Verifying Well-Typedness Preservation of Refactorings using Scope Graphs

Luka Miljak
l.miljak@tudelft.nl
Delft University of Technology
The Netherlands

Casper Bach Poulsen
c.b.poulsen@tudelft.nl
Delft University of Technology
The Netherlands

Flip van Spaendonck
p.h.m.v.spaendonck@tue.nl
Eindhoven University of Technology
The Netherlands

ABSTRACT

The goal of automated refactoring is to reduce maintenance effort. To realize this, programmers need to be able to trust or manually check that refactorings actually preserve behavior. To allow programmers to focus on such checks, automated refactorings should preserve program well-typedness. However, historically automated refactorings in popular IDEs could break well-typedness. The reason is that modern languages have complex name binding semantics which makes it hard to guarantee well-typedness in general.

In recent work, *scope graphs* have been proposed as a uniform model for name binding. The model supports complex name binding patterns, and its uniformity makes it attractive to consider for verifying that refactorings preserve well-typedness. This paper explores how to prove that refactorings preserve well-typedness, using scope graphs. We consider a simple refactoring for merging modules in a toy module language, and prove that this refactoring preserves well-typedness. We give a generic template for proving well-typedness preservation using scope graphs, and discuss how this template relates to refactorings more generally.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; • **Theory of computation** → **Program verification**.

KEYWORDS

safe refactoring, scope graph, name binding, static semantics

ACM Reference Format:
Luka Miljak, Casper Bach Poulsen, and Flip van Spaendonck. 2023. Verifying Well-Typedness Preservation of Refactorings using Scope Graphs. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605156.3606455

1 INTRODUCTION

Refactoring is an important aspect of large software project development. The idea is to modify the *internal structure* of code components while preserving their *externally observable behavior*, such that it becomes easier to understand and modify components in the future. For large software projects, manual refactorings can be prohibitively expensive. For this reason, modern IDEs and code transformation tools provide integrated support for *automated refactoring*. In a study on behavior preservation approaches, AlOmar et al. [1] conclude that the behavior of many refactorings are under-researched. A key challenge of automated refactoring is the difficulty of guaranteeing that code transformations preserve external behavior. To mitigate this, automated refactoring is, according to Tip et al. [11], an interactive process relying on the programmer to manually check that behavior is preserved post refactoring. To reduce the work of programmers, automated refactorings should preserve the well-typedness of programs. While not as strong a property as behavioral preservation, well-typedness would catch many behavioral problems as well. Name binding issues in particular are a common cause for bugs in new refactorings [8].

Modern tools for program transformation, such as Stratego 2 and Rascal [9, 15], can guarantee that transformed programs are syntactically correct. This guarantee arises from the fact that these transformations are defined directly on the underlying syntactic model of *abstract syntax trees*, specified by *context-free grammars*.

However, it is challenging to guarantee that refactorings preserve well-typedness. Modern languages provide fine-grained control over how to structure and reuse code components via complex name binding features, such as class-based inheritance, nested classes, static members, traits, imports, generics, etc. Refactoring engines in modern IDEs address this challenge by incorporating types in their engine [11]. But, with the exception of the work of Schäfer et al. [7], refactoring engines do not come with formal guarantees that they preserve well-typedness. We conjecture that a reason for this lack of formal guarantee is that modern programming languages have lacked a uniform model of name binding, unlike syntax.

Visser and coauthors [3, 6, 13, 14, 16] have recently developed *scope graphs* as a candidate for this lacking model. Scope graphs provide a formalism and data structure for defining the static semantics of name binding and resolution. This uniform data structure supports declarative specification of static semantics [6, 13, 14]. From these specifications, we can derive both executable type checkers and *language parametric editor services* such as semantic code completion [4]. In § 2, we provide a more formal explanation for scope graphs used throughout the remainder of the paper.

This paper explores how scope graphs can be used to prove that refactorings preserve well-typedness. We consider a small module language which, in spite of its simplicity, has features that are representative of more realistic languages.

Contributions. They key contribution in our paper is a proof template for verifying that a refactoring preserves well-typedness. We instantiate the template on a refactoring for a module merge refactoring for a small module language (§ 3), and discuss how the template scales to more realistic languages and refactorings (§ 4).

FTfJP '23, July 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0246-4/23/07.
https://doi.org/10.1145/3605156.3606455

2 BACKGROUND: TYPE SYSTEM SPECIFICATION USING SCOPE GRAPHS

A scope graph is a generic model that represents the name binding structure of a language. For simplicity, we consider a simplified notion of scope graphs compared to the traditional notion due to Visser and coauthors [3, 6, 13, 14, 16]. We discuss our simplifications in detail in § 4. For our purpose, we define scope graphs as follows:

Definition 2.1 (Scope Graph). A scope graph G is a triple (S, E, ρ), where S is a set of identifiers representing scopes (nodes) in the graph. E is a function $S \to \mathcal{P}(S)$, where $E(s)$ represents the set of outgoing edges of scope s. Finally, ρ is a function $S \to \mathcal{P}(D)$, where $\rho(s)$ is the set of *declarations* associated with scope s. The set of all possible declarations D varies from language to language. We will write S_G, E_G, and ρ_G to project the corresponding components from a graph.

