and proof relies on a denotational model of name binding for OCaml which they invented for the purpose of characterizing sound renamings. The main difference between their approach and the approach we explore in this abstract, is that their semantics exists independently of the typing semantics of OCaml, in our type and scope discipline, the semantics of name binding *is* the typing semantics. On the other hand, while scope graphs have been demonstrated to subsume many binding patterns, it is a question for future work whether our approach could scale to a language like OCaml.

The work of Thompson and Li [19] and Thompson and Horpácsi [18] also explores trustworthy transformation tools for functional languages. Li and Thompson [7] formalizes behavior preservation of a range of refactorings, including refactorings for a module calculus. They prove behavior preservation of their transformations with respect to an equational specification of the module calculus. We consider only how to prove type preservation, using our type and scope discipline. While the property we prove is a weaker one, this weakness may be a strength for some applications, such as verifying transformations that preserve well-typedness but not behavior. Furthermore, we expect that our discipline scales to more ambitious binding patterns than the ones considered in our (deliberately) simple module calculus.

Our approach to verifying transformations is inspired by recent work by [9] who present a methodology for proving that transformations preserve well-typedness using scope graphs. Their work leaves open the question of how to adapt paths and qualify names during transformation. We provide a solution to this, using a different notion of scope graphs that provides a direct correspondence between paths and qualified names.

Pelsmaeker et al. [12] describe a language-parametric framework for synthesizing (qualified) names. Their framework uses the generic constraint solver for the *Statix* language [1, 21] to re-qualify names to prevent unintended name capture, using only the typing rules of a language to synthesize qualified names. The approach described in this paper uses a notion of scope graphs that provides a direct correspondence between paths and qualified names. The typing rules in our approach explicitly compose fragments, whereas *Statix* uses a separation logic-inspired constraint syntax to handle such fragment composition implicitly.

# Bibliography


[1] van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proc. ACM Program. Lang. **2**(OOPSLA), 114:1–114:30 (2018), https://doi.org/10.1145/3276484, URL https://doi.org/10.1145/3276484

[2] Erdweg, S., van der Storm, T., Dai, Y.: Capture-avoiding program transformations with name-fix. In: Aßmann, U., Demuth, B., Spitta, T., Püschel, G., Kaiser, R. (eds.) Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany, LNI, vol. P-239, pp. 93–94, GI (2015), ISBN 978-3-88579-633-6, URL https://dl.gi.de/handle/20.500.12116/21184

[3] Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley (1999), ISBN 978-0-201-48567-7, URL http://martinfowler.com/books/refactoring.html

[4] Griswold, W.G.: Program Restructuring As an Aid to Software Maintenance. Ph.D. thesis, Seattle, WA, USA (1992)

[5] Kang, H.J., Thung, F., Lawall, J., Muller, G., Jiang, L., Lo, D.: Semantic patches for java program transformation (experience report). In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom, LIPIcs, vol. 134, pp. 22:1–22:27, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), ISBN 978-3-95977-111-5, https://doi.org/10.4230/LIPICS.ECOOP.2019.22, URL https://doi.org/10.4230/LIPIcs.ECOOP.2019.22

[6] Lawall, J., Muller, G.: Automating program transformation with coccinelle. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings, Lecture Notes in Computer Science, vol. 13260, pp. 71–87, Springer (2022), ISBN 978-3-031-06772-3, https://doi.org/10.1007/978-3-031-06773-0_4, URL https://doi.org/10.1007/978-3-031-06773-0_4

[7] Li, H., Thompson, S.J.: Formalisation of haskell refactorings. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005, Trends in Functional Programming, vol. 6, pp. 95–110, Intellect (2005)

[8] Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G. (eds.) Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2505, pp. 286–301, Springer (2002), ISBN 3-540-44310-X, https://doi.org/10.1007/3-540-45832-8_22, URL https://doi.org/10.1007/3-540-45832-8_22

[9] Miljak, L., Poulsen, C.B., van Spaendonck, F.: Verifying well-typedness preservation of refactorings using scope graphs. In: Tomb, A. (ed.) Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023, pp. 44–50, ACM (2023), https://doi.org/10.1145/3605156.3606455, URL https://doi.org/10.1145/3605156.3606455

[10] Neron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A theory of name resolution. In: Vitek, J. (ed.) 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, vol. 9032, pp. 205–231, Springer (2015), ISBN 978-3-662-46668-1, https://doi.org/10.1007/978-3-662-46669-8_9, URL https://doi.org/10.1007/978-3-662-46669-8_9

[11] Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA (1992)

[12] Pelsmaeker, D.A.A., Zwaan, A., Bach Poulsen, C., Mooij, A.: Language-parametric reference synthesis (2024), under review for OOPSLA'25.

