Modular hygienic program transformations

Modulare hygienische Programmtransformationen
Bachelor-Thesis von Nico Ritschel
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Technology Group

Modular hygienic program transformations
Modulare hygienische Programmtransformationen

Vorgelegte Bachelor-Thesis von Nico Ritschel

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 18. März 2015

(Nico Ritschel)

Abstract

Applying a program transformation can unintendedly alter the bindings of variables and other identifiers by moving them into different scopes or adding identifiers with already bound names to the code. To prevent unintended captures between originally unrelated identifiers, several solutions have been proposed. One of them is the algorithm NameFix that allows a retroactive fixing of name captures for already transformed programs. The original version of NameFix however poses several strict constraints that prevent its application on many real-life scenarios.

This thesis presents ways to lift NameFix' constraints on how names are allowed to be related, demonstrates its application to semantically illegal code, and adds extensions that allow its application in a modular context. Using these extensions, it is possible to detect and fix captures across module borders and incrementally add new modules depending on those already fixed. Additionally, constraints can be defined that prevent the alteration of existing module interfaces or other names that are supposed to remain static during name fixing. The implementation of the presented algorithms is finally illustrated on the example of Lightweight Java, a subset of the popular Java programming language.

Contents

List of Figures 7

1 Introduction 9

2 Background 11
- 2.1 Transformation hygiene 12
- 2.2 Name graphs 14
- 2.3 NameFix 15

3 Hygiene for transitive name graphs 19
- 3.1 Capture-avoidance for transitive name graphs 20
- 3.2 NameFix for transitive name graphs 22
- 3.3 Name graphs and NameFix for multi-referencing names 25

4 Hygiene for programs with name resolution issues 27

5 Hygiene for modular programs 31
- 5.1 Modular name resolution 31
- 5.2 Hygiene for cross-module references 34
- 5.3 Propagation of interface renamings 40
- 5.4 Avoidance of renaming exported identifiers 44

6 Case study: Lightweight Java 47
- 6.1 Implementation of Lightweight Java 47
- 6.2 Hygienic transformations for Lightweight Java 49
- 6.3 Modular hygiene for Lightweight Java 52

7 Related Work 55

8 Conclusion and Future Work 57

Bibliography 59

List of Figures

2.1 Example of a small Java program before any transformations are applied. 11
2.2 Example transformations for the program from Fig. 2.1. Left: transformed using a mix of original and synthesized names; Right: transformed using fresh names for all identifiers. . 11
2.3 Example of an extended Java program (left) that is transformed to invalid Java code (right) by a non-hygienic Java program transformation. 12
2.4 Two solutions for the naming conflict in Fig. 2.3: By renaming the synthesized *getCount*-method (left) and by renaming the original *getCount*-method (right). 13
2.5 Name graphs for the example program from Fig. 2.1 (left) and the first transformed program from Fig. 2.2 (right). Nodes/edges added by the transformation are dotted. . . . 15
2.6 The *name-fix* algorithm developed by Erdweg, van der Storm and Dai. 16
2.7 Name graphs for the transformed example program from Fig. 2.3 (left) and the result after NameFix is applied. Class/field names are not included for clarity reasons. 17

3.1 Java program with transitive references (left), and the resulting program when it was insufficiently handled by the NameFix algorithm (right). 19
3.2 Name graphs for the example program from Fig. 3.1 (left) and the version insufficiently renamed by NameFix (right). Synthesized nodes/edges are dotted, class names are not included for clarity reasons. 20
3.3 Original name graph with transitive references and examples of transformation results. . . 21
3.4 Definition of a recursive helper function that finds all relations of an ID in a name graph. . 22
3.5 Altered version of the NameFix function *find-captures* that is based on the new definition of capture avoidance. 23
3.6 Altered version of the NameFix function *comp-renaming* that computes renamings for sets of related original IDs. 23
3.7 Transformed Java program with transitive references as in Fig. 3.1 (left), and the resulting program when it was correctly handled by the redefined NameFix algorithm (right). 24
3.8 Name graph of the transformed program from Fig. 3.1 (left), and the resulting name graph when it was correctly handled by the redefined NameFix algorithm (right). 24
3.9 Altered version of *find-relations-recursive* that supports extended name graphs. Changes to the version from Fig. 3.5 are marked green. 25

4.1 Example of a Java program with two conflicting field declarations and ambiguous references to them. 28
4.2 Declaration conflict between original and synthesized IDs modeled in a name graph symmetrically (left), asymmetrically (middle), and the fixed version of both graphs (right). . . 28
4.3 Ambiguous references between original and synthesized IDs modeled in a name graph (left) and the fixed version of the graph (right). 29
4.4 Name graphs for the example program from Figure 4.1 (left) and the fixed version (right). 29

5.1 Example of an acyclic dependency graph for a program consisting of seven modules. . . . 33
5.2 The example program from Figure 2.3, divided into two packages with an external reference from the right package to the left one. 33
5.3 Modular name graph for the module *longcounter* from Figure 5.2, depending on the interface of the module *counter*. Inter-module references are marked in blue whereas interfaces that are not actually available as name graphs are marked gray. 34
5.4 Definition of *name-fix-modules* that applies NameFix on a set of modules. 35

5.5 Original name graph (left) and transformed version with inter-module captures (right). . 36
5.6 Altered version of the functions *find-relations*, *find-captures* and *comp-renaming* supporting external references. Changes to the version from Fig. 3.9 are marked green. 37
5.7 Fixing steps for the modular name graph from Fig. 5.5, with virtual fixed name graph (left) and final fixed name graph (right) 38
5.8 Definition of *name-fix-module* using an added function *name-fix-virtual* for computing a capture-free virtual name graph for a module m_t. 38
5.9 Definition of *apply-virtual-graph* that calculates a renaming with minimal effect on exported names. 39
5.10 Modular name graph accessing a renamed interface (left), causing unintended references to be added and intended ones to be lost. Even after NameFix is applied, the name graph (right) doesn't reflect the intended name bindings. 40
5.11 Modified version of *name-fix-module* that undos renamings on interfaces when computing the fixed virtual name graph. 41
5.12 Definition of a function *add-intended-relations* that adds intended references to a module, and modified definition of *apply-virtual-graph* that calls *add-intended-relations*. 42
5.13 Modular name graph with inter-module capture and renamed interface, and fixing steps using modular NameFix as presented. 43
5.14 Extended definition of *select-renaming* that minizes renamings of exported identifiers. . . . 45

6.1 Excerpts of the definitions of the AST nodes *Program* and *ClassDefinition* 48
6.2 Example of a Lightweight Java program and the parsed AST. 49
6.3 Example Lightweight Java program with local variable declaration (left) and result of the unhygienic program transformation to regular LJ code (right). 51
6.4 Name graph of the unhygiencially transformed program from Fig. 6.3, and the resulting Lightweight Java program after NameFix has been applied. 52
6.5 Definitions of the traits *MetaInterface* and *NominalModular* in Scala. 52
6.6 Scala implementation of the function *add-intended-relations* as originally defined in Figure 5.12. 53
6.7 Definition of *ClassInterface* in Scala, representing the meta data for name resolution generated for each Lightweight Java class. 54

1 Introduction

Over the past years, the role of *program transformations* in modern software engineering has shifted drastically and this trend is likely to continue in the future. In the past, the only transformation that found common application was the compilation of program code to assembly or bytecode in a process that was usually presented as an atomic, single-step operation to the user. Today however, program transformations have become increasingly diverse and complex, and a state-of-the-art programming infrastructure demands for increasing granularity and transparency in each transformation step: The ability to add or rewrite code through *Refactorings* is as important for *Integrated Development Environments (IDEs)* as support for a large range of programming languages, often combined within a single project.

New trends like *Domain-Specific Languages (DSLs)* also add to this trend, as it is not reasonable to write compilers and toolchains for each DSL by hand. Instead, the domain-specific code is, often on-the-fly, transformed into generic language code or even into another DSL for further processing. Multi-step program transformations become almost unavoidable when using *Embedded Domain-specific Languages (EDSLs)*, as embedded code needs to be evaluated in the context of the surrounding host language as well as the host language needs to provide support the EDSL.

An issue most program transformations need to cope with is *transformation hygiene*: When a transformation moves, adds or alters identifier names in a program, there is always the possibility of name clashes that lead to unexpected results. These can confuse programmers, break the intention of the code or the transformation, and lead to bugs that are hard to detect and eliminate. Surprisingly, there is still no well-established standard for handling hygiene in transformation engines or IDEs. In practice, real hygiene is only achieved in very specific areas, and solutions are often limited to a single programming language, transformation engine or usage scenario. More often, workarounds or conventions are installed that place restrictions on users and transformation developers with the intent to make hygiene issues improbable but not impossible.

In 2014, Erdweg, van der Storm and Dai presented an algorithm called *NameFix*, which aims to guarantee real hygiene, independent of the used language, and needs only minor support by the transformation engine [1]. However, there are several shortcomings of the algorithm that limit its practical applicability. The needed background knowledge to understand the algorithm, its features and its shortcomings is introduced in Chapter 2 of this thesis.

In Chapter 3, the limitations of NameFix regarding name resolution are addressed and a modified version of NameFix is presented to lift them: On the one hand, the original algorithm reduces identifiers to disjunct declarations and references, while on the other hand allowing only a single binding for each reference. These simplifications are not applicable for many real life scenarios and naive workaround attempts can lead to unexpected, often invalid fixing results.

In languages with a hierarchical name structure like Java, name clashes often lead to conflicting method or field declarations and for this reason to illegal code. Chapter 4 builds on the extensions introduced in the previous section to develop methods for handling such hygiene issues correctly.

Another aspect of hygiene is modularity: In real scenarios, large parts of the program code are usually not accessible to be modified by NameFix. Additionally, when a transformation is applied on a limited section of a program, it would be desirable if changes for hygiene concerns were also limited to the same section. Another problem is the handling of publicly available interfaces, whose alteration can be more significant than modifications of other parts of the program. All these issues are explained and possible solutions are presented in Chapter 5.

To demonstrate the implementation and practical application of the presented concepts and algorithms, a case study on the programming language *Lightweight Java*, which is a subset of the Java programming language [7], is presented in Chapter 6. The chapters 7 and 8 discuss related and future work and conclude this thesis.

2 Background

While the term ”program transformation” covers all sorts of code modifying algorithms like classical compilers, macro engines or template systems, in this thesis it is used especially for those algorithms whose in- and output are supposed to be readable by humans. When compared to classic code compilation, this kind of transformations results in contrary design goals: Compiled code usually drops all symbolic names used in a program, or replaces them by unique IDs that are only mapped back to their original naming by additional meta-data for debugging purposes.

Program transformations however usually need to keep the symbols to keep the resulting code readable and understandable by humans. Even if the actual code is directly compiled after the transformation is applied, symbols are usually required to generate debugging information or to allow feedback from the compiler to be relatable to the original program code.

```java
class Counter {
  int count;

  int addCount() {
    count += 1;
    return count;
  }
}
```

Figure 2.1: Example of a small Java program before any transformations are applied.

```java
class Counter {
  private int count;

  int getCount() {
    return count;
  }

  void setCount(int newCount) {
    count = newCount;
  }

  int addCount() {
    setCount(getCount() + 1);
    return getCount();
  }
}
```

```java
class Counter {
  private int f0001;

  int m0001() {
    return f0001;
  }

  void m0002(int p1) {
    f0001 = p1;
  }

  int m0003() {
    m0002(m0001() + 1);
    return m0001();
  }
}
```

Figure 2.2: Example transformations for the program from Fig. 2.1. Left: transformed using a mix of original and synthesized names; Right: transformed using fresh names for all identifiers.

Figure 2.1 shows an example of a very simple Java class *Counter* that implements a counter of how often the method *addCount* has been called. This example however doesn’t comply to Java best practices as access to the field *count* is not wrapped using getter- and setter-methods. Adding such wrappers is a typical example of a program transformation that is implemented in most Java IDEs[1].

Figure 2.2 shows two example results of transformations that add getters and setters the field *Count*. The transformation resulting in the code on the left generates names by combining the original variable name with a prefix in the same way, a programmer would modify the code manually. The transformation resulting in the code on the right however replaces all symbolic names by unique, fully synthetic names as a classical compiler would do. While both transformations result in semantically equal code, the fully synthetic interface has become almost unusable by a programmer and thereby defeats the purpose of the whole transformation.

2.1 Transformation hygiene

As demonstrated, to achieve transparency and granularity in program transformations, it is often unavoidable to retain original names and even try to generate synthesized names based on them. This however can result in another kind of problem as seen in Figure 2.3: The same transformation as seen in Figure 2.2 was applied here to class *LongCounter*, but now there is a class hierarchy with a super-class that already implements a method named *getCount*.

```java
public class Counter {
    private int count;

    public int getCount() {
        return count;
    }
}

public class LongCounter extends Counter {
    public long count;

    public long addCount() {
        count += 1;
        return count;
    }

    public int getSuperCount() {
        return getCount();
    }
}
```

```java
class Counter ...
// See left as this class was not affected by the transformation

class LongCounter extends Counter {
    private long count;

    long getCount() {
        return count;
    }

    void setCount(int newCount) {
        count = newCount;
    }

    long addCount() {
        setCount(getCount() + 1);
        return getCount();
    }

    int getSuperCount() {
        return getCount();
    }
}
```

Figure 2.3: Example of an extended Java program (left) that is transformed to invalid Java code (right) by a non-hygienic Java program transformation.