Edges represent the access between the different scopes. E.g., An edge from a scope s_1 to s_2 could indicate that s_1 is a lexical child of s_2, or that s_1 imports s_2. References in a program are resolved by finding paths in the scope graph, which are denoted using a reachability relation $\twoheadrightarrow$.

Definition 2.2 (Reachability). For a scope graph G with $s \in S_G$ and $d \in D$, we say that d is reachable from s if $s \twoheadrightarrow d$ holds. The relation $\twoheadrightarrow$ is defined as follows:

$$\frac{d \in \rho_G(s)}{s \twoheadrightarrow d} \qquad \frac{s' \in E(s) \qquad s' \twoheadrightarrow d}{s \twoheadrightarrow d}$$

A proof for $s \twoheadrightarrow d$ can be represented as a path $s \cdot s_1 \cdot \ldots \cdot s_n \cdot d$ with length n.

The typing rules of a language define a relation $G \vdash p$ (read: program p is well-typed under scope graph G). With this, we can define what it means for a refactoring to preserve well-typedness.

Definition 2.3 (Preservation of Well-Typedness). Let $[\![\_]\!]$ be some refactoring from program to program. The refactoring *preserves well-typedness* if for all well-typed programs p under scope graph G (i.e. $G \vdash p$), there exists a scope graph G' under which the refactored program is well-typed ($G' \vdash [\![p]\!]$).

Simple Module Language. In Figure 1, we show the typing rules of a simple conceptual language with modules (SML). An SML program is a sequence of modules, each having a name, a sequence of import statements, and fields. Variables have access to all fields within the same module and the imported modules. Imports work transitively. As a simplification, we will not account for shadowing and assume that every name occurring in a program is unique.

The SML-prog rule asserts that a program has a single global program scope $s \in S_G$ with no outgoing edges. The SML-mod rule asserts for each module, that there is a scope $s_\text{m} \in S_G$ with an outgoing edge to s, indicating that the module is a lexical child of the top-level program. A module scope has an additional set of outgoing edges s_is representing imports. This specifies that references in a module can reach all declarations from other modules it imports. The transitive import behavior is a result from imported module scopes having edges to their own imported modules.

The program scope contains module declarations, whereas the module scopes contain field declarations.

Syntax

i	$\in \mathbb{Z}$		integers	
x	$\in \mathcal{X}$	:=	some countable set	
e		::=	$i \mid x$	expressions
f		::=	$x := e$	fields
imp		::=	`import` x	imports
m		::=	`mod` x; imp^*; f^*	modules
p		::=	`prog` m^*	programs

Typing Rules

$$\text{(SML-int)}\ \frac{}{G, s_\text{m} \vdash i : \texttt{int}} \qquad \text{(SML-var)}\ \frac{s_\text{m} \twoheadrightarrow \texttt{FIELD}(x, t)}{G, s_\text{m} \vdash x : t}$$

$$\text{(SML-fld)}\ \frac{G, s_\text{m} \vdash e : t}{G, s_\text{m} \vdash x := e : \texttt{FIELD}(x, t)}$$

$$\text{(SML-imp)}\ \frac{s \twoheadrightarrow \texttt{MOD}(x, s_\text{i})}{G, s \vdash \texttt{import}\ x : s_\text{i}}$$

$$\text{(SML-mod)}\ \frac{\begin{array}{c} s_\text{m} \in S_G \qquad E_G(s_\text{m}) = \{s\} \cup s\_is \\ (\forall i\ \ G, s \vdash is_i : s\_is_i) \\ \rho_G(s_\text{m}) = ds \qquad (\forall i\ \ G, s_\text{m} \vdash fs_i : ds_i) \end{array}}{G, s \vdash \texttt{mod}\ s;\ is;\ fs : \texttt{MOD}(x, s_\text{m})}$$

$$\text{(SML-prog)}\ \frac{\begin{array}{c} s \in S_G \qquad E_G(s) = \emptyset \\ \rho_G(s) = ds \qquad (\forall i\ \ G, s \vdash ms_i : ds_i) \end{array}}{G \vdash \texttt{prog}\ ms}$$

Figure 1: Syntax and typing rules of a simple module language.

(1) As shown by SML-fld, a field declaration is represented by $\texttt{FIELD}(x, t)$, where x is the name of the field and t the type.[1] When resolving variables in SML-var, we assert the existence of a path to a `FIELD` declaration.
(2) As shown by SML-mod, a module declaration is represented by $\texttt{MOD}(x, s_\text{m})$, where x is the name of the module and s_m is the scope that corresponds to the module. Including the module scope in the declaration allows us to use it to create import edges towards that module scope. When resolving import statements (SML-imp), we assert the existence of a path to a `MOD` declaration. Having scopes be part of declarations is known as the *scopes as types* paradigm [12].