[13] Rouvoet, A., van Antwerpen, H., Poulsen, C.B., Krebbers, R., Visser, E.: Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. Proc. ACM Program. Lang. **4**(OOPSLA), 180:1–180:28 (2020), https://doi.org/10.1145/3428248, URL https://doi.org/10.1145/3428248

[14] Rowe, R.N.S., Férée, H., Thompson, S.J., Owens, S.: Characterising renaming within ocaml's module system: theory and implementation. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, pp. 950–965, ACM (2019), ISBN 978-1-4503-6712-7, https://doi.org/10.1145/3314221.3314600, URL https://doi.org/10.1145/3314221.3314600

[15] Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for java. In: Harris, G.E. (ed.) Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, pp. 277–294, ACM (2008), ISBN 978-1-60558-215-3, https://doi.org/10.1145/1449764.1449787, URL https://doi.org/10.1145/1449764.1449787

[16] Schäfer, M., Ekman, T., de Moor, O.: Formalising and verifying reference attribute grammars in coq. In: Castagna, G. (ed.) Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5502, pp. 143–159, Springer (2009), ISBN 978-3-642-00589-3, https://doi.org/10.1007/978-3-642-00590-9_11, URL https://doi.org/10.1007/978-3-642-00590-9_11

[17] Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, pp. 286–301, ACM (2010), ISBN 978-1-4503-0203-6, https://doi.org/10.1145/1869459.1869485, URL https://doi.org/10.1145/1869459.1869485

[18] Thompson, S.J., Horpácsi, D.: Refactoring = substitution + rewriting: Towards generic, language-independent refactorings. In: Lämmel, R., Mosses, P.D., Steimann, F. (eds.) Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands, OASIcs, vol. 109, pp. 26:1–26:9, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), ISBN 978-3-95977-267-9, https://doi.org/10.4230/OASICS.EVCS.2023.26, URL https://doi.org/10.4230/OASIcs.EVCS.2023.26

[19] Thompson, S.J., Li, H.: Refactoring tools for functional languages. J. Funct. Program. **23**(3), 293–350 (2013), https://doi.org/10.1017/S0956796813000117, URL https://doi.org/10.1017/S0956796813000117

[20] Tip, F.: Refactoring using type constraints. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings, Lecture Notes in Computer Science, vol. 4634, pp. 1–17, Springer (2007), ISBN 978-3-540-74060-5, https://doi.org/10.1007/978-3-540-74061-2_1, URL https://doi.org/10.1007/978-3-540-74061-2_1

[21] Zwaan, A., van Antwerpen, H., Visser, E.: Incremental type-checking for free: using scope graphs to derive incremental type-checkers. Proc. ACM Program. Lang. **6**(OOPSLA2), 424–448 (2022), https://doi.org/10.1145/3563303, URL https://doi.org/10.1145/3563303

## A Typing Rules

The typing rules of our module calculus are provided in fig. 8.

The rule Num asserts that a number is well-typed in any scope, it constructs no scopes, edges, and it does not rely on any paths.

The first premise of the Ref rule uses the function $\mathsf{reify} : \mathit{Path} \to \mathit{Ref}$ print the (potentially qualified) name that a path corresponds to. The second premise asserts that $p$ is a well-formed path; the third that the labels of the path satisfies the well-formedness constraint discussed in section 2.2; and the fourth that $p$ is the least path, using the path ordering also discussed in section 2.2. It is worth noting that the Ref rule disallows names that do not correspond to a least path; e.g., overly qualified names. This guarantees that names in programs are "canonical". For practical purposes, it may be desirable (and it is perfectly possible) to give a more relaxed rule that allows names to be qualified even though the qualification is not needed.

The rules for $\lambda$s and application are mostly standard. Unlike standard typing rules for $\lambda$, they propagate the disjoint union of the scope graph fragments that each sub-term constructs. The Lam rule asserts that $\lambda$s construct a lexical sub-scope $s_2$ of the current sub-scope, and a sub-scope $s_3$ which is connected to $s_2$ via an edge that declares the binder $x : T$.

The rule for Def is conceptually similar to the rule for Lam: it connects a sub-scope to its lexical context, and declares a binder in an edge label. A difference between the Lam rule and the Def rule is that the Lam rule has the binder edge associated with the sub-scope ($s_2$) of the current scope ($s_1$), whereas in Def, the binder edge is associated with the current scope ($s_1$). The Mod rule asserts ($F_1$) that a module scope $s_2$ is lexically connected to the current scope ($s_1$), and that the current scope has a **mod** $X$ declaration edge which makes the module reachable and visible in the graph. The scope fragments of each module member is also propagated by the rule (in $F_2$). The Import rule asserts that an import is a well-formed and unique (potentially) qualified reference. The Prog rule asserts that a program is typed w.r.t. a root scope $\bullet$, and that the scope fragments that the program constructs is exactly the scoping structure contained in the scope graph $G$ that the program is typed and scoped w.r.t.