Since the *count*-field in the *Counter*-class is private, it is not visible from inside the class *LongCounter*. Therefore, in the original code, the only method in *LongCounter* that refers to the super-class is *getSuperCount*, which calls the method *getCount* that is implemented there.

---

[1] The most prevalent Java IDEs *Eclipse* and *IntelliJ IDEA* both support this feature

The transformed code on the right side of Figure 2.3 would be the result of the most naive approach to the problem: The new synthesized methods are added without any special handling just like they were in Figure 2.2. This leads to a most likely unintended side-effect, since the newly added method *getCount* now overrides the super-class method with the same name. The impact of this can be seen in method *getSuperCount*, that now no longer returns the super-class method's result but the one of the newly added getter method.

Since the original and the inserted method have different return types in the given example, the resulting Java code does no longer compile due to type checking errors. Therefore, the unintended capture may easily be noticed and fixed by the programmer if the transformation was called intentionally. Yet, if the transformation was applied automatically, the problem may become much more difficult to spot. In this case, the error can not be localized in the original code since the program is correctly typed, and even a naive look at the transformation's source code might not allow the problem to be noticed immediately.

Additionally, unintended naming conflicts don't necessarily need to result in invalid code: If the *LongCounter* class also used integer types as return values, there would be no type checker errors and the code would seemingly compile correctly. However, the *getCount*-method would still shadow the super-class method and therefore *getSuperCount* would not work as intended and produce unexpected results when called. For programmers relying on automatic transformations to always work as expected, having to debug such a problem is probably a worst case scenario. Other languages like *Scheme* that allow arbitrary overwriting of already declared identifiers are especially prone to such unintended name overrides.

The avoidance of all possible outcomes of unintended name binding is called *hygiene*. Most research on hygiene was originally conducted with a focus on macro engines, which has lead to approaches that are rather specialized on this area. The idea behind macro hygiene is that all references crossing the borders between a macro definition and the surrounding program are unintended, except for explicitly defined macro parameters. To ensure hygiene, each name synthesized by the macro is checked for such references, and if there are any, they are removed by replacing the synthesized name with a fresh name [3].

```java
class Counter {
  ...
    int getCount() ...
}

class LongCounter extends Counter {
  ...
    long getCount_0() ...

    long addCount() {
        setCount(getCount_0() + 1);
        return getCount_0();
    }

    int getSuperCount() {
        return getCount();
    }
}
```

```java
class Counter {
  ...
    int getCount_0() ...
}

class LongCounter extends Counter {
  ...
    long getCount() ...

    long addCount() {
        setCount(getCount() + 1);
        return getCount();
    }

    int getSuperCount() {
        return getCount_0();
    }
}
```

Figure 2.4: Two solutions for the naming conflict in Fig. 2.3: By renaming the synthesized *getCount*-method (left) and by renaming the original *getCount*-method (right).

The goal of hygiene for macros is to remove any dependencies between the naming of the macro and the surrounding program to ensure that the semantics of a macro don't depend on the naming context it is called in. This can be generalized for all types of program transformations: Thre result of a program transformation should be independent of the naming in the original program. Based on this goal, the following types of references can be considered as unintended:

- References from original names that are altered through the transformation
- References from original names that are added through the transformation
- References from synthesized to original names

In contrast to macros, the first two types of references can also occur between two names from the original program: As the transformation is not limited to adding a continuous segment of code but can completely rearrange the original program, it can also create new references between previously unrelated names. These references are also considered as unintended, since the names were not related previously and the result of the transformation would for this reason differ if names match by coincidence. A program transformation can be considered as *capture-avoiding* if it ensures that none of the listed references can occur for any program to be transformed [1].

The transformed example from Figure 2.3 has an altered reference from the original name in ln. 16 that now points to the name synthesized by the transformation in ln. 8. To avoid this capture, there are two possible fixes as shown in Figure 2.4: The synthesized name can be replaced by a fresh name or the original declaration and reference have to be renamed. Which of both solutions is favorable has to be decided independently for each case. Approaches for trying to avoid the change of externally visible interfaces and propagating changes to external references are presented later in this thesis.

2.2 Name graphs

When an identifier is renamed to solve an unintended capture, it is often necessary to also rename other occurrences of the same identifier to preserve bindings that are actually intended by the user. An example of such a case can be seen in ln. 18 of the left renaming in Figure 2.4, where the call of the method *getCount* needs to be renamed by giving it the same name as the renamed method declaration.

In most programming languages, not all identifiers with the same name can automatically be considered as related. So, to actually detect intended or unintended name bindings, it is necessary to apply the individual scoping and name lookup rules of the used language. As transformations may even transfer the original program to a different language, the lookup and comparison of references before and after a transformation was applied can become even more complicated. To avoid the development of a specialized algorithm that can only be applied to a very limited set of transformations, language independence is an important design goal.

A language independent representation of identifiers and their bindings is achieved by *name graphs*. A name resolution algorithm that is dependent on the actual programming language and its scoping rules is used to transform a representation of the program into a language neutral name graph. A name graph G is formally defined as a pair (V, ρ), with the identifiers' *IDs* as nodes V and their references as edges ρ, modeled through a partial function $\rho \in V \mapsto V$. If an ID is referenced by another one, a string-wise equality of both names is automatically implied [1].

The usage of IDs as nodes has an important role in the name graph model: To allow a comparison of two name graphs, there needs to be a way to uniquely identify identifiers across multiple evaluations of the algorithm on the same program or an original program and its transformed version. This obviously can not be achieved by just comparing the name strings of two identifier nodes, since completely unrelated nodes may still have the same name string. Instead, an additional ID is required to represent the same identifier in two different name graphs.

For transformations within the same programming language, the most straight-forward approach to implement IDs for identifiers is to use a pointer or other internal representation of the identifier, like its node in the program's *abstract syntax tree (AST)*. For illustrative purposes, the string naming and line number at which an identifier occurs in the corresponding program are used as IDs for all name graph figures in this thesis.

Name graphs generated for the example Java program from Figure 2.1 and its transformed version from Figure 2.2 (left) can be seen in Figure 2.5. Nodes and edges added by the transformation are printed dotted in the right graph and their line number from the transformed program has an added tick. This is however only done for better clarity and there is no actual difference between the nodes when only looking on the name graph itself.

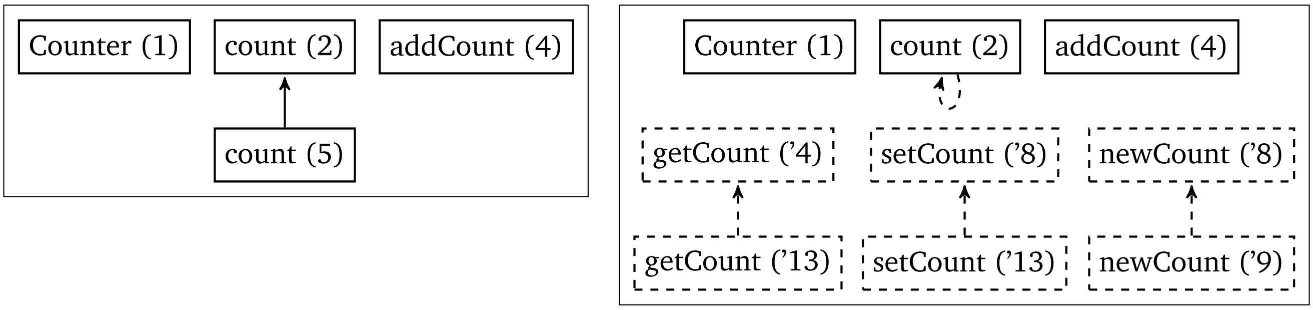


Figure 2.5: Name graphs for the example program from Fig. 2.1 (left) and the first transformed program from Fig. 2.2 (right). Nodes/edges added by the transformation are dotted.

When a program transformation is applied, it also has to fulfill another requirement that is already assumed in Figure 2.5: Original nodes are not re-defined by the transformation and therefore still keep their original ID. This means that they can be identified across multiple transformation steps and, as an additional result, nodes added by a program transformation can easily be identified by calculating the difference between the nodes of the transformed and the original name graph.

Another detail in Figure 2.5 that will become important later on are the missing nodes for the added identifiers *count* in ln. 5 and 9 of the transformed program. Here, the transformation copied the original identifier *count* from ln. 2 to create the intended references to it. For such intended copies, the original ID of the copied node needs is preserved. This results in the copies not appearing in the name graph and their references being represented by a single self-reference for the original node. While this approach may seem unintuitive at first, it is important, since it is the only way to explicitly model intended references from synthesized to original names.

In general, the name graph representation might not seem very favorable, since it is not trivial to preserve all the constraints presented above for each program transformation, especially for cross-language transformations that need to cope with different name resolution algorithms. The advantage of name graphs however is that once they are generated, they are completely independent of the language a program is written in. This not only allows language-independent analysis of name bindings but also eliminates any need for further handling of cross-language transformations.

2.3 NameFix

Erdweg, van der Storm and Dai presented an algorithm called *NameFix* that can provide a language- and transformation-independent guarantee of transformation hygiene [1]. This algorithm only requires the ability to generate name graphs for the original and target language as presented in Section 2.2, as well as interfaces for the target language to generate fresh, legal names based on existing ones and to actually apply calculated renamings. The pseudo-code implementation of NameFix can be seen in Figure 2.6[2].

```
Syntactic convention:
  $t^{@v} = x$    name $x$ of the identifier $v$ in program $t$

$\mathit{find\text{-}captures}((V_s, \rho_s), (V_s, \rho_t)) = \{$
  notPresrvRef1 $= \{\rho_t(v) \mid v \in \mathit{dom}(\rho_t), v \in V_s, v \in \mathit{dom}(\rho_s), \rho_s(v) \neq \rho_t(v)\}$;
  notPresrvRef2 $= \{\rho_t(v) \mid v \in \mathit{dom}(\rho_t), v \in V_s, v \notin \mathit{dom}(\rho_s), v \neq \rho_t(v)\}$;
  notPresrvDef $= \{\rho_t(v) \mid v \in \mathit{dom}(\rho_t), v \notin V_s, \rho_t(v) \in V_s\}$;
  return notPresrvRef1 $\cup$ notPresrvRef2 $\cup$ notPresrvDef;
}

$\mathit{comp\text{-}renaming}((V_s, \rho_s), (V_s, \rho_t), t, V_{rename}) = \{$
  $\pi = \emptyset$;
  foreach $v$ in $V_{rename}$ {
    usedNames $= \{t^{@v_0} \mid v_0 \in V_t\} \cup \mathit{codom}(\pi)$;
    fresh $= \mathit{gensym}(t^{@v}$, usedNames$)$;
    if $(v \in V_s \wedge v \notin \pi)$
      $\pi = \pi \cup \{(v \mapsto$ fresh$)\} \cup \{(v_r \mapsto$ fresh$) \mid v_r \in \mathit{dom}(\rho_s), \rho_s(v_r) = v_d\}$;
    if $(v \notin V_s \wedge v \notin \pi)$
      $\pi = \pi \cup \{(v_0 \mapsto$ fresh$) \mid v_0 \in V_t \setminus V_s, t^{@v_0} = t^{@v}\}$;
  }
  return $\pi$;
}

$\mathit{name\text{-}fix}(G_s, t) = \{$
  $G_t = \mathit{resolve}(t)$;
  capture $= \mathit{find\text{-}captures}(G_s, G_t)$;
  if (capture $== \emptyset$) return t;

  $\pi = \mathit{comp\text{-}renaming}(G_s, G_t, t,$ capture$)$;
  $t' = \mathit{rename}(t, \pi)$;
  return $\mathit{name\text{-}fix}(G_s, t')$;
}
```

Figure 2.6: The *name-fix* algorithm developed by Erdweg, van der Storm and Dai.

The core idea of NameFix is the detection of unintended captures by comparing the pre- and post-transformation name graph. In the pseudo-code in Figure 2.6, this is implemented in the function *find-captures*. References are assumed as unintended based on the definition given in Section 2.1: The set *notPresrvRef1* contains edges from original names that were altered by the transformation, *notPresrvRef2* contains added edges and *notPresrvDef* contains synthesized names referencing original names. For the calculation of *notPresrvRef2*, self-references are explicitly excluded as they model intended references from synthesized to original identifiers as explained in Section 2.2.

---

[2] The version of NameFix in Figure 2.6 was slightly altered compared to its original definition by Erdweg, van der Storm and Dai to allow better re-usage of functions in the extensions presented later. The semantics of NameFix are not affected by this alterations.

After NameFix has detected all references matching one of the cases listed above, it calculates a map of renamings for the captured variables as seen in function *comp-renaming*: It calls the interface provided by the programming language to generate a fresh name for each of the captured nodes. Since there might be intended references to the node, they need to be also looked up. For original names this can be done by just searching references to the name in the original name graph, as those are assumed to be intended. Synthesized nodes however can not be looked up, so for those, a string-wise naming comparison with other synthesized nodes is the only way to find possibly intended references. As already mentioned in Section 2.2, unrelated IDs may be addressed this way. However, as all other synthesized IDs with the same naming are also addressed, the result would only be an unnecessary renaming that wouldn't affect the program's semantics.

One constraint that needs to be satisfied to allow NameFix to work as intended is that name graphs are required to be bipartite. This will be addressed in detail in Chapter 3. A consequence of this constraint is however that each name is required to be either a reference or a declaration, but never both at once. This makes it easy to calculate nodes with intended references that need to be preserved, since they all need to directly point to the first renaming.