In Figure 2, we give an example of an SML program with two modules, including its corresponding scope graph. The circles in the figure represent scopes, where s is the program scope and s_foo and s_bar are the module scopes for the corresponding modules. Arrows between scopes represent edges. A square-tip arrow pointing to a rectangle represents a declaration belonging to a scope. Finally, a dashed arrow from declaration to scope indicates that the declaration contains said scope. Module `bar` contains a reference to field `x`. This means a `FIELD` declaration with name `x` should be *reachable*

[1] Because SML only supports the `int` type for variables, we could have also excluded the type argument from `FIELD`.

```
prog
  mod foo; ; x := 5
  mod bar; import foo; y := x
```

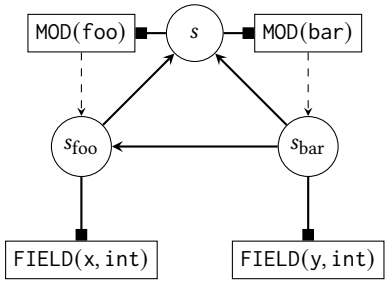

Figure 2: An example SML program with corresponding scope graph following the rules from Figure 1.

from scope s_{bar}. We can see in the figure that this will successfully resolve using the edge from s_{bar} to s_{foo}.

3 EXAMPLE: MERGING MODULES

In Figure 3, we define a refactoring on SML, previously outlined in Figure 1, that merges the first and second modules of a program together. It involves concatenating the fields and the imports of the two modules together. The name of the merged module is set to the name of the first module. We ignore cases where a program has less than two modules. If the two modules have the same import statement, then the merged module will have a duplicate import statement. Similarly, the merging might introduce a self-import. While these are not desirable properties, they do not break well-typedness and can be ignored for the sake of simplicity. An example application of this refactoring is shown in Figure 4.

Modules that used to import the second module will now have to import the first. This is the purpose that the function $\mathit{FixImp}[\![m, x, y]\!]$ serves. Here, x and y are the names of the first and second module respectively. The $\mathit{FixImp}[\![\_]\!]^*$ notation indicates we are mapping the function on a *sequence* of modules.

Our goal is to prove that this refactoring preserves well-typedness as per Definition 2.3. Before going into greater detail, we will first sketch the idea behind the proof through an example.

In graph illustration included in Figure 4 contains two (partially shown) scope graphs, the top half represents the scope graph before refactoring. For simplicity, we left out the module declarations and fields `a` and `b`. The lower half of the graph is a new transformed graph after merging `foo` and `bar`. We have left out the transformed program scope. The dashed lines indicate an association between scopes in the original graph and the transformed graph. The fact that both s_{foo} and s_{bar} associate with s'_{foo} is an indication that s'_{foo} is the merging of the former two scopes.

Our module `bar` includes a reference to field `x`. How would we show that `x` is still reachable from the merged module s'_{foo}? For this, we define some relation $\gg_{\mathcal{M}}$ between scopes in the original graph and the transformed graph. We write $s \gg_{\mathcal{M}} s'$ to indicate that all declarations that can be reached from s, can also be reached from s'. In other words, we wish to prove that $s_{\text{bar}} \gg_{\mathcal{M}} s'_{\text{foo}}$.

Syntax Transformation

$$[\![\texttt{prog}\ [(\texttt{mod}\ x;\ is;\ fs) :: (\texttt{mod}\ y;\ js;\ gs) :: ms]]\!]$$
$$= \texttt{prog}\ \mathit{FixImp}[\![[(\texttt{mod}\ x;\ is\ js;\ fs\ gs) :: ms], x, y]\!]^*$$

$$\mathit{FixImp}[\![(\texttt{mod}\ a;\ is;\ fs), x, y]\!] = \texttt{mod}\ a;\ (\mathit{FixImp}[\![is, x, y]\!]);\ fs$$

$$\begin{array}{lcll} \mathit{FixImp}[\![(\texttt{import}\ a), x, y]\!] & = & \texttt{import}\ x & \text{where } a = y \\ \mathit{FixImp}[\![(\texttt{import}\ a), x, y]\!] & = & \texttt{import}\ a & \text{where } a \neq y \end{array}$$

Graph Transformation

$$\left[\!\!\left[\frac{s \in S_G \quad \ldots \quad G, s \vdash \texttt{mod}\ x; \ldots : \texttt{MOD}(x, s_{\text{mx}}) \quad G, s \vdash \texttt{mod}\ y; \ldots : \texttt{MOD}(y, s_{\text{my}})}{G \vdash \texttt{prog}\ [(\texttt{mod}\ x;\ is;\ fs) :: (\texttt{mod}\ x;\ js;\ gs) :: ms]} \right]\!\!\right]_{\mathcal{G}}$$

$$\begin{array}{lll} = & \texttt{new}\ s' \quad \texttt{with} & \delta_{\mathcal{P}}(s) = s' \\ & & E(s') = \emptyset \\ & & \rho(s') = \{\texttt{MOD}(x, s'_{\text{mx}})\} \cup ds \\ \sqcup & \texttt{new}\ s'_{\text{mx}} \quad \texttt{with} & \delta_{\mathcal{M}}(s_{\text{mx}}) = \delta_{\mathcal{M}}(s_{\text{my}}) = s'_{\text{mx}} \\ & & E(s'_{\text{mx}}) = \{s'\} \cup s\_is \cup s\_js \\ & & \rho(s'_{\text{mx}}) = \rho_G(s_{\text{mx}}) \cup \rho_G(s_{\text{my}}) \\ \sqcup & \bigsqcup_i [\![(G, s \vdash ms_i \vdash \ldots),\ s']\!]_{\mathcal{G}} & \end{array}$$