## B Path Ordering

Figure 9 defines the partial order we use on paths. Fewer prefers names with fewer qualifiers over names with more. The rule uses the helper relation $p_1 \triangleright p_2 \sim p$ to segmentize a path into an "initial" segment and a "qualified name" segment, such that we can compare names modulo number of qualifiers. Same says that, in case of names with the same number of qualifiers, we use the step-wise ordering $<_{step}$ which prefers paths whose label prefixes are step-wise equal but one is a prefix of the other, or, following the shared prefix, one traverses an import edge (preferred) and the other a lexical parent edge.

The judgment $p \doteq p$ says that two paths are equal when neither is shorter than the other.

NUM

$$\frac{}{G \vdash n^s : \mathbf{Int} \dashv \langle []; []; [] \rangle}$$

REF

$$\frac{\epsilon \vdash p : s \xrightarrow{w} s' \qquad \begin{array}{c} r = \mathsf{reify}\ p \\ w \in \mathcal{L}\left(\textsc{lex}^* \cdot \mathbf{imp}^? \cdot (\exists X.\ \mathbf{mod}\ X)^* \cdot \exists x.\ x : T\right) \end{array} \qquad \epsilon \vdash \mathsf{least}\ p}{\langle \zeta; \epsilon; \psi \rangle \vdash (r^p)^s : T \dashv \langle []; []; [p] \rangle}$$

LAM

$$\frac{G \vdash e^{s_2} : T_2 \dashv F_1 \qquad F_2 = \langle [s_2, s_3]; [s_2 \twoheadrightarrow s_1, s_2 \overset{x:T_1}{\twoheadrightarrow} s_3]; [] \rangle}{G \vdash (\lambda x.\ e^{s_2})^{s_1} : T_1 \rightarrow T_2 \dashv F_1 \sqcup F_2}$$

APP

$$\frac{G \vdash e_1^s : T_1 \rightarrow T_2 \dashv F_1 \qquad G \vdash e_2^s : T_1 \dashv F_2}{G \vdash (e_1^s\ e_2^s)^s : T_2 \dashv F_1 \sqcup F_2}$$

DEF

$$\frac{G \vdash e^{s_2} : T \dashv F_1 \qquad F_2 = \langle [s_2]; [s_2 \twoheadrightarrow s_1, s_1 \overset{x:T}{\twoheadrightarrow} s_2]; [] \rangle}{G \vdash (\mathbf{def}\ x : T = e^{s_2})^{s_1} \dashv F_1 \sqcup F_2}$$

MOD

$$\frac{\begin{array}{c} F_1 = \langle [s_2]; [s_2 \twoheadrightarrow s_1, s_1 \overset{\mathbf{mod}\ X}{\twoheadrightarrow} s_2]; [] \rangle \\ \overline{F} = \left[\ F \ \middle|\ \exists d^{s_2} \in \overline{d^{s_2}}.\ G \vdash d^{s_2} \dashv F\ \right] \qquad F_2 = \bigsqcup_{F \in \overline{F}} F \end{array}}{G \vdash (\mathbf{mod}\ X\ \{\ \overline{d^{s_2}}\ \})^{s_1} \dashv F_1 \sqcup F_2}$$

IMPORT

$$\frac{\epsilon \vdash p : s \xrightarrow{w} s' \qquad \begin{array}{c} r = \mathsf{reify}\ p \\ w \in \mathcal{L}\left(\textsc{lex}^* \cdot \mathbf{imp}^? \cdot (\exists X.\ \mathbf{mod}\ X)^+\right) \end{array} \qquad \epsilon \vdash \mathsf{least}\ p}{\langle \zeta; \epsilon; \psi \rangle \vdash (\mathbf{import}\ r^p)^s \dashv \langle []; [s \overset{\mathbf{imp}}{\twoheadrightarrow} s']; [p] \rangle}$$

PROG

$$\frac{G \vdash m^\bullet \dashv \langle \zeta; \epsilon; \psi \rangle \qquad \langle \bullet, \zeta; \epsilon; \psi \rangle = G}{G \vdash m}$$