After the map of renamings has been computed, the interface of the programming language is called in order to actually rename the identifiers according to the renaming map. Finally, NameFix recursively runs again on the resulting, partially fixed program. The recursion is required since the renaming might have caused new hygiene issues that were previously hidden by the ones now fixed. This is also the reason why a newly generated name graph is used in each recursion step. Recursion only stops if there are no more captures matching one of the three problematic cases are found. Then, the result of the last renaming is returned.

Figure 2.7 shows the name graph of the unhygienically transformed program from Figure 2.3 on the left. For this example, NameFix would detect the unintended reference from the method call of *getCount* in ln. 18 of the original program to the synthesized declaration in ln. 7 of the transformed program. Consequently, the captured declaration is scheduled for renaming and, as the declaration is synthesized, all other synthesized identifiers with the same name are also renamed accordingly. The resulting name graph can be seen on the right side of Figure 2.7: As the synthesized declaration was renamed, the intended reference to the original declaration is restored. The resulting program would be equivalent to the one on the left side of Figure 2.4.

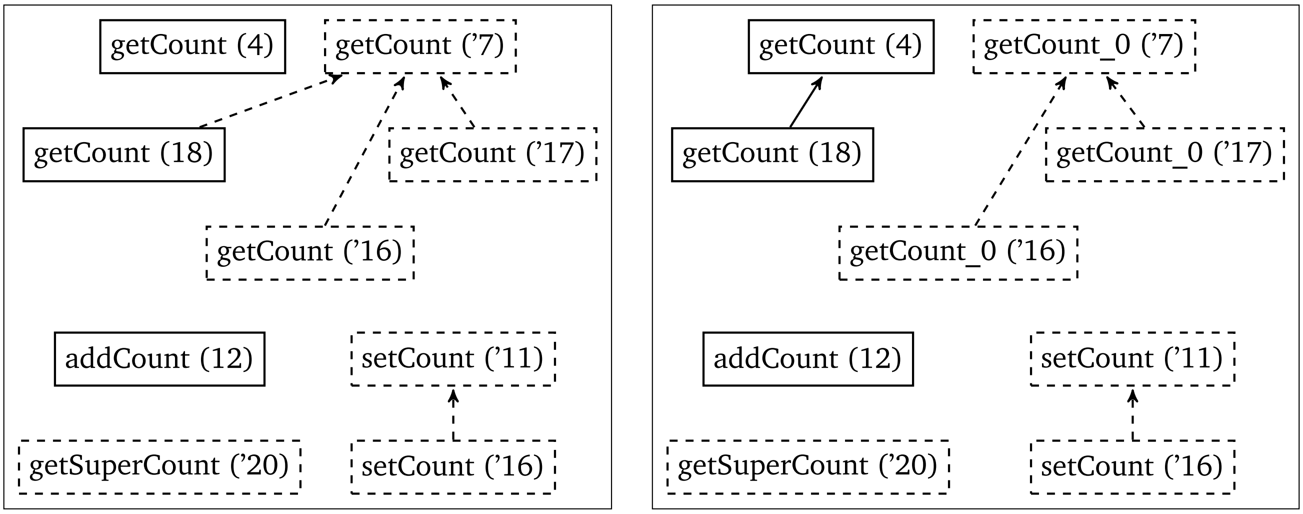


Figure 2.7: Name graphs for the transformed example program from Fig. 2.3 (left) and the result after NameFix is applied. Class/field names are not included for clarity reasons.

As already mentioned in this and the previous sections, there are several shortcomings of NameFix:

- Name graphs are required to be bipartite, resulting in the inability to handle transitive dependencies between names.
- It is essential for NameFix that name graphs are available for the original and the transformed program. This however might not always be the case, since generation of name graphs for invalid code is not specified.
- NameFix operates based on the assumption that all names of a program can be modeled into a single name graph and that all names are available to renaming. In reality however, there are often restrictions like shared libraries or pre-compiled code that cannot be renamed.

In the next sections, this thesis is going to address each of these shortcomings and present ways to improve the existing algorithm to handle them.

3 Hygiene for transitive name graphs

The name graph model as introduced in Section 2.2 doesn't enforce a separation of nodes to declarations and references. As an ID can be in the domain and the co-domain of the reference function ρ, transitive references can be implemented rather intuitively. Yet, the NameFix algorithm introduced in Section 2.3 strictly requires name graphs to be bipartite as a precondition.

There are some widely used programming language features that require identifiers to have both roles: Method overriding, present in Java and many other object oriented languages, requires the declaration of the overriding subclass method to provide a reference to the overridden super-class method to make sure their relation isn't lost when renaming one of the methods. However, the method still needs to have the status of a declaration as it is referenced itself by all calls of the method. Other problematic examples are method overloading or the usage of constructor methods that need to match the name of their class. In general, all language features that require several definitions implicitly share the same naming cannot be represented by a bipartite name graph.

A reasonable attempt to introduce transitive references between names would be to just apply the version of NameFix as presented in Section 2.3 to name graphs that aren't bipartite. The example program on the left side of Figure 3.1 uses method overriding in Java and for this reason results in a name graph with a transitive reference as seen on the left side of Figure 3.2.

```
 1  class A {
 2    // Added by program transformation
 3    void method() ...
 4  }
 5
 6  class B extends A {
 7    // From in the original program
 8    void method() ...
 9  }
10
11  class C extends B {
12    // From in the original program
13    void method() ...
14
15    void anotherMethod() {
16      // From the original program
17      method();
18      // Added by program transformation
19      method();
20    }
21  }
```

```
 1  class A {
 2    // Added by program transformation
 3    void method() ...
 4  }
 5
 6  class B extends A {
 7    // From in the original program
 8    void method_1() ...
 9  }
10
11  class C extends B {
12    // From in the original program
13    void method_0() ...
14
15    void anotherMethod() {
16      // From the original program
17      this .method_0();
18      // Added by program transformation
19      this .method();
20    }
21  }
```

Figure 3.1: Java program with transitive references (left), and the resulting program when it was insufficiently handled by the NameFix algorithm (right).

The program from Figure 3.1 is the result of an unhygienic transformation: The first function call in *anotherMethod* in ln. 17 is supposed to call the declaration of *method* in ln. 13 of the original program. The second call in ln. 19 however was inserted by a transformation and should point to the also inserted declaration of *method* in ln. 3. Since the original declaration unintendedly captures the reference, both

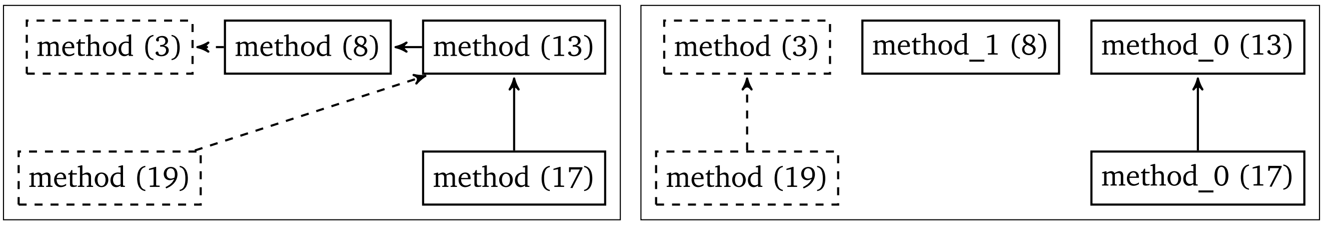


Figure 3.2: Name graphs for the example program from Fig. 3.1 (left) and the version insufficiently renamed by NameFix (right). Synthesized nodes/edges are dotted, class names are not included for clarity reasons.

calls are referencing the original declaration in ln. 13. When NameFix is applied to the program, it should fix the captured reference and ensure that all method calls reference their intended declaration.

An actual application of NameFix detects two unintended captures: The first one is the method call in ln. 19 unintendedly referencing the declaration in ln. 13. The second one is the unintended override reference from ln. 8 to ln. 3. The first capture results in the declaration in ln. 13 and all original references to it being renamed to a fresh name (here assumed as *method_0*). Since the target of the second capture is already to be renamed, this capture is skipped.

In the second iteration however, another capture appears as the declaration in ln. 8 was not renamed and therefore still shadows the synthesized declaration in ln. 3. This results in another renaming of this declaration to a fresh name (here assumed as *method_1*). This result is capture-free and returned by the algorithm. The resulting program and name graph can be seen on the right sides of Figures 3.1 and 3.2.

The result, while being capture-free, doesn't reflect the intended name bindings: The declaration in ln. 14 no longer overrides the one in ln. 8, since they were renamed to different fresh names. This outlines the problem of the current specification of NameFix: Since only direct references to the target of an unintended capture are added to the renaming, but neither references from the target nor indirect references, intended original references can get lost.

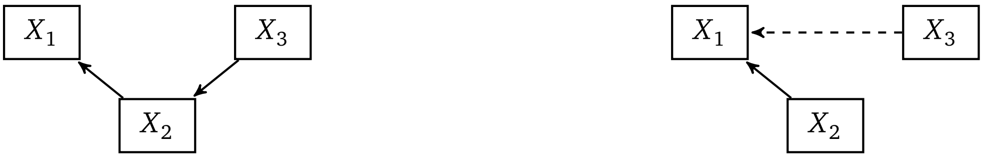
While it would be rather simple to extend NameFix to also rename indirect references, there are several other cases that need to be considered. As shown in detail in the next section, there are situations where a naively modified version of NameFix can encounter a conflict of interests, as names can be related intendedly and through an unintended capture at the same time. To ensure termination and reasonable results for all these cases, it is necessary to evaluate and redefine the original definition of capture avoidance.

3.1 Capture-avoidance for transitive name graphs

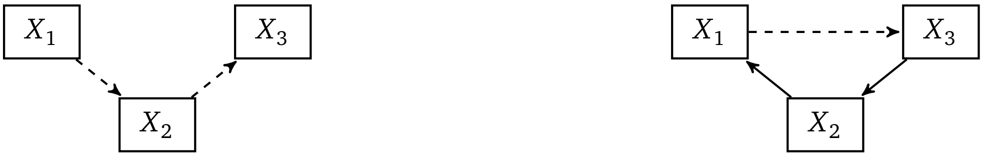
In many cases, like the one that was presented in Figure 3.1, the handling of transitive references is rather intuitive: All original relations to a captured node need to be considered when it is renamed. Yet, it is possible to construct special cases of rather simple transformations that lack an intuitive solution when using the current definition of capture avoidance. Some of these cases are illustrated in Figure 3.3.

Figure 3.3a shows the original name graph of an example program. The case which is likely the most relevant for real transformations is presented in Figure 3.3b. In this transformed name graph, the reference from X_3 to X_2 is altered to point to X_1. This alteration is obviously a case that would be detected by NameFix as an unintended reference that needs to be fixed. However, when looking at the original name graph in Figure 3.3a, there was already a transitive relation between the IDs X_3 and X_1. With the previous assumption that originally related IDs need to be considered when the captured ID is renamed, this results in all three nodes being always renamed together. The effect of this is that the capture of X_1 is impossible to fix without also breaking originally intended relations.

Finding a way to handle such a special case becomes even more difficult assuming the examples from Figure 3.3c and 3.3d: A reversed declaration-reference relationship between two IDs would be



(a) Original name graph **(b)** Transformed name graph with direct reference



(c) Transformed name graph with reversed references **(d)** Transformed name graph with cycle

Figure 3.3: Original name graph with transitive references and examples of transformation results.

considered as unintended by the original NameFix algorithm. Transferred to transitive references, this would require all nodes in Figure 3.3c to be separated. Figure 3.3d however shows another example that, while not likely to occur in real transformations, is not explicitly prohibited: If an additional reference is added causing a reference cycle, it is impossible to remove this capture without also removing at least one of the two other, legitimate references.

Instead of finding a complex solution that regards all possible scenarios resulting from transitive references with special solutions, it is a better approach to revisit the definition of capture avoidance and its intention: While none of the transformations in Figure 3.3 satisfies the definition of capture avoidance, they still all preserve the original intention of hygiene: If all related occurrences of X in the original name graph are consistently renamed, the relations in the transformed name graphs are not affected in any way. This leads to the conclusion that none of the added or altered references in Figure 3.3 are actual unintended captures, and therefore they don't need to be fixed.

Before actually redefining capture avoidance, the concept of IDs being related, that was only specified informally up to this point, needs to be formalized:

Definition 1. Let $G = (V, \rho)$ be the name graph of a program p, resolved by $G = resolve(p)$.

1. In G, an ID $v_1 \in V$ is *related* to an ID $v_2 \in V$ if there is a path from v_1 to v_2 under the assumption that all edges $\in \rho$ are bidirectional.
2. The function $rel_G : V \mapsto \mathscr{P}(V)$ maps each ID v to the set of all IDs related to v in G.

Based on this definition, the nodes X_1, X_2 and X_3 are related in all examples presented in Figure 3.3. Accordingly, $rel_G(X_1) = rel_G(X_2) = rel_G(X_3) = \{X_1, X_2, X_3\}$ for all the name graphs. In general, since the definition of related IDs is transitive, symmetric and reflexive, it divides the set of nodes in a name graph into equivalence classes, and the function rel_G is the projection of this equivalence class.

Using this definition, it is possible to create a new definition of capture avoidance, that is based on a comparison of relations instead of references.