$$\begin{array}{lll} \text{where} & ds & = \bigcup_i \mathit{TransDecl}[\![G, s \vdash ms_i : \ldots]\!] \\ & s\_is & = \bigcup_i \mathit{TransEdge}[\![G, s \vdash is_i : \ldots]\!] \\ & s\_js & = \bigcup_i \mathit{TransEdge}[\![G, s \vdash js_i : \ldots]\!] \end{array}$$

$$[\![(G, s \vdash \texttt{mod}\ x;\ is;\ fs : \texttt{MOD}(x, s_{\text{m}})),\ s']\!]_{\mathcal{G}}$$

$$\begin{array}{lll} = & \texttt{new}\ s'_{\text{m}} \quad \texttt{with} & \delta_{\mathcal{M}}(s_{\text{m}}) = s'_{\text{m}}, \\ & & E(s'_{\text{m}}) = \{s'\} \cup s\_is \\ & & \rho(s'_{\text{m}}) = \rho_G(s_{\text{m}}) \\ \text{where} & s\_is = \bigcup_i \mathit{TransEdge}[\![G, s \vdash is_i : \ldots]\!] & \end{array}$$

$$\mathit{TransDecl}[\![G, s \vdash \texttt{mod}\ x;\ is;\ fs : \texttt{MOD}(x, s_{\text{m}})]\!] = \texttt{MOD}(x, \delta_{\mathcal{M}}(s_{\text{m}}))$$

$$\mathit{TransEdge}[\![G, s \vdash \texttt{import}\ x : s_{\text{i}}]\!] = \delta_{\mathcal{M}}(s_{\text{i}})$$

Figure 3: Refactoring for the SML language that merges the first two modules in together.

At the base case, we get that $s_{\text{qux}} \gg_{\mathcal{M}} s'_{\text{qux}}$, because they both have the declaration `x`. We can show that $s_{\text{baz}} \gg_{\mathcal{M}} s'_{\text{baz}}$ using one inductive step, as the import edge from s_{baz} to s_{qux} can be "simulated" by taking the import edge s'_{baz} to s'_{qux}. With an additional inductive step, we also get that $s_{\text{bar}} \gg_{\mathcal{M}} s'_{\text{foo}}$ by simulating the edge from s_{bar} to s_{baz} by the edge from s'_{foo} to s'_{baz}. This shows that field `x` will still be reachable from the merged module.

In the remainder of this section, we prove the preservation of well-typedness in greater detail (while ignoring some cases in the interest of compactness).

3.1 Step 1 - The Graph Transformation

In proving the preservation of well-typedness, we need to prove the existence of a new scope graph under which the output program is well-typed. Our first step in this proof should therefore be constructing the graph transformation function.

```
mod foo; ; a := 1
mod bar; import baz; b := x
mod baz; import qux;
mod qux; ; x := 5

// after refactoring
mod foo; import baz; a := 1, b := x
mod baz; import qux;
mod qux; ; x := 5
```

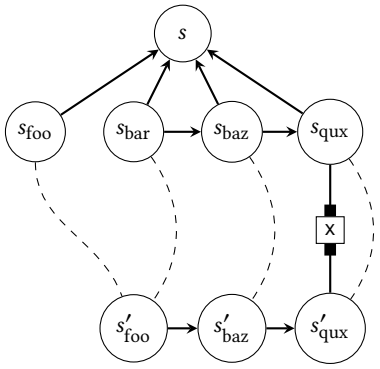

Figure 4: An example merge refactoring with (partially shown) corresponding scope graphs.

We wish to re-use elements from the original graph in our transformation function. However, graphs are defined in a manner too abstract to properly pattern match on. E.g., how do we extract the global program scope given just the tuple (S, E, ρ)? Instead, we define a graph transformation $[\![\_]\!]_{\mathcal{G}}$ by pattern matching on the *typing judgements* from Figure 1. Rather taking just the graph G as input, we take the entire proof tree $G \vdash p$ as input.

Figure 3 includes the graph transformation for the merge refactoring. Within this function, we use the notation $\texttt{new}\ s\ \texttt{with}\ c^*$ to create a graph with a single fresh scope s that satisfies each constraint c. Specifically, the constraints tell what the edges and declarations of s are (and a mapping δ, explained below). The $\sqcup$ operator unions two graphs together.

The function constructs a new program scope s' and a new module scope s'_{mx}, where s'_{mx} represents the merging of the modules with scopes s_{mx} and s_{my}.

To relate scopes from the input graph to the output graph, we construct *two* different mappings $\delta_{\mathcal{P}}$ and $\delta_{\mathcal{M}}$, reflecting the different types of scopes that might occur. Mapping $\delta_{\mathcal{P}}$ only maps the program scope, whereas $\delta_{\mathcal{M}}$ maps the module scopes. In the example illustration of Figure 4, the dashed lines between the scope graphs represent $\delta_{\mathcal{M}}$.