**Fig. 8.** Typing rules of a module calculus

$$\textsc{Segmentize}\quad \frac{\mathsf{word}\ p_1 \in \mathcal{L}(\textsc{lex}^* \cdot \mathbf{imp}^?) \qquad \mathsf{word}\ p_2 \in \mathcal{L}\left((\exists X.\ \mathbf{mod}\ X)^* \cdot ((\exists X.\ \mathbf{mod}\ X) + (\exists x, T.\ x : T))\right)}{p_1 \triangleright p_2 \sim p}$$

$$\textsc{Fewer}\quad \frac{p_1 \triangleright p_2 \sim p \qquad p_1' \triangleright p_2' \sim p' \qquad |p_2| < |p_2'|}{p < p'}$$

$$\textsc{Same}\quad \frac{p_1 \triangleright p_2 \sim p \qquad p_1' \triangleright p_2' \sim p' \qquad |p_2| = |p_2'| \qquad p_1 <_{step} p_1'}{p < p'}$$

$$\textsc{Shorter}\quad \frac{}{[] \cdot s <_{step} (s_1 \overset{\ell}{\rightarrow} s_2, \epsilon) \cdot s'} \qquad \textsc{Imp-Pref}\quad \frac{}{(s_1 \overset{\mathbf{imp}}{\rightarrow} s_2, \epsilon) \cdot s <_{step} (s_1' \overset{\textsc{lex}}{\rightarrow} s_2', \epsilon') \cdot s'}$$

$$\textsc{Step}\quad \frac{\epsilon \cdot s <_{step} \epsilon' \cdot s'}{(s_1 \overset{\ell}{\rightarrow} s_2, \epsilon) \cdot s <_{step} (s_1' \overset{\ell}{\rightarrow} s_2', \epsilon') \cdot s'} \qquad p \doteq p' \iff p \not< p' \wedge p' \not< p$$

**Fig. 9.** Path ordering

## C Renaming Function

$$\mathsf{rename} : (\mathit{Path} \to \mathit{Path}) \to \mathit{Name} \to \mathit{Scope} \to \mathit{Name} \to \mathit{Prog} \to \mathit{Prog}$$
$$\mathsf{rename}\ f\,x\,s\,y\,m = \mathsf{rename_m}\ f\,x\,s\,y\,m$$

$$\mathsf{rename_m} : (\mathit{Path} \to \mathit{Path}) \to \mathit{Name} \to \mathit{Scope} \to \mathit{Name} \to \mathit{ADecl} \to \mathit{ADecl}$$
$$\mathsf{rename_m}\ f\,x\,s\,y\,(\mathbf{def}\ x : T = a)^{s_1} = \begin{cases} \mathbf{def}\ y : T = \mathsf{rename_a}\ f\,x\,s\,y\,a & \text{if } s = s_1 \\ \mathbf{def}\ x : T = \mathsf{rename_a}\ f\,x\,s\,y\,a & \text{otherwise} \end{cases}$$
$$\mathsf{rename_m}\ f\,x\,s\,y\,(\mathbf{mod}\ X\ \{\ \overline{m}\ \})^{s_1} = \mathbf{mod}\ X\ \{\ \mathsf{map}\ (\mathsf{rename_m}\ f\,x\,s\,y)\ \overline{m}\ \}$$
$$\mathsf{rename_m}\ f\,x\,s\,y\,(\mathbf{import}\ (r^p))^{s_1} = (\mathbf{import}\ (\mathsf{reify}\ (f\,p))^{f\,p})^{s_1}$$

$$\mathsf{rename_a} : (\mathit{Path} \to \mathit{Path}) \to \mathit{Name} \to \mathit{Scope} \to \mathit{Name} \to \mathit{AExpr} \to \mathit{AExpr}$$
$$\mathsf{rename_a}\ f\,x\,s\,y\,n^{s_1} = n^{s_1}$$
$$\mathsf{rename_a}\ f\,x\,s\,y\,(r^p)^{s_1} = ((\mathsf{reify}\ (f\,p))^{f\,p})^{s_1}$$
$$\mathsf{rename_a}\ f\,x\,s\,y\,(a_1\ a_2)^{s_1} = ((\mathsf{rename_a}\ f\,x\,s\,y\,a_1)\ (\mathsf{rename_a}\ f\,x\,s\,y\,a_2))^{s_1}$$
$$\mathsf{rename_a}\ f\,x\,s\,y\,(\lambda x.\, e^{s_2})^{s_1} = \begin{cases} (\lambda y.\, (\mathsf{rename_a}\ f\,x\,s\,y\,e^{s_2}))^{s_1} & \text{if } s_2 = s \\ (\lambda x.\, (\mathsf{rename_a}\ f\,x\,s\,y\,e^{s_2}))^{s_1} & \text{otherwise} \end{cases}$$