Definition 2. A transformation $f : S \mapsto T$ is *capture-avoiding* if for all source programs $s \in S$ with name graphs $G_s = (V_s, \rho_s)$, and all target programs $t \in T$ with name graphs $G_t = (V_t, \rho_t)$: $\forall v \in V_t : v \in V_s \Rightarrow rel_{G_t}(v) \setminus rel_{G_s}(v) = \emptyset$

The intention of the definition is that for each ID from the original name graph that is still present in the transformed name graph, the scope of relations should not have been extended by the transformation. This is checked by subtracting the original scope from the transformed scope and requiring the result to be empty. Consequently, all transformations in Figure 3.3 are already capture-avoiding if they are only defined for the example program from from Figure 3.3a. This can be easily verified as none of the relations of the source nodes was altered, and the difference between the transformed and the original name graph is consequently always the empty set.

Especially for name graphs without transitive references, the new definition is similar to the original:

- References from original IDs that are altered through the transformation are still in the definition of capture avoidance if they refer to IDs not related in the original name graph. For references to synthesized IDs, this is always the case, since they were not part of V_s and therefore not of the codomain of rel_{G_s}. If other IDs from the original name graph were already related with an ID, added references to them are now no longer seen as unintended captures, meaning that the new definition of capture-avoidance is weaker than the previous one. Previously unrelated IDs however are still detected as they are part of $rel_{G_t}(v) \setminus rel_{G_s}(v)$.
- References from original IDs that are added through the transformation are handled equally to altered references. However, the explicit exception of IDs being allowed to have an added reference to themselves is now no longer necessary, as IDs are always related to themselves.
- References from synthesized to original IDs are a special case: As the definition of capture avoidance only makes a statement for original IDs, synthesized ones seem to be allowed to have any kind of relations. This is however not the case as relations are always symmetrical and for this reason, a reference from a synthesized ID to an original ID would also yield a relation between the original and the synthesized ID.

Using the new definition of capture avoidance, it is possible to also redefine the NameFix algorithm accordingly to ensure capture avoidance for transformations based on the presented definitions.

3.2 NameFix for transitive name graphs

The original definition of NameFix was closely tied to the original definition of capture avoidance. With the new definition from Section 3.1, there are two areas that require modifications: The detection of unintended captures has to conform to the new definition, and when renamings for an ID are scheduled, all related IDs need to be scheduled as well. However, determining the relations of an ID is not a completely trivial task, and it has to be considered first as it is foundation for all upcoming modifications.

```
$find\text{-}relations(n, G) = find\text{-}relations(n, G, \emptyset)$

$find\text{-}relations(n, (V, \rho), R) = \{$
  $R' = R \cup n;$
  if $(n \in dom(\rho) \land \rho(n) \notin R')$ {
    $R' = R' \cup find\text{-}relations(\rho(n), (V, \rho), R');$
  }
  foreach $v$ in $V$ {
    if $(v \notin R' \land v \in dom(\rho) \land \rho(v) == n)$
      $R' = R' \cup find\text{-}relations(v, (V, \rho), R');$
  }
  return $R';$
}
```

Figure 3.4: Definition of a recursive helper function that finds all relations of an ID in a name graph.

A definition of a function that calculates all relations of an ID n in a name graph G can be seen in figure 3.4 with the name *find-relations*. Since the function is recursive and needs an accumulator as argument, an overloaded wrapper function is also defined, that allows a more intuitive function call without an empty accumulator. The actual implementation of *find-relations* is rather simple: It recursively follows outgoing and incoming references of the current ID using depth-first search and returns the accumulated set of all processed IDs.

Since name graphs are always finite and the whole chain of related IDs has to be processed before the algorithm can return, the actual implementation of the function is not limited to the presented one or depth-first search in general. As the function will be used quite extensively in the new version of NameFix, and the set of related IDs is equal for all other IDs that are returned as part of the relation, calculating the equivalence classes for a name graph once and using a cache afterwards would most likely lead to drastic performance enhancements. Yet, real name graphs don't tend to be overly complex in most cases, especially when they are modularized as will be shown in Chapter 5. Accordingly, the implementation of the algorithms presented in this thesis is focussed on clarity instead of performance.

```
$find\text{-}captures((V_s,\rho_s),(V_s,\rho_t)) = \{$
  $\text{captureNodes} = \{v \mid v \in V_t, v \in V_s, find\text{-}relations(v,(V_t,\rho_t)) \setminus find\text{-}relations(v,(V_s,\rho_s)) \neq \emptyset\};$

  return $\text{captureNodes};$
$\}$
```

Figure 3.5: Altered version of the NameFix function *find-captures* that is based on the new definition of capture avoidance.

Using the function *find-relations*, it is possible to extend NameFix to support transitive references. Figure 3.5 shows an altered version of the function *find-captures*, which is used to find captured nodes in NameFix. Like the original function, the implementation is based strictly on the definition of capture-avoidance and finds all references violating this definition. It should be noted that, contrary to the original definition of *find-captures* as in Figure 2.6, the returned set doesn't contain the captured IDs, but instead all original IDs that are related to a capture. As a result, this modification is not compatible with the original version of the function *comp-renaming*, which will be replaced in the next step.

```
$comp\text{-}renaming(G_s,(V_t,\rho_t),t,V_{rename}) = \{$
  $\pi = \emptyset;$

  foreach $v$ in $V_{rename}$ {
    if $(find\text{-}relations(v,(V_t,\rho_t)) \cap \pi == \emptyset)$ {
      $\text{usedNames} = \{t^{@v_0} \mid v_0 \in V_t\} \cup codom(\pi);$
      $\text{fresh} = gensym(t^{@v}, \text{usedNames});$
      $\text{relatedNames} = find\text{-}relations(v, G_s);$
      $\pi = \pi \cup \{(v_0 \mapsto \text{fresh}) \mid v_0 \in \text{relatedNames}\};$
    }
  }
  return $\pi;$
$\}$
```

Figure 3.6: Altered version of the NameFix function *comp-renaming* that computes renamings for sets of related original IDs.

The function *comp-renaming* also needs to be modified to not accidentally break indirect relations between identifiers. In the altered version in Figure 3.6, the dependencies are computed by again using *find-relations* on the original name graph. Since *find-captures* now always returns the non-synthesized identifiers related to an unintended capture, we no longer need to handle the renaming of synthesized identifiers at all. On the other hand, to prevent unnecessary renamings, it has to be checked if the current node or any identifier related to the capture was already scheduled for renaming. If this is the case, the renaming has to be skipped until the remaining captures have been calculated the next iteration of NameFix

As for the definition of *find-relations*, it is obvious that the performance of the presented implementations is not optimal since the relations between nodes, that were already calculated in the function *find-captures*, are discarded and calculated again in the function *comp-renaming*. Yet, this allows the rest of the original definition of NameFix to be reused without additional modifications.

With the modifications to the functions presented here, there is no need for additional changes to the core structure of NameFix: If declarations are renamed, it is still possible that more unintended captures are revealed that were hidden by the first capture. Therefore, it is still required to repeat the search for captures recursively.

```
 1  class A {
 2    // Added by program transformation
 3    void method() ...
 4  }
 5
 6  class B extends A {
 7    // From in the original program
 8    void method() ...
 9  }
10
11  class C extends B {
12    // From in the original program
13    void method() ...
14
15    void anotherMethod() {
16      // From the original program
17      method();
18      // Added by program transformation
19      method();
20    }
21  }
```

```
 1  class A {
 2    // Added by program transformation
 3    void method() ...
 4  }
 5
 6  class B extends A {
 7    // From in the original program
 8    void method_0() ...
 9  }
10
11  class C extends B {
12    // From in the original program
13    void method_0() ...
14
15    void anotherMethod() {
16      // From the original program
17      this.method_0();
18      // Added by program transformation
19      this.method();
20    }
21  }
```

Figure 3.7: Transformed Java program with transitive references as in Fig. 3.1 (left), and the resulting program when it was correctly handled by the redefined NameFix algorithm (right).

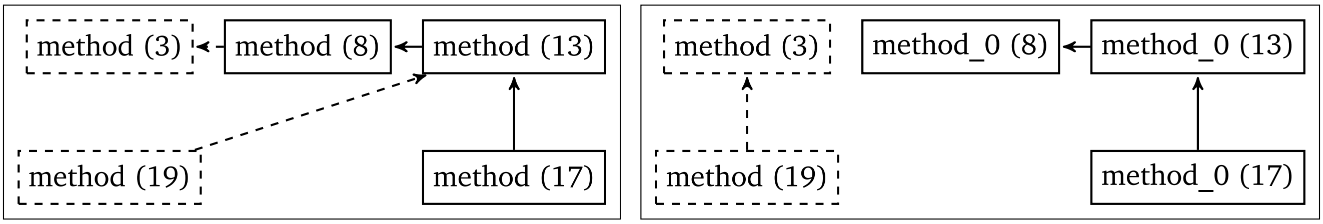


Figure 3.8: Name graph of the transformed program from Fig. 3.1 (left), and the resulting name graph when it was correctly handled by the redefined NameFix algorithm (right).

The Figures 3.7 and 3.8 show the already presented example program and name graph with transitive references, and the results after the example from Figure 3.1 was processed by the new version of

NameFix. The declarations of *method* in ln. 8 and ln. 13, as well as the method call in ln. 17 are related in the original name graph. After the transformation, the declaration in ln. 3 and ln. 19 are added to this relation. So, the three original IDs are scheduled for renaming, which results in the name graph seen in Figure 3.8. Since after this step, the relations of the original IDs are equal for both graphs, the resulting program as in Figure 3.7 is returned.

3.3 Name graphs and NameFix for multi-referencing names

The new definition of capture avoidance presented in this section also allows another prerequisite of the original NameFix algorithm to be dropped: In the current name graph model, each ID can have only one outgoing reference because references are modeled as a function ρ, which only maps one node of the graph to one other node. This can however be a restriction to name resolution algorithms for real programming languages, as there might be more complex or diverse types of relations between names that need to be represented in the name graph.

To overcome this limitation, the current model of name graphs needs to be extended. There are several ways how to model edges in graphs, the most common one being a relation between nodes or just a set of node pairs. However, the mayor design goal here is to make it easier to collect all outgoing references of an ID and to reduce the amount of modifications required to NameFix and the function *find-relations* introduced in Chapter 3. So, the most fitting model is redefining ρ so that it maps IDs to sets of referenced IDs:

Definition 3. The *extended name graph* G_{ex} of a program p, resolved by the function $G_{ex} = resolve_{ex}(p)$, is a pair (V, ρ) where V is the set of identifier IDs in p, $\rho \in V \mapsto \mathscr{P}(V)$ is a partial function from IDs to their sets of referenced IDs, and if $v_d \in \rho(v_r)$, then the identifiers represented by v_r and v_d have equal string-wise namings.

Since the previous definitions in this section are not directly using the concept of references but only paths within the name graph, they are also applicable to extended name graphs. Especially, having multiple outgoing references per ID doesn't affect the ability to divide the nodes in the name graph into equivalence classes based on their relations to each other. As the relation-based version of NameFix already abstracts from references to these equivalence classes, the changes required to the algorithm are also going to be rather minor.

```
$find\text{-}relations(n, (V, \rho), R) = \{$
  $R' = R \cup n;$
  if $(n \in dom(\rho))$ {
    foreach $v$ in $(\rho(n) \setminus R')$
      $R' = R' \cup find\text{-}relations(v, (V, \rho), R');$
  }
  foreach $v$ in $V$ {
    if $(v \notin R' \wedge v \in dom(\rho) \wedge n \in \rho(v));$
      $R' = R' \cup find\text{-}relations(v, (V, \rho), R');$
  }
  return $R';$
}
```

Figure 3.9: Altered version of *find-relations-recursive* that supports extended name graphs. Changes to the version from Fig. 3.5 are marked green.

To use the extended name graph definition, the most obvious change required for NameFix is the replacement of the resolution function *resolve* by the extended resolution function $resolve_{ex}$. Since the altered definitions of the NameFix functions *find-captures* and *comp-renamings* as presented in Figures 3.5 and 3.6 in Chapter 3 don't access ρ directly but use the helper function *find-relations* instead, this is the only point of NameFix that requires additional modification. Yet even there, only minor changes are required, as can be seen in Figure 3.9, showing the modified version of the function.

Both the ability to handle transitive references and multiple outgoing references per ID are huge improvements on the expressivity of name graphs. When combined, they allow almost any real-life resolution scenario to be expressed. This provides a foundation for all the further improvements and extensions presented in the following sections of this thesis.

4 Hygiene for programs with name resolution issues

As explained in Section 2.2, an interface for the generation of name graphs for the original and the transformed program needs to be provided by the transformation engine. Yet, this generation might not be possible in all situations, depending on the applied transformation and the used programming language: A program transformation is not guaranteed to always generate legal code that allows unambiguous name resolution or even the recognition of all identifiers found in the program.

Program transformations can, similar to macro systems, be categorized by their abstraction level as either being character-, token-, syntax- or semantics-based. While character- and token-based transformations work on the plain program text or its tokenized form, syntax transformations work on the already parsed *Abstract Syntax Tree (AST)* of the program. Semantic transformations can additionally consider type or identifier lookup information [4]. Hygiene issues can occur on every abstraction level, while lower levels are more prone to them, as conflicts with language keywords or mixed-up parts of identifier names can occur. NameFix however works on the highest, semantic level, as it depends on name resolution information. Therefore, only transformations working on this level can guarantee name graph generation to be possible at all times.