The new program scope s' will have no outgoing edges. Its declarations will change though. We include the module declaration $\texttt{MOD}(x, s'_{\text{mx}})$, representing the merged module, in $\rho(s')$. Next we need to include the module declarations *ds* representing the trailing modules *ms*. To extract a declaration from its module, we have created the function *TransDecl*$[\![\_]\!]$. This function has the following behavior: For a module whose original declaration was $\texttt{MOD}(x, s_{\text{m}})$, we get a new declaration $\texttt{MOD}(x, \delta_{\mathcal{M}}(s_{\text{m}}))$. Re-using the constructed mapping $\delta_{\mathcal{M}}$ allows for this shorthand writing.[2]

In the creation of the new module scope s'_{mx}, the idea is to union the edges and declarations of the original scopes s_{mx} and s_{my}. The union of the declarations can be done using straightforward set union $\rho_G(s_{\text{mx}}) \cup \rho_G(s_{\text{my}})$. For the edges, we include the program scope s', the set of *converted* imported scopes *s_is* from the first module, and the set *s_js* from the second module. Similarly to how we converted the module declarations for the program scope, we can convert import edges using $\delta_{\mathcal{M}}$, defined in *TransDecl*$[\![\_]\!]$.

Lastly, we construct the new scopes for all remaining trailing modules *ms* using $[\![\_, \_]\!]_{\mathcal{G}}$. This part remains equivalent to the original scope graph, with the exception that we again convert the import edges using the mapping $\delta_{\mathcal{M}}$.

3.2 Step 2 - Defining Path Transformations

The main difficulty in proving well-typedness preservation lies in the *path assertions* from our typing rules, i.e., assertions of the form $s \twoheadrightarrow d$. In order to prove that a path exists in our output graph, we would like to re-use and "transform" paths from the input graph. We do this by defining *path transformation relations* $\gg_{\mathcal{P}}$ and $\gg_{\mathcal{M}}$.

The idea behind a path transformation relation $\gg$ is that when writing $s \gg s'$, we claim that every "relevant" declaration that can be reached from $s \in S_G$, can also be reached from $s' \in S_{G'}$, modulo some possible change on the declaration.

Relation $\gg_{\mathcal{P}}$ is used for the global program scope and its module declarations. It should reflect the following: All module declarations $\texttt{MOD}(z, s_{\text{m}})$ in the program scope should still be reachable in the output graph, with the exception of the module named y (the second module in the program). The scope s_{m} transforms into $\delta(s_{\text{m}})$.

Relation $\gg_{\mathcal{M}}$ is used by module scopes and its field declaration. It should reflect that all reachable field declarations $\texttt{FIELD}(x, t)$ from a module scope should still be reachable in the output graph.

Definition 3.1 ($\delta_{\mathcal{P}}$ and $\delta_{\mathcal{M}}$ path transformation). Given graphs G and G', with scopes $s \in S_G$ and $s' \in S_{G'}$, we write $s \gg_{\mathcal{P}} s'$ if the following holds for all declarations $d = \texttt{MOD}(z, s_{\text{mz}})$:

$$\text{if } s \twoheadrightarrow \texttt{MOD}(z, s_{\text{mz}}) \text{ and } z \neq y, \text{ then, } s' \twoheadrightarrow \texttt{MOD}(z, \delta_{\mathcal{M}}(s_{\text{mz}})),$$

where y is an identifier representing the name of the second module in a program. Identifier y, as well as mapping $\delta_{\mathcal{M}}$, should be clear from the context.

Furthermore, we write $s \gg_{\mathcal{M}} s'$ if the following holds for all declarations $d = \texttt{FIELD}(x, t)$:

$$\text{if } s \twoheadrightarrow \texttt{FIELD}(x, t), \text{ then, } s' \twoheadrightarrow \texttt{FIELD}(x, t).$$

3.3 Step 3 - Proving Path Transformations

We claim that the scope mapping $\delta_{\mathcal{P}}$ corresponds with $\gg_{\mathcal{P}}$, i.e., $s \gg_{\mathcal{P}} \delta(s)$, as well as that $\delta_{\mathcal{M}}$ corresponds with $\gg_{\mathcal{M}}$.

Lemma 3.2 ($\delta_{\mathcal{P}}$-map correctness). *Given a well-typed program $G \vdash p$, where $p = \texttt{prog}\ [(\texttt{mod}\ x; is; fs) :: (\texttt{mod}\ y; js; gs) :: ms]$, let $G' = [\![G \vdash p]\!]_{\mathcal{G}}$ with corresponding mapping $\delta_{\mathcal{P}}$. Then, for all scopes $s_2 \in S_G$ for which $\delta_{\mathcal{P}}(s_2) \in S_{G'}$ exists, it holds that $s_2 \gg_{\mathcal{P}} \delta_{\mathcal{P}}(s_2)$.*

[2]The algorithm would need staging, as $\delta_{\mathcal{P}}$ and $\delta_{\mathcal{M}}$ have to be constructed *before* it can be re-used. This can be done because the construction of the scopes can be made independent on the construction of edges and declarations.