Character- and token-based transformations can break the transformed code so that not even the generation of an AST is possible any more. Such issues, even if they can be related to hygiene, are almost impossible to fix using only the original and the transformed program, and would require completely different fixing attempts that work on character- and token-level themselves. Additionally, the issues of such low-level transformations have already be extensively researched and syntax-based transformations are an practicable and effective solution for most of them [4]. For this reason, this thesis will only focus on hygiene issues resulting in programs with legal syntax but invalid name lookup semantics.

Regarding name lookup for a syntactically correct program, there are three types of issues preventing a possible lookup:

1. Declarations can be *conflicting*, meaning that they are declaring the same name in the same context. This is not necessarily an issue in all cases as there might be precedence rules explicitly allowing the overwriting of other names. However, many common languages, like Java or C++, require identifier names to be unique, at least in their direct context.
2. References can be *ambiguous*, meaning that there are multiple declarations in their lookup context they could be referencing. This case is usually the result of a declaration conflict, since most languages have a clearly specified hierarchy of precedence for all other cases. Language features like *Generics* in Java or optional method parameters in *Scala* can however cause ambiguity without conflicting declarations.
3. References can be *unbound*, meaning that they are semantically supposed to reference another identifier, but there is no fitting declaration in their lookup context they could be bound to.

Declaration conflicts and ambiguous references can be the direct result of an unintended capture, meaning that two or more declarations unintendedly share the same name and are consequently in a conflict or cause ambiguities for references to them. Unbound references on the other hand can only be an additional result of another hygiene issue, for example if their supposed binding is already blocked by another reference.

Since there is no way to directly rebind an unbound reference by renaming if it already shares the same name with the intended declaration, this case is not directly solvable without altering the program's structure. Therefore, this case is not directly solvable by NameFix and consequently not explicitly explained in this chapter. There are however possible situation where NameFix also fixes unbound references as a side-effect.

Figure 4.1 shows an example of a transformation result that is no longer legal Java code, as the name of the field *value* is no longer unique, resulting in a declaration conflict. Additionally, the assignments to the field in ln. 9 and ln. 11 could reference both declarations and are therefore ambiguous.

```
 1  class Number {
 2    // From the original program
 3    int value;
 4    // Added by the program transformation
 5    int value;
 6
 7    Number (int a, int b) {
 8      // From the original program, supposed to assign the original field
 9      value = a;
10      // Added by the program transformation, supposed to assign the synthesized field
11      value = b;
12    }
13  }
```

Figure 4.1: Example of a Java program with two conflicting field declarations and ambiguous references to them.

Representing such a situation using the original name graph model as presented in Section 2.2 would have been difficult, especially since other, non-ambiguous references between names would still need to be represented. However, the extensions to name graphs shown in the previous section provide enough expressivity to model and fix hygiene-based declaration conflicts and ambiguities.

Declaration conflicts can be seen as just a new way of names being related. Consequently, it makes sense to use references as connections between conflicting IDs to represent their relation. While it would be sufficient to just let one of the conflicting IDs point to all the others, it might be easier for name resolution algorithms to let each ID point to all other conflicting names symmetrically. As the number or direction of connections between IDs doesn't have an effect on their relation status, both approaches are equally sufficient.

Figure 4.2 shows name graphs with a more complex conflict that effects four IDs. On the left side, the conflict is modeled symmetrically by adding references between each pair of nodes. In the middle, the asymmetrical version is illustrated with only one node arbitrarily selected to reference all other conflicting IDs. As explained, both versions are equivalent in their result, establishing a relation between all four IDs. However, it becomes clear that the symmetrical version can become unnecessarily complex for larger conflicts.

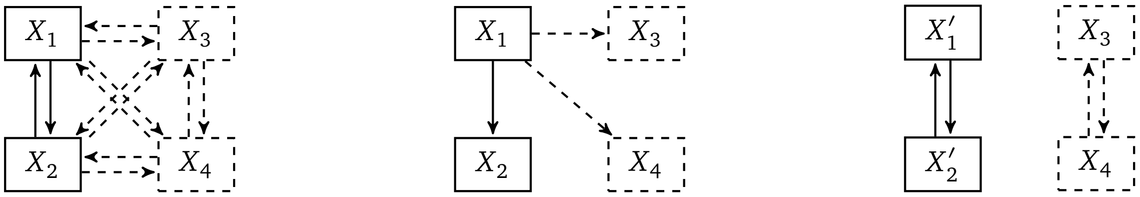


Figure 4.2: Declaration conflict between original and synthesized IDs modeled in a name graph symmetrically (left), asymmetrically (middle), and the fixed version of both graphs (right).

In both versions, NameFix finds the extended set of relations for the original nodes X_1 and X_2 and, since they were already connected in the original name graph, renames the identifiers accordingly to the same fresh name X'. The resulting name graph is shown on the right side of Figure 4.2, modeled as a symmetrical version. Obviously, not all conflicts between the nodes have been solved, but instead, the conflict was divided into two smaller conflicts.

While this may not seem to be the desired result of NameFix since the program is still invalid, the task of NameFix is only to fix hygiene-related issues. As the original IDs X_1 and X_2 were already part of a declaration conflict in the original name graph, symbolized by the original references between them, there already was a relation between them. Consequently, NameFix has no reason to separate this relation and preserves it when applying renamings. The same holds true for the two synthesized IDs X_3 and X_4 and their conflict. As both nodes and their relation were added by the transformation, the conflict is not the result of a hygiene issue but more likely of a faulty transformation. As NameFix has no way to differentiate between legal references and illegal declaration conflicts, it can only fix hygiene-based declaration conflicts while preserving all other conflicts.

Ambiguous references can also be modeled rather intuitively using extended name graphs. Instead of arbitrarily selecting one of the potential declarations, both can be referenced at once, representing the potential relation between the ambiguous reference and the possible declarations. For ambiguities that are caused by declaration conflicts, it is especially important that they are still modeled this way and not ignored to make sure that all intended references are preserved.

The result of ambiguity modeled as described can be seen on the left side of Figure 4.3. The references of the IDs X_5 and X_6 can not be resolved to either of the declarations X_1, X_2, X_3 or X_4. Consequently, all potential references are added to the name graph. In this example, the declarations are not in a conflict with each other, as there are no references between them.

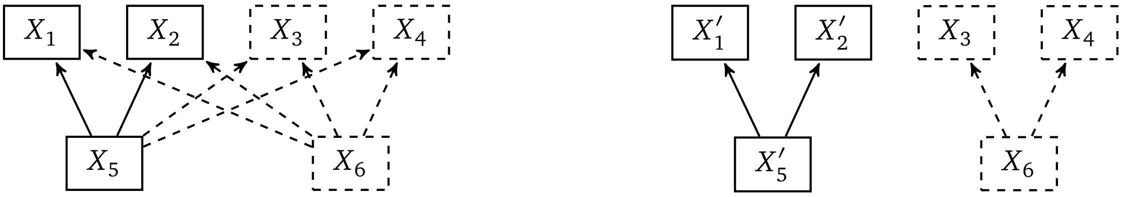


Figure 4.3: Ambiguous references between original and synthesized IDs modeled in a name graph (left) and the fixed version of the graph (right).

When NameFix is applied to the example, it detects the extended relation set for the original nodes X_1, X_2 and X_5 and therefore renames the identifiers to a new name X'. Like in the previous example for declaration conflicts, this doesn't ensure that all ambiguities are resolved, but separates them into purely original and purely synthesized ones as seen on the right side of Figure 4.3. As already explained for conflicts, NameFix can only fix hygiene-related ambiguities affecting original names. In this example, the original program was already ambiguous and the code added by the transformation was also ambiguous on its own. NameFix preserves these ambiguities while removing those which are hygiene-related.

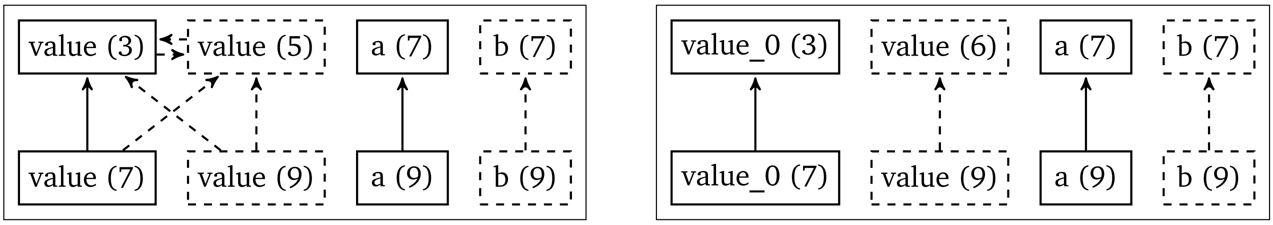


Figure 4.4: Name graphs for the example program from Figure 4.1 (left) and the fixed version (right).

While the presented examples were intended to demonstrate the capabilities of NameFix in large conflicts and overlapping ambiguities, the most common example can be assumed to be a conflicting definition between two names and a resulting ambiguity in the related references. The left side of Figure 4.4 shows the name graph for the program from Figure 4.1 that is an example for such a scenario. In the case presented here, NameFix detects the extended relations of the original names and renames them to new, fresh names based on their connection in the original name graph. The result can be seen on the right side, showing that both the conflict and the ambiguity have been resolved simultaneously.

5 Hygiene for modular programs

When looking at either of the previous definitions of name graphs or extended name graphs, it appears that they always consider programs as monolithic, isolated blocks of code: All identifiers in the program need to be available when the name graph is created, and they all are required to be renamable if NameFix decides to do so. There is no way to reference external names that are not part of the name graph themselves, and it is also not possible to ensure that renamings in the graph don't break references from external programs. In real scenarios however, programs make extensive use of libraries or other external resources that are shared globally and therefore can not be renamed – or even worse, they are already compiled and their internal implementation is not available at all. NameFix isn't capable to handle these issues and therefore, they all limit its applicability in practice.

To provide hygiene in such a modular context, it is necessary to adapt the formal models of name graphs and capture avoidance as well as the implementation of the NameFix algorithm: On the one hand, it should be possible to generate partial name graphs for each program module that only depend on a specified interface of other module's graphs. NameFix on the other hand should be able to propagate renamings applied to these interfaces and fix unintended inter-module references. Since it will be shown that global hygiene cannot be guaranteed for all possible situations, another desirable behaviour is the minimization of the global impact of renamings. These adaptations and ways to implement them are presented and discussed in the following sections.

5.1 Modular name resolution

Before an actual definition of modular name resolution can be developed, it is important to establish a precise definition of modularity itself: While this term is often used rather vaguely for a programming style that focusses a clear separation of different functionality, actual modularity needs to be provided by the used programming language and its compilation pipeline. Modularity is generally achieved through the division of a program into *modules* and a *separate compilation* of them. Before or during runtime, the compiled modules are then *linked* back into a complete program. While such an approach can increase the flexibility of a program as modules can be exchanged or updated even after compilation, well-defined interfaces are required that allow the modules to refer to each other. The point at which these interfaces are actually fixed can differ, depending on the type of linking applied to the modules: While *safe linking* requires an exact interfaces to be defined during compilation and matched when linking, *dynamic linking*, which is the most common form for languages compiled to bytecode like Java, allows the alteration of interfaces right up to the actual runtime of the program [5].

As program transformations can be applied at any point before or during compilation, it is difficult to determine how much information about the module dependencies and interfaces can be considered as already established. Considering name resolution, a fixed interface allows external references to point directly to their target identifiers, while in other cases, only a weakly defined, string-based name can be used as reference target. The latter approach is especially problematic when trying to establish hygiene, as it is not possible to detect unintended captures without knowing the actual targets of references. Therefore, all approaches in this section, although they are using Java as programming language, are based on the assumption that external references can be resolved to a well-defined interface with distinct identifiers.

For the name resolution to name graphs as established in the previous sections, there is no way to explicitly model external references at all. While it can be an option to simply add the external identifiers to the local name graph, this prevents a separate handling for both types of references. As already outlined, such a separation is important as external identifiers are not supposed to be renamed by NameFix. Therefore, it is unavoidable to extend the name graph model to support inter-modular references.

As name graphs are intended to be language-independent, they are not supposed to define the actual properties of the referenced module interfaces that need to be present during name resolution. Using only the abstract model of a *meta interface* containing all the relevant data, it is however still possible to formalize the concepts required for modular name resolution:

Definition 4. For a program p divided into a set of *modules* M, let there be the following for each module $m \in M$:

- Let there be a *module identifier* ID_m for m, which is unique in p.
- Let $dep : M \mapsto \mathscr{P}(\mathrm{ID})$ be a function that computes the module identifiers of the modules, m is depending on for name resolution.
- Let meta_m be the meta interface that is required to resolve modules depending on m.
- Let $id(\mathrm{meta}_\mathrm{m})$ extract ID_m from the meta interface of m.
- Let $export(\mathrm{meta}_\mathrm{m})$ be a function generating the set of exported IDs contained in the meta interface of the module m.

The module identifiers introduced in the above definition, which may but don't have to be modeled as identifier IDs, allow a module to be linked to its meta interface and its dependencies to be linked to matching interfaces. While it is possible to exchange a module by another one with the same module identifier and interface, only one of them is allowed to be present at the same time. Additionally, module identifiers are supposed to remain unchanged through program transformations as they allow the relation of the pre- and post-transformation version of the same module or interface. The actual name resolution for this modular context can be defined as following:

Definition 5. Let $m \in M$ be a module of the program p and $\mathrm{Meta}_\mathrm{dep} = \{\mathrm{meta}_\mathrm{d} \mid id(\mathrm{meta}_\mathrm{d}) \in dep(m)\}$ be the set of required meta interfaces to resolve m. Then, $resolve_{mod}$ is a function that resolves the *modular name graph* as well as the meta interface of the module m as $(G_m, \mathrm{meta}_\mathrm{m}) = resolve_{mod}(m, \mathrm{Meta}_\mathrm{dep})$. G_m is a tuple (V, ρ, ρ^{out}), where:

- V is the set of name IDs in m,
- $\rho \in V \mapsto \mathscr{P}(V)$ is a partial function from identfier IDs to their referenced internal identifier IDs,
- $\rho^{out} \in V \mapsto \mathscr{P}(\mathrm{Export}_\mathrm{dep})$ is a partial function from identifier IDs to their referenced external identifier IDs. The set of external identifier IDs that can be referenced, $\mathrm{Export}_\mathrm{d}\mathrm{ep}$, contains all exported IDs in the meta interfaces that m depends on: $\mathrm{Export}_\mathrm{dep} = \bigcup\{export(\mathrm{meta}) \mid \mathrm{meta} \in \mathrm{Meta}_\mathrm{dep}\}$
- meta_m is the meta interface of m as defined in Def. 4.
- and if $v_d \in \rho(v_r)$ or $v_d \in \rho^{out}(v_r)$, then the identifiers represented by v_r and v_d have equal string-wise namings.

In the presented definition, the meta interface of each module is the point where further modules are connected to it. The result can be a pipeline of modules, each using the previous modules' interface for their own resolution and to provide a new interface for further dependencies. A constraint that results from this is that programs are not allowed to have cyclic dependencies between their modules. Obviously, such a cycle would leave no starting point for name resolution, as each module needs another's name resolution to be finished before it can be resolved. If there are however no cycles, all modules of the program can be modeled as a directed acyclic dependency graph as seen in Figure 5.1.

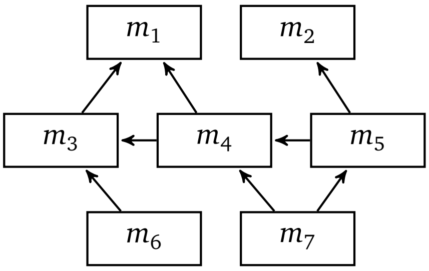


Figure 5.1: Example of an acyclic dependency graph for a program consisting of seven modules.

As the meta interface for each module is fixed once generated, it can be stored for further usage. Therefore, the order of evaluation for a module dependency graph is easy to derive, as the first module with no unresolved dependencies can be resolved next until all modules are evaluated. It is important that while indirect dependencies, like the one from module m_6 to m_1, determine the evaluation order of the graph, they don't imply that the meta interface of m_1 is used to resolve m_6. For m_6 to reference m_1, an additional dependency would need to be added to the graph, as in the case of m_7 and m_5.

As modular programs can grow if new modules are added, the generated meta interface for all modules needs to be stored, regardless if there currently are any references from other modules. In the example from Figure 5.1, this means that although m_7 doesn't have any modules depending on it, there might still be a new dependency added later that requires access to the stored interface.

A simplification that is used in the presented definition of name graphs is that relations between external IDs are not reflected in the model. While such relations obviously can exist, they are not relevant for the algorithms that are about to be presented, as external IDs are considered static and are never renamed. One specific consequence of the simplification is that conflicts between imported names can not be modeled, as this would be achieved by references to each other using the representation introduced in Chapter 4. As such a situation would however be the result of a conflict that can not be solved without modifying external identifiers, there is no possible solution without using more complex algorithms operating on a global level. The best option for a modular name resolution algorithm would therefore be to fail to allow the early detection of such issues.

```
package counter;

// No dependencies for this module

public class Counter {
    private int count;

    public int getCount() {
        return count;
    }
}
```

```
package longcounter;

import counter.Counter;

public class LongCounter extends Counter {
    public long count;

    public long addCount() {
        count += 1;
        return count;
    }

    public int getSuperCount() {
        return getCount();
    }
}
```

Figure 5.2: The example program from Figure 2.3, divided into two packages with an external reference from the right package to the left one.

Figure 5.2 shows a version of the original example program from Figure 2.3 that was used to demonstrate transformation hygiene. While in the original version, the two classes of the program were located in the same Java *package*, they are split into separate packages here. This means that the class *LongCounter* needs to import the class *Counter* before it can reference to its members, and it can only reference to members that are part of the public interface of the *counter* package.

In this and upcoming examples in this chapter, each Java package will be considered as an individual module. In this example, this means there are two modules *counter* and *longcounter* in the program. In the resulting module dependency graph, the module *longcounter* depends on *counter*, which does not have any further dependencies itself. The resolution order is therefore easily determined: The module *counter* needs to be resolved so that its interface can then be used to resolve *longcounter*.

Figure 5.3 shows the modular name graph of the module *longcounter* from Figure 5.2. It is assumed that the module *counter* was already resolved and only the publicly visible interface is accessible to the resolution of *longcounter* The external references to the interface are marked in blue and would be stored in the separate map ρ^{out} as part of the name graph of the module *longcounter*.

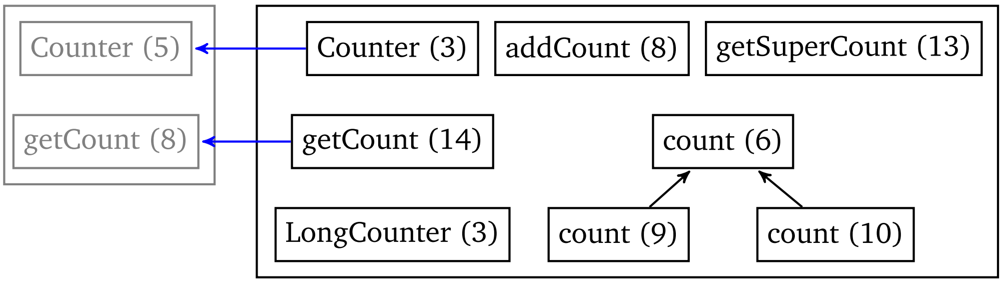


Figure 5.3: Modular name graph for the module *longcounter* from Figure 5.2, depending on the interface of the module *counter*. Inter-module references are marked in blue whereas interfaces that are not actually available as name graphs are marked gray.

The concepts of modular name graphs and their resolution as presented here provide the foundation for actual hygiene considerations. In the upcoming sections, issues and possible solutions for them will be presented and discussed.

5.2 Hygiene for cross-module references

To allow any hygiene considerations for modular name graphs, modules need to be persistently identifiable before and after program transformations. Only such a persistency allows the comparison of the pre- and post-transformation name graphs that is required to find unintended captures. New modules added by a transformation can however still be handled by considering all their nodes as synthesized. Since a modular handling of the program doesn't allow a global tracking of all identifiers, those moved between modules can only be considered as synthesized by the transformation in their new module's name graph.

To handle all modules of a program, the transformed dependency graph would need to be evaluated similarly to the name resolution as described in the previous section. While it is not an essential requirement that the dependency structure of the original modules remains completely unchanged, modified dependencies can complicate the evaluation order of the modules. An option to handle this is to first generate all name graphs of the pre-transformation modules that are supposed to be handled, store them in a cache, and use them when evaluating the post-transformation name graph. As a simplification, altered dependencies are not considered in the rest of this thesis, so that the pre- and post-transformation name graph can just be evaluated side by side.

```
$name\text{-}fix\text{-}modules(M_s, \mathrm{Meta}_s, M_t, \mathrm{Meta}_t) = \{$
  if $(M_t == \emptyset)$ return $\emptyset$;

  $m_t = m \in M_t : \forall \mathrm{ID}_{dep} \in dep(m) : \exists \mathrm{meta}_{dep} \in \mathrm{Meta}_t : id(\mathrm{meta}_{dep}) == \mathrm{ID}_{dep}$;
  $(G_s, \mathrm{meta}_s) = ((\emptyset, \emptyset, \emptyset), \emptyset)$;

  if $(\exists m \in M_s : \mathrm{ID}_m == \mathrm{ID}_{m_t})$ {
    $m_s = m \in M_s : \mathrm{ID}_m == \mathrm{ID}_{m_t}$;
    $(G_s, \mathrm{meta}_s) = resolve_{mod}(m_s, \mathrm{Meta}_s)$;
  }

  $(m_{fixed}, \mathrm{meta}_{fixed}) = name\text{-}fix\text{-}module(G_s, m_t, \mathrm{Meta}_t)$;
  return $m_{fixed} \cup name\text{-}fix\text{-}modules(M_s, \mathrm{Meta}_s \cup \{\mathrm{meta}_s\}, M_t \setminus \{m_t\}, \mathrm{Meta}_t \cup \{\mathrm{meta}_{fixed}\})$;
}
```

Figure 5.4: Definition of *name-fix-modules* that applies NameFix on a set of modules.

The function *name-fix-modules* defined in Figure 5.4 handles a set of modules and fixes them in an order based on their dependencies. While the parameters M_s and M_t are the sets of pre- and post-transformation modules that should be fixed, the sets $Meta_s$ and $Meta_t$ contain the required meta interfaces for modules that were already fixed or are not directly available to the algorithm. Already compiled modules that are not affected by the applied program transformation can simply have their original module interface added for both parameters.

From the set of transformed modules to be handled, one with no unresolved dependencies is selected and it is checked if there is a module with a matching ID in the set of original modules. If a match was found, the name graph of the original module is generated, while in the other cases, an empty original name graph is used to mark the module as fully synthetic. Finally, a yet to be defined function *name-fix-module* is called to actually fix the module. The function then calls itself recursively with the remaining set of modules to handle and the generated interfaces added to the according sets.

While the presented algorithm resolves names for the source programs dynamically, this is not an actual requirement for modular name fixing: The only essential information about the source program is its modular name graph and a way to relate this name graph to the matching transformed module. Such a relatability could for example also be provided by a simple map of module identifiers to source name graphs. As seen on the parameters used for the call of the *name-fix-module* function, the actual source module and its meta interface are not passed to the actual name fixing algorithms.

Considering the two-module example from Figure 5.3, *name-fix-modules* can be applied in many different use cases: If all the modules from the program are available in their original and transformed version, they can simply be used as input for the algorithm. If the module *counter* has already be compiled and only its interface is available, either its original and transformed interface can be used as parameters, or, if it was not altered by the transformation, the original interface can simply be used for both parameters. In contrast to global name resolution, the module could also be replaced by another module with the same ID and interface, as the actual implementation is not relevant for fixing the dependent module *longcounter*.

As an intermediate step, one could assume the not yet defined function *name-fix-module* to be equivalent to the definition of *name-fix* that was presented and extended in previous sections. The only necessary modification would be the replacement of the function *resolve* by $resolve_{mod}$, using the parameter $Meta_t$ to allow modular name resolution. This version would already be sufficient to allow for local references to be fixed as expected, while leaving external references completely unhandled as NameFix only considers the internal references in ρ.

Hygiene issues are however not limited to local references. Figure 5.5 shows an example of unintended references to the interface of a referenced module: As the IDs X_3 and X_4 are synthesized by the transformation, they are not allowed to point to the external ID X_1 from the original module's interface.

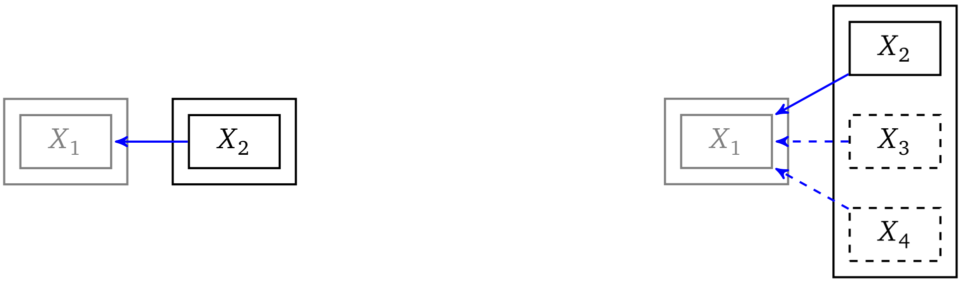


Figure 5.5: Original name graph (left) and transformed version with inter-module captures (right).

Modifying the function *name-fix-module* to support the fixing of these kinds of unintended captures is not possible using only the existing definitions of relations between IDs and capture avoidance. Instead, they need to be adapted to provide support for external references. In an extended definition of relatedness, IDs can also be related to ones in other modules' interfaces:

Definition 6. Let $G = (V, \rho, \rho^{out})$ be the modular name graph of a module m in a program p.

1. In G, two identifier IDs $v_1, v_2 \in (V \cup codom(\rho^{out}))$ are *related* if there is a path from v_1 to v_2 under the assumption that all edges $\in \rho$ and external edges $\in \rho^{out}$ are bidirectional.
2. The function $rel_G : V \mapsto \mathscr{P}(V \cup codom(\rho^{out}))$ maps each identifier ID v to the set of IDs related to v in G and the interfaces of other modules in p that G depends on.

Using this definition, the definition of capture avoidance only needs to be modified to also ensure that external names have no new relations added by the transformation.

Definition 7. A modular transformation $f : M \mapsto M'$ is *capture-avoiding* if for all source modules $s \in M$ with modular name graphs $G_s = (V_s, \rho_s, \rho_s^{out})$, and all target modules $t \in M'$ with name graphs $G_t = (V_t, \rho_t, \rho_t^{out})$: $\forall v \in (V_t \cup codom(\rho_t^{out})) : v \in (V_s \cup codom(\rho_s^{out})) \Rightarrow rel_{G_t}(v) \setminus rel_{G_s}(v) = \emptyset$.