Proof. From $G \vdash p$, we can infer the following: $s \in S_G$, $E_G(s) = \emptyset$, $\rho_G(s) = \{\texttt{MOD}(x, s_{\text{mx}}), \texttt{MOD}(y, s_{\text{my}})\} \cup ds$.

The transformation $[\![G \vdash p]\!]_{\mathcal{G}}$ creates a graph with new program scope s', $E_{G'}(s') = \emptyset$, $\rho_{G'}(s') = \texttt{MOD}(x, s'_{\text{mx}})$, with $\delta_{\mathcal{P}}(s) = s'$ and $\delta_{\mathcal{M}}(s_{\text{mx}}) = s'_{\text{mx}}$.

Note that the program scope s is the only scope for which a mapping $\delta_{\mathcal{P}}(s)$ exists. Because we only consider cases where s_2 has a $\delta_{\mathcal{P}}$-mapping, we can instantiate s_2 to s.

We need to prove that for all declarations $d = \texttt{MOD}(z, s_{\text{mz}})$, where $z \neq y$, if we have a path q for $s \twoheadrightarrow \texttt{MOD}(z, s_{\text{mz}})$, then we also have a path q' for $s' \twoheadrightarrow \texttt{MOD}(z, \delta_{\mathcal{M}}(s_{\text{mz}}))$.

Because s has no outgoing edges, we only need to consider the empty path case where $d \in \rho_G(s)$. We will show that $\texttt{MOD}(z, \delta_{\mathcal{M}}(s_{\text{mz}})) \in \rho_{G'}(s')$. There are two possible cases for d where $z \neq y$:

(1) $d = \texttt{MOD}(x, s_{\text{mx}})$. By definition, we have a declaration $\texttt{MOD}(x, s'_{\text{mx}}) = \texttt{MOD}(x, \delta_{\mathcal{M}}(s_{\text{m}})) \in \rho_{G'}(s')$.
(2) $d = ds_i$, with $G, s \vdash ms_i : ds_i$. By definition, we have that $[\![G, s \vdash ms_i]\!]_{\mathcal{D}} \in \rho_{G'}(s')$. The transformation $[\![\_]\!]_{\mathcal{D}}$ produces the exact declaration we desire.

□

Lemma 3.3 ($\delta_{\mathcal{M}}$-map correctness). *Given a well-typed program $G \vdash p$, where $p = \texttt{prog}\ [(\texttt{mod}\ x; is; fs) :: (\texttt{mod}\ y; js; gs) :: ms]$, let $G' = [\![G \vdash p]\!]_{\mathcal{G}}$ with corresponding mapping $\delta_{\mathcal{M}}$. Then, for all scopes $s_2 \in S_G$ for which $\delta_{\mathcal{M}}(s_2) \in S_{G'}$ exists, it holds that $s_2 \gg_{\mathcal{M}} \delta_{\mathcal{M}}(s_2)$.*

Proof. From $G \vdash p$ we can infer that $s, s_{\text{mx}}, s_{\text{my}} \in S_G$, $E_G(s_{\text{mx}}) = \{s\} \cup s\_is$, $E_G(s_{\text{my}}) = \{s\} \cup s\_js$, $\rho_G(s_{\text{mx}}) = dfs$, $\rho_G(s_{\text{my}}) = dgs$. The graph transformation $[\![G \vdash p]\!]_{\mathcal{G}}$ generates a new scope graph with scopes s', s'_{mx} where $E_{G'}(s'_{\text{mx}}) = \{s'\} \cup s\_is' \cup s\_js'$, $\rho_{G'}(s'_{\text{mx}}) = \rho_G(s_{\text{mx}}) \cup \rho_G(s_{\text{my}})$, $\delta_{\mathcal{P}}(s) = s'$, $\delta_{\mathcal{M}}(s_{\text{mx}}) = \delta_{\mathcal{M}}(s_{\text{my}}) = s'_{\text{mx}}$. It also creates the sub-graph $\bigsqcup_i [\![(G, s \vdash ms_i), s']\!]$.

The goal is to show that for all declarations $d = \texttt{FIELD}(z, t)$, if $s_2 \twoheadrightarrow d$, then $\delta(s_2) \twoheadrightarrow d$. There are several possible instantiations for s_2 we have to consider.

(1) Case $s_2 = s_{\text{mx}}$. We prove this by induction on the length of the path $s_{\text{mx}} \twoheadrightarrow d$.
  (a) In the case where the path is empty, i.e., $d \in \rho_G(s_{\text{mx}})$, we get $d \in \rho_G(s_{\text{mx}}) \cup \rho_G(s_{\text{mx}}) = \rho_{G'}(s'_{\text{mx}})$, which shows that $s'_{\text{mx}} \twoheadrightarrow d$.
  (b) Suppose we have a non-empty path, i.e., $s_3 \in E_G(s_{\text{mx}})$ and $s_3 \twoheadrightarrow d$. We can ignore the case where s_3 equals the program scope s, as s does not declare FIELDs, nor does it have outgoing edges. Instead, we assume $s_3 \in s\_is$ (an import edge). The set of transformed import edges s_is' is defined through the function $[\![G, s \vdash \texttt{import...}]\!]_{\mathcal{E}}$. This function essentially maps all edges s_is through $\delta_{\mathcal{M}}$. Thus, we know that there exists a scope $s'_3 = \delta_{\mathcal{M}}(s_3) \in s\_is' \subseteq E_{G'}(s'_{\text{mx}})$. In other words, we simulate the edge from s_{mx} to s_3 by taking the edge from s'_{mx} to s'_3. By the induction hypothesis, we know that $s_3 \gg_{\mathcal{M}} s'_3$, which finalizes our path transformation.
(2) Case $s_2 = s_{\text{my}}$. This proof is analogous to the previous case.
(3) Next, we need to consider all cases where s_2 is a module scope for a module in the non-merged modules ms, with transformed graph $\bigsqcup_i [\![(G, s \vdash ms_i), s']\!]$. We shall skip over this case as these sub-graphs essentially remain unchanged, with the exception of import edges being mapped through $\delta_{\mathcal{M}}$, which have already shown to be properly transforming paths.