The modifications required on NameFix to support modular capture avoidance are marginal. Figure 5.6 shows the definitions of the functions *find-relations*, *find-captures* and *comp-renaming* extended to support external references.

The modified version of *find-relations* shown in Figure 5.6 already covers most of the required modifications: When calculating the IDs that need to be renamed, the version of *find-relations* correctly detects external captures and *comp-renaming* finds all external relations that need to be preserved. This however causes a new problem once NameFix tries to rename the captured IDs: As the scope of the provided renaming function is limited to the current module, it is not possible to alter other modules and rename external IDs.

In many situations however, such a renaming isn't actually necessary to remove unintended captures. Considering the example from Figure 5.5 again, there are two possible ways to fix the graph: The option currently selected by NameFix would be to rename X_1 and X_2, which can not be applied in practice. The second option is to rename X_3 and X_4, which is possible without renaming any external references. However, selecting this option isn't as trivial as it may seem: As the transformed version of the module is not necessarily correct, there might be intended references between synthesized identifiers that are currently shadowed and only revealed in the fixed version of the graph.

```
Syntactic convention:
  $m^{@v} = x$   name $x$ of the identifier $v$ in module $m$ or the interface used by $m$ it was defined in.

$find\text{-}relations(n, (V, \rho, \rho^{out}), R) = \{$
  $R' = R \cup n$;
  if $(n \in dom(\rho))$ {
    foreach $v$ in $(\rho(n) \setminus R')$
      $R' = R' \cup find\text{-}relations(v, (V, \rho, \rho^{out}), R')$;
  }
  if $(n \in dom(\rho^{out}))$ {
    foreach $v$ in $(\rho^{out}(n) \setminus R')$
      $R' = R' \cup find\text{-}relations(v, (V, \rho, \rho^{out}), R')$;
  }
  foreach $v$ in $V \cup codom(\rho^{out})$ {
    if $(v \notin R' \wedge (v \in dom(\rho) \wedge n \in \rho(v)) \vee (v \in dom(\rho^{out}) \wedge n \in \rho^{out}(v)))$
      $R' = R' \cup find\text{-}relations(v, (V, \rho, \rho^{out}), R')$;
  }
  return $R'$;
}

$find\text{-}captures((V_s, \rho_s, \rho_s^{out}), (V_t, \rho_t, \rho_t^{out})) = \{$
  $\text{captureNodes} = \{v \mid v \in (V_t \cup codom(\rho_t^{out})), v \in (V_s \cup codom(\rho_s^{out})),$
      $find\text{-}relations(v, (V_t, \rho_t, \rho_t^{out})) \setminus find\text{-}relations(v, (V_s, \rho_s, \rho_s^{out})) \neq \emptyset\}$;
  return captureNodes;
}

$comp\text{-}renaming(G_s, (V_t, \rho_t, \rho_t^{out}), t, V_{rename}) = \{$
  $\pi = \emptyset$;
  foreach $v$ in $V_{rename}$ {
    if $(find\text{-}relations(v, (V_t, \rho_t, \rho_t^{out})) \cap \pi == \emptyset)$ {
      $\text{usedNames} = \{t^{@v_0} \mid v_0 \in (V_t \cup codom(\rho_t^{out}))\} \cup codom(\pi)$;
      $\text{fresh} = gensym(t^{@v}, \text{usedNames})$;
      $\text{relatedNames} = find\text{-}relations(v, G_s)$;
      $\pi = \pi \cup \{(v_0 \mapsto \text{fresh}) \mid v_0 \in \text{relatedNames}\}$;
    }
  }
  return $\pi$;
}
```

Figure 5.6: Altered version of the functions *find-relations*, *find-captures* and *comp-renaming* supporting external references. Changes to the version from Fig. 3.9 are marked green.

A solution to ensure that all intended references stay intact is to simulate a renaming of the external identifiers and resolve the name graph on the resulting,*virtual name graph*. Only if all captures are fixed in this simulated graph, all intended references are guaranteed to be restored and the actual renaming can take place.

Figure 5.7 shows an example of such a two-step fixing of the example module from Figure 5.5. In the first step, all external names are considered to be fully accessible and a simulated renaming is applied. As seen on the left side of the example, an intended reference between the IDs X_3 and X_4 is revealed after applying this renaming. As there are no more captures in the left graph, it can be assumed to be correct and the actual renaming can be applied. As X_3 and X_4 are supposed to be related, they need to be renamed to the same fresh name, which results in an equivalent corrected name graph as seen on the right side of the Figure.

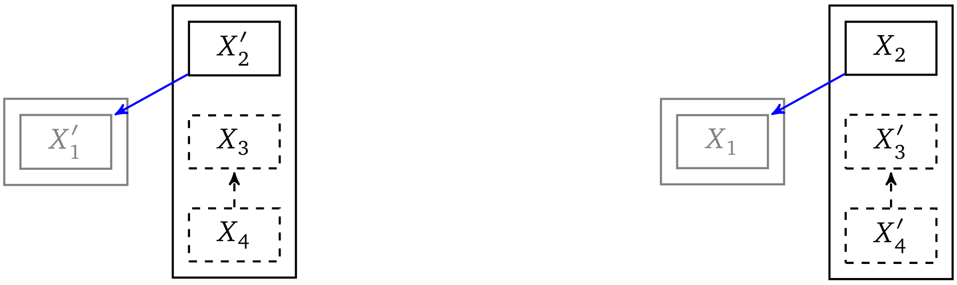


Figure 5.7: Fixing steps for the modular name graph from Fig. 5.5, with virtual fixed name graph (left) and final fixed name graph (right)

The current model of name resolution doesn't support a virtual name resolution as it is required to generate the intermediate, virtual name graph. So, an additional type of resolution needs to be defined.

Definition 8. Let p be a program divided into a set of modules M, and π a renaming map for identifiers exported by these modules. Let $m \in M$ be one of these modules with $\text{Meta}_{\text{dep}} = \{\text{meta} \mid id(\text{meta}) \in dep(m)\}$ being the required meta interfaces to resolve m. The function $resolve_{virtual}$ resolves the *virtual modular name graph* of m as $G_v = resolve_{virtual}(m, \text{Meta}_{dep}, \pi)$. G_v is equivalent to the name graph resolved by calling $resolve_{mod}(m, \text{Meta}'_{\text{dep}})$ with $\text{Meta}'_{\text{dep}}$ being the result of applying the renaming π on each exported identifier in Meta_{dep}.

The ability to resolve a virtual name graph for a module is a special feature that needs to be supported by the name resolution algorithm of the used programming language. However, it will be shown in Chapter 6 that its implementation can be very straight-forward when reusing the existing modular name resolution.

```
$name\text{-}fix\text{-}module(G_s, m_t, \text{Meta}_t) = \{$
  $G_{virtual} = name\text{-}fix\text{-}virtual(G_s, m_t, \text{Meta}_t, \emptyset);$
  return $apply\text{-}virtual\text{-}graph(m_t, \text{Meta}_t, G_{virtual});$
$\}$

$name\text{-}fix\text{-}virtual(G_s, m_t, \text{Meta}_t, \pi) = \{$
  $G_t = resolve_{virtual}(m_t, \text{Meta}_t, \pi);$
  $\text{capture} = find\text{-}captures(G_s, G_t);$
  if $(\text{capture} == \emptyset)$ return $G_t;$

  $\pi_{new} = comp\text{-}renaming(G_s, G_t, m_t, \text{capture});$
  $m'_t = rename(m_t, \pi_{new});$
  return $name\text{-}fix\text{-}virtual(G_s, m_t, \text{Meta}_t, \pi \cup \pi_{new});$
$\}$
```

Figure 5.8: Definition of *name-fix-module* using an added function *name-fix-virtual* for computing a capture-free virtual name graph for a module m_t.

Using the redefined functions *find-relations-recursive*, *find-captures* and *comp-renaming* from Figure 5.6 and the ability to resolve virtual name graphs, it is finally possible to find and fix all unintended captures in the modular name graph and return a virtual fixed name graph as on the left side of Figure 5.7. The resulting algorithm can be seen in Figure 5.8.

First looking a at the function *name-fix-virtual*, it seems very similar to the original definition of NameFix, although there is a fundamental difference between them: While the original algorithm can actually apply all the computed renamings on the program, *name-fix-virtual* can only apply the internal ones and simulate the results of the external ones using the virtual name resolution. While the resulting module is not practically usable as it depends on its virtual context, its name graph has the desired, capture-free structure that is supposed to match the final graph of the program returned by NameFix. As a consequence, the virtual name graph is returned after all captures are fixed, and will be used as a reference in the upcoming steps that are performed by the function *apply-virtual-graph*.

```
$select\text{-}renaming(\mathrm{rel}_1, \mathrm{rel}_2, \mathrm{fresh}, (V, \rho, \rho^{out}), \mathrm{meta}) = \{$
  if $(\mathrm{rel}_1 \cap codom(\rho^{out}) == \emptyset)$
    return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_1\}$;
  else if $(\mathrm{rel}_2 \cap codom(\rho^{out}) == \emptyset)$
    return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_2\}$;
  else fail ('Unable to fix module without renaming external identifiers!');
$\}$

$apply\text{-}virtual\text{-}graph(m, G_{virtual}) = \{$
  $((V_m, \rho_m, \rho_m^{out}), \mathrm{meta}_m) = resolve_{mod}(m, \mathrm{Meta}_m)$;
  $\pi = \emptyset$;

  foreach $v$ in $V_m$ {
    $\mathrm{usedNames} = \{m^{@v_0} \mid v_0 \in (V_m \cup \rho_m^{out})\} \cup codom(\pi)$;
    $\mathrm{fresh} = gensym(m^{@v}, \mathrm{usedNames})$;

    $\mathrm{rel}_m = find\text{-}relations(v, (V_m, \rho_m, \rho_m^{out}))$;
    $\mathrm{rel}_v = find\text{-}relations(v, G_{virtual})$;
    $\mathrm{rel}_\setminus = \bigcup\{find\text{-}relations(v_c, G_{virtual}) \mid v_c \in \mathrm{rel}_m \setminus \mathrm{rel}_v\}$;
    if $(\mathrm{rel}_\setminus \neq \emptyset \wedge (\mathrm{rel}_\setminus \cup \mathrm{rel}_v) \cap \pi_{alt} == \emptyset)$
      $\pi = \pi \cup select\text{-}renaming(\mathrm{rel}_v, \mathrm{rel}_\setminus, \mathrm{fresh}, G_{virtual}, \mathrm{meta}_m)$;
  }

  if $(\pi == \emptyset)$ return $(m, \mathrm{meta}_m)$;

  $m' = rename(m, \pi)$;
  return $apply\text{-}virtual\text{-}graph(m', \mathrm{Meta}_m, G_{virtual})$;
$\}$
```

Figure 5.9: Definition of *apply-virtual-graph* that calculates a renaming with minimal effect on exported names.

The definition of *apply-virtual-graph* is shown in Figure 5.9. The function takes the unfixed transformed module and the fixed virtual graph as parameters and computes the relations for each identifier in the unfixed module's name graph and the virtual graph.[1] For relations that were split through the fixing, either the remaining relation or all the split-off identifiers need to be renamed.

---

[1] As there can't be any references that don't include at least one local node, it is not necessary to explicitly handle referenced external identifiers

The function *select-renaming* selects one of these options: If one of the sets doesn't contain an external identifier, it is selected for renaming. However, if all sets contain external identifiers, there is no possible solution to fix the module using only local renamings and the algorithm therefore terminates with a failure message. To minimize the number of required renamings, the algorithm processes each set of names related in any of the graphs only once. However, similar to NameFix itself, it calls itself recursively until all relations are equal for the virtual and the actual name graph.

The definitions and algorithms presented in this subsection extend the capabilities of NameFix to ensure local and inter-modular capture avoidance if it can be achieved by only modifying the currently processed module. There is however one shortcoming that still remains and that will be addressed in the next subsection: The renaming of identifiers that are part of a module's exported interface can cause references from dependent modules to be altered. While a fixing of these modules can ensure that no unintended new references are added as a result, it doesn't restore any originally intended references broken by the renaming. To retain all intended references, NameFix needs to propagate renamings into depending modules.

5.3 Propagation of interface renamings

While the algorithms presented in the previous section ensure that only identifiers contained in the handled module's local name graph are renamed, they don't consider if any of the renamed identifiers is exported to other modules itself. While NameFix ensures that intendedly related local IDs are always renamed with each other to preserve the references between them, there is no way to determine the potential usages of exported IDs in other modules.

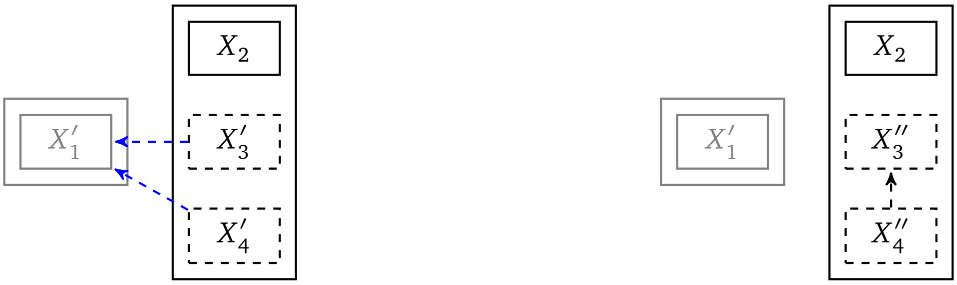


Figure 5.10: Modular name graph accessing a renamed interface (left), causing unintended references to be added and intended ones to be lost. Even after NameFix is applied, the name graph (right) doesn't reflect the intended name bindings.