□

3.4 Step 4 - Finalizing the proof

We can combine the different moving parts constructed in the above steps to state and prove the preservation of well-typedness.

Lemma 3.4 (Merging Module preserves well-typedness). *For a well-typed program $G \vdash p$, we get that $[\![G \vdash p]\!]_{\mathcal{G}} \vdash [\![p]\!]$.*

Proof. This can be proven by expanding $G \vdash p$. For this proof, we make a distinction between the different assertions in the typing rules of Figure 1. We shall skip the simple scope, edge, or declaration assertions (which have the form $\_ \in S_{G'}$, $E_{G'}(\_) = ...$, and $\rho_{G'}(\_) = ...$), and jump straight to the more difficult cases which require us to prove path assertions (which have the form $\_ \twoheadrightarrow \_$). These are specifically used for references to module declarations (in import statements) and references to fields (in variable expressions). Because these two cases work more or less analogously, we will only consider the case of variables.

We expand $G \vdash p$ until we get a case $G, s_{\text{m}} \vdash x : t$ with hypothesis $s_{\text{m}} \twoheadrightarrow \texttt{FIELD}(x, t)$. Through this expansion, we get that transformed graph $G' = [\![G \vdash p]\!]_{\mathcal{G}}$ has a scope s'_{m} with $\delta_{\mathcal{M}}(s_{\text{m}}) = s'_{\text{m}}$. We need to show that variable x still resolves in the new graph G', i.e., $s'_{\text{m}} \twoheadrightarrow \texttt{FIELD}(x, t)$. This follows directly from Lemma 3.3. □

3.5 The General Proof Template

The proof we displayed used a template whose steps are summarized below. In § 4 we discuss the template's reusability.

(1) Define a graph transformation corresponding the refactoring by pattern matching on the typing judgements of the language. Include mappings $\delta : S_G \rightarrow S_{G'}$ to associate scopes from the input graph to scopes in the output graph.[3]
(2) Analyze the path assertions from the typing rules of the language and construct path transformation relations $\gg$.
(3) Prove $s \gg \delta(s)$ for each relation $\gg$ and corresponding δ, typically done by induction on the size of the path.
(4) Finalize the proof.

4 DISCUSSION AND LIMITATIONS

We discuss two aspects of our work. First is the scalability of the presented template to more realistic use cases. Secondly, we ask ourselves how reasonable and trustworthy our § 3 proof is.

4.1 Towards Realistic Languages and Refactorings

We have showcased a simple refactoring for a simple language, but how well does this scale to more realistic use cases? While we chose the SML language for its simplicity, it also allowed us to showcase a wide range of what scope graphs have to offer. We showed

[3]It is not necessary for every $s \in S_G$ to have a mapping $\delta(s)$ (for example when a scope gets removed)

two important concepts that are also prevalent in most modernly used programming languages. Firstly, *reachability* represents the idea of resolving references, such as variables, to their declaration. Secondly, the *scopes as types* paradigm allows you to specify that declarations can be accessed through other declarations such as imports. Because we made our template in § 3.5 dependent solely on the scope graph model, and because the scope graphs themselves provide a uniform model for name resolution, we conjecture that our approach would work on other languages whose type systems are specified using scope graphs.

A fair point to mention is that the scope graph model we presented in § 2 is only a simplification. The most important feature we omitted is *labeled edges*. Labels serve two purposes. Firstly, they are used to *filter out incorrect paths*. As an example, a more complete type specification for SML might have specified that paths which traverse more than one import edge should be ignored. This would have yielded a non-transitive import behavior. Secondly, labels allow for the specification of some ordering on paths, which would allow us to express name binding concepts such as *shadowing*. For SML, we could have expressed an ordering that prefers paths with no import edges over paths with import edges. Type specifications for more realistic languages require these features. Which means that in order for our template to scale well, it would have to support these two features.

The next question to ask is how well the template scales to more realistic refactorings. While the merge module refactoring might not be the most prevalent refactoring in IDEs, we would argue that other common refactorings do work in a similar manner. Function inlining, moving a method, promoting a local variable to a field, and renaming all work by moving code around from one location to another, or by making minor changes to declarations or references. These can all be expressed as changes in a scope graph. These changes induce path transformations that need to be shown to preserve well-typedness.

A limitation in our proof template lies in its reliance on expressing refactorings as path transformations. Path transformations mostly interact with the *name resolution* aspect of the type system, and name resolution is just one aspect of a type system. However, we argue that name resolution is typically the most intricate aspect, because of its dependence on complicated type environments (or in our case, scope graphs).

4.2 Complexity of the Proof

Another point for discussion is the complexity of the proof template we contributed. One could question whether using traditional type contexts [5] over scope graphs would yield simpler and shorter proofs. We conjecture that using type contexts would be simpler, given that it is not a uniform model and can be tailored specifically to suit your language, whereas scope graphs give a more constrained model. However, we would like to remark that we chose the scope graph model specifically for its uniformity. We argue that a proof template akin to § 3.5 would not be possible without such a uniform model.

What about the trustworthiness of the § 3 proof? We hand-waved many details away. We explicitly ignored cases that were deemed uninteresting or analogous to other cases. Some operators were described only informally, specifically the scope graph union $\sqcup$ and the new scope operator `new` s `with` c. We also made several unmentioned assumptions, among which are the following:

- We ignored *graph minimality* both in our type specification in § 2 as well as our proof. Graph minimality states that if a program is well-typed under a graph G, then G is the smallest graph for said program. This problem arises because our typing rules only assert the *existence* of scopes in the graph, while there might exist scopes whose existence has not been asserted (dangling scopes). However, we argue that this is a negligible problem, as dangling scopes are not reachable from non-dangling scopes.
- *Scope uniqueness* is an important implicit assumption we made. Again, because we only assert the existence of scopes, it is possible that two scopes $s_1, s_2 \in S_G$ might be equal. In SML, this might give strange behavior where there could, for instance, be two modules in a program that share the same module scope. Enforcing uniqueness would require the partitioning of the scope graph in the typing rules in a manner akin to separation logic.
- Our graph transformation function re-used mapping $\delta_{\mathcal{M}}$ for the conversion of scopes in order to, for example, create the import edges. However, we informally defined that a mapping for some scope is optional. This means that before re-using a mapping, a proof would be required that the mapping exists at all.

While we ignored these details for the sake of conciseness, they do harm the trustworthiness of the proof. Mechanizing the proof in a proof assistant such as Agda or Coq [2, 10] would boost confidence. However, at the time of writing this paper, no specification language using scope graphs has yet been mechanized in such a proof assistant. We consider such a mechanization a prerequisite before our proof could be mechanized. As such, it fell outside the scope of the work we intended to present.

5 CONCLUSION AND FUTURE WORK

In this paper, we have explored how program refactoring interacts with scope graphs and how scope graphs can be used to prove that a refactoring preserves well-typedness with respect to name resolution. For this, we presented a proof template and applied it on a toy example refactoring that we argued is representative for a wider range of refactorings. We argued that the template scales well to more realistic languages, as scope graphs provide a uniform model that supports these languages.

As future work, we would like the template to support the full expressive power of scope graphs. This includes labeled edges, filtering over paths, and providing an ordering on paths. Furthermore, to increase confidence in the correctness of our proofs, we would like to mechanize them in a proof assistant such as Agda.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thorough evaluation and detailed comments. This is supported by the Programming and Validating Restructurings project (17933, NWO-TTW, MasCot).

REFERENCES

[1] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* 140 (2021), 106675. https://doi.org/10.1016/j.infsof.2021.106675

[2] Coq Development Team. 2023. *The Coq Proof Assistant.* Retrieved May 26, 2023 from https://coq.inria.fr/

[3] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. 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, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[4] Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-parametric static semantic code completion. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527329

[5] Frank Pfenning. 2004. Benjamin C. Pierce. Types and programming languages. The MIT Press, Cambridge, Massachusetts, 2002, xxi 623 pp. *Bulletin of Symbolic Logic* 10, 2 (2004), 213–214. https://doi.org/10.1017/S1079898600003954

[6] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428248

[7] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2009. Formalising and Verifying Reference Attribute Grammars in Coq. In *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)*, Giuseppe Castagna (Ed.). Springer, 143–159. https://doi.org/10.1007/978-3-642-00590-9_11

[8] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. 2012. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1233–1257. https://doi.org/10.1109/TSE.2012.13

[9] Jeff Smits and Eelco Visser. 2020. Gradually Typing Strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3426425.3426928

[10] The Agda Community. 2023. *The Agda Proof Assistant.* Retrieved May 26, 2023 from https://wiki.portal.chalmers.se/agda/

[11] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3 (2011), 9:1–9:47. https://doi.org/10.1145/1961204.1961205

[12] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (oct 2018), 30 pages. https://doi.org/10.1145/3276484

[13] 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

[14] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276484

[15] Jeroen van den Bos, Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2011. Rascal: From Algebraic Specification to Meta-Programming. *Electronic Proceedings in Theoretical Computer Science* 56 (jun 2011), 15–32. https://doi.org/10.4204/eptcs.56.2

[16] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium (EVCS 2023) (Open Access Series in Informatics (OASIcs), Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:13. https://doi.org/10.4230/OASIcs.EVCS.2023.32

Received 2023-05-26; accepted 2023-06-23