Figure 5.10 shows an example similar to the one in Figure 5.5 from the previous section. This time however, the ID X_1 was renamed to X'_1 by NameFix when its module was handled. The resulting alteration of the module's interface can have two types of consequences that can be observed on the left side of the figure: The ID X_2 that was originally supposed to reference to X_1 has lost its reference through the renaming, while the IDs X'_3 and X'_4 coincidentally share the same name with the one selected when X_1 was renamed[2], and therefore reference to it.

Applying modular NameFix as previously presented to the module removes the added references from the IDs X'_3 and X'_4 as they are detected as unintended captures. The result can be seen in the resulting name graph on the right side of Figure 5.5, where the IDs were renamed to X''_3 and X''_4. Yet, the intended reference between X_2 and X'_1 is not restored as NameFix is only designed to remove or preserve existing edges. As the intention of NameFix is however to restore the originally intended name bindings of the program, this task needs to be added to NameFix' capabilities.

---

[2] This situation is possible as names selected by *gensym* are only guaranteed to be unique in the context of their own module and its dependencies.

However, an issue that needs to be considered first is the lack of information available to NameFix: When a module is handled, only the pre-transformation and the fixed and therefore already renamed meta interfaces of the referenced modules are available. Both don't allow any inference of intended references that were broken by renaming. This information can only be extracted from the unfixed transformed meta interface that is no longer available after a module is fixed.

To overcome this limitation, two possible solutions can be considered: One the one hand, the unfixed meta interface could be preserved even after NameFix is applied, while on the other hand, a reverse map of renamings could be generated, that allows the original namings to be restored. While the first option is likely easier to implement as no new data needs to be gathered, the second one could save the redundancy of storing information about identifiers not renamed by NameFix, which would usually be the majority. Either way, the additional data would need to be stored with the fixed module as part of its meta interface. Abstracting from the actual implementation, an language interface is required that allows NameFix to access the original naming of an identifier:

Definition 9. Let there be a module m with meta interface meta_m, that is dependent on a set of meta interface Meta_{dep}. Let $v \in \text{meta}_m$ be an identifier ID exported by the module.

1. Let $m_{original}$ be the original version of the module m before any renaming was applied. The function $\textit{original-name}(v, \text{meta}_m)$ extracts the original naming $x = m^{@v}_{original}$ from the interface of m.
2. The renaming $\pi_{reverse}(\text{Meta}_{dep})$ maps each ID $v \in \textit{exports}(\text{meta}_x)$ for any $\text{meta}_x \in \text{Meta}_{dep}$ to its original name as resolved by $\textit{original-name}(v, \text{meta})$.

The actual propagation of renamings needs to be performed after the current module was fixed as presented in the previous subsection, as only then, references are guaranteed to be intended. However, NameFix relies on the computed name graph to already contain all references that need to be preserved. To overcome this circular dependency of the name fixing steps, a virtual name graph based on the reverse renaming can be used as seen in Figure 5.11. The renaming $\pi_{reverse}$ is used as starting point when resolving the virtual name graph of the module. Consequently, the resulting name graph $G_{virtual}$ is not only capture-free but also contains all intended references, regardless if they would be lost in the actual name graph.

```
$\textit{name-fix-module}(G_s, m_t, \text{Meta}_t) = \{$
  $G_{virtual} = \textit{name-fix-virtual}(G_s, m_t, \text{Meta}_t, \pi_{reverse}(\text{Meta}_t));$
  return $\textit{apply-virtual-graph}(m_t, \text{Meta}_t, G_{virtual});$
}
```

Figure 5.11: Modified version of *name-fix-module* that undos renamings on interfaces when computing the fixed virtual name graph.

Similar to the previous subsection, with the correct structure of the name graph being known at this point, the one remaining task is to apply all the necessary renamings to the actual name graph of the module. While the function *apply-virtual-graph* is already sufficient to ensure that all unintended references are removed, it misses the ability to add relations between names. As both tasks are generally independent of each other, a new function can be added that only adds missing relations, as seen in Figure 5.12.

In its version from Figure 5.12, the function *apply-virtual-graph* is reduced to a wrapper that first calls a function *add-intended-relations* and then calls another function *remove-unintended-relations* on the resulting module. The function *remove-unintended-relations* is equal to the original definition of *apply-virtual-graph* in Figure 5.9 and removes relations from the module that are not in the correct virtual name graph. The newly added function *add-intended-relations* however renames identifiers to share the same name if they are supposed to reference each other according to the virtual graph.

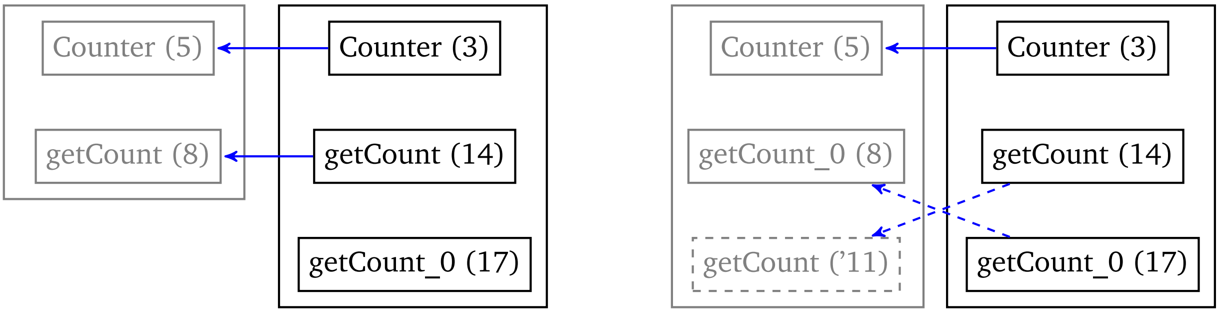
```
$\mathit{apply\text{-}virtual\text{-}graph}(m, \mathrm{Meta}_m, G_{virtual}) = \{$
  $m' = \mathit{add\text{-}intended\text{-}relations}(m, \mathrm{Meta}_m, G_{virtual});$
  return $\mathit{remove\text{-}unintended\text{-}relations}(m', \mathrm{Meta}_m, G_{virtual});$
$\}$

$\mathit{remove\text{-}unintended\text{-}relations}(m, G_{virtual}) = \{$
  // The previous definition of apply-virtual-graph as defined in Figure 5.9.
$\}$

$\mathit{add\text{-}intended\text{-}relations}(m, \mathrm{Meta}_t, (V_v, \rho_v, \rho_v^{out})) = \{$
  $((V_m, \rho_m, \rho_m^{out}), \mathrm{meta}_m) = \mathit{resolve}_{mod}(m, \mathrm{Meta}_m);$
  $\pi = \emptyset;$

  foreach $v$ in $V_m$ {
    $\mathrm{rel}_m = \mathit{find\text{-}relations}(v, (V_m, \rho_m, \rho_m^{out}));$
    $\mathrm{rel}_v = \mathit{find\text{-}relations}(v, (V_v, \rho_v, \rho_v^{out}));$

    if $((\mathrm{rel}_v \setminus \mathrm{rel}_m) \neq \emptyset \land \mathrm{rel}_v \cap \pi == \emptyset)$ {
      $\mathrm{propagatedNames} = \{m^{@v} \mid v \in (\mathrm{rel}_v \cap \mathit{codom}(\rho_v^{out}))\};$
      if $(|\mathrm{propagatedNames}| == 1)$ {
        $\mathrm{propagatedName} = \mathrm{name} \in \mathrm{propagatedNames};$
        $\pi = \pi \cup \{(v_0 \mapsto \mathrm{propagatedName}) \mid v_0 \in (\mathrm{rel}_v \setminus \mathit{codom}(\rho_v^{out}))\};$
      }
      else
        fail ('Unable to retain relations to external identifiers with different names!');
    }
  }

  return $\mathit{rename}(m, \pi);$
$\}$
```

Figure 5.12: Definition of a function *add-intended-relations* that adds intended references to a module, and modified definition of *apply-virtual-graph* that calls *add-intended-relations*.

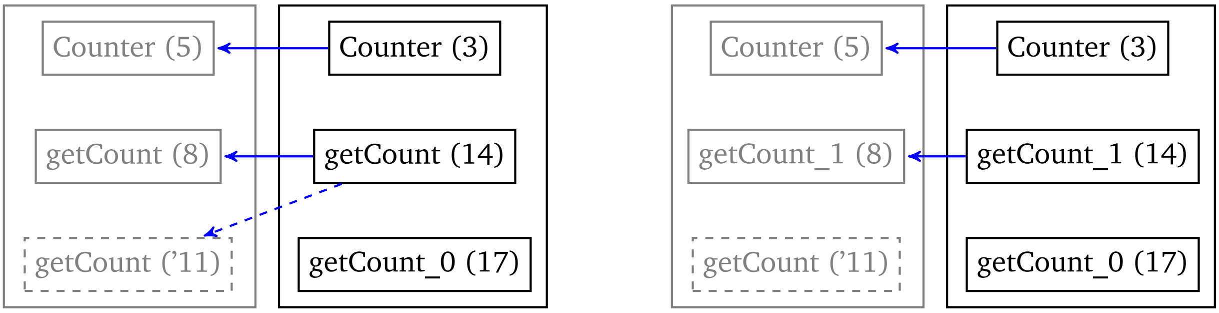
While, as already mentioned, the tasks of removing and adding references are generally independent, adding intended relations can cause new hygiene issues that need to be handled. The selected order allows this handling to take place solely as part of the function *remove-unintended-relations*, which is already implemented sufficiently.

For each local name in the name graph, *add-intended-relations* calculates the relations in the actual and the virtual name graph. Only if there are references that are missing in the actual name graph, the algorithm needs to take action. As such missing relations are always caused by a renaming of an external identifier, at least one external ID has to be in the set of related IDs for the virtual name graph. However, there is also the possibility that more than one external ID should be related. As the local identifier can only be renamed to one of the external names, it is impossible to solve such a situation if the external identifiers have different names. In this case, the algorithm terminates with a failure. If there is exactly one external name that the local IDs can be renamed to, this name is selected and all local IDs are scheduled for renaming.

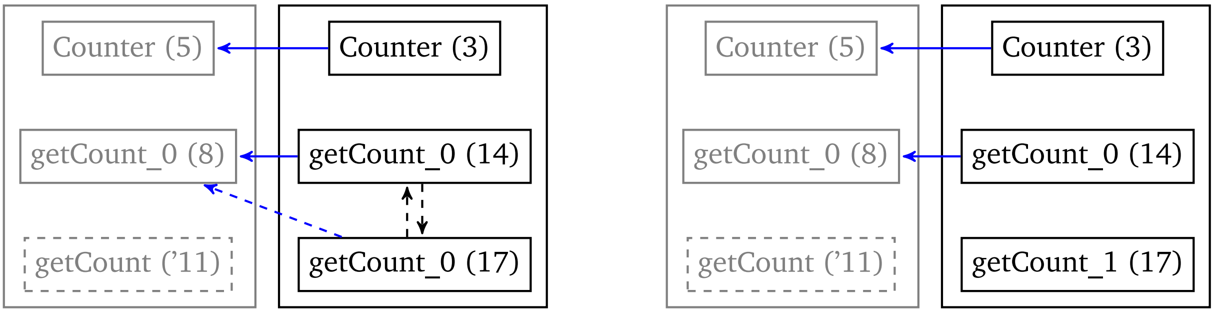
Compared to the removal of unintended relations, *add-intended-relations* doesn't require recursion as there is only one possible solution per identifier that is fully defined through the current naming of the used interface and the virtual name graph. There can however be new hygiene issues resulting from the applied renamings. These are handled in the next step as part of *remove-unintended-relations*, which tries to solve them or terminates with a failure if there is no possible solution.


Counter (5)
Counter (3)
getCount (8)
getCount (14)
getCount_0 (17)
Counter (5)
Counter (3)
getCount_0 (8)
getCount (14)
getCount ('11)
getCount_0 (17)

(a) Original name graph **(b)** Transformed name graph with renamed interface


Counter (5)
Counter (3)
getCount (8)
getCount (14)
getCount ('11)
getCount_0 (17)
Counter (5)
Counter (3)
getCount_1 (8)
getCount_1 (14)
getCount ('11)
getCount_0 (17)

(c) Unfixed virtual graph with original interface **(d)** Fixed virtual graph with original interface


Counter (5)
Counter (3)
getCount_0 (8)
getCount_0 (14)
getCount ('11)
getCount_0 (17)
Counter (5)
Counter (3)
getCount_0 (8)
getCount_0 (14)
getCount ('11)
getCount_1 (17)

(e) Transformed name graph after intended references are restored **(f)** Transformed name graph after unintended references are removed

Figure 5.13: Modular name graph with inter-module capture and renamed interface, and fixing steps using modular NameFix as presented.

Figure 5.13 shows a step-by-step example with all intermediate steps of virtual and actual name fixing. The first two name graphs 5.13a and 5.13b show the original and transformed name graphs of a Java program similar to the example presented in Figure 5.2. In this example, the only action performed by the transformation itself was to add a method named *getCount* to the module whose interface is seen on the left side of the graph. To solve the naming conflict with the existing method of the same name, NameFix renamed the original method *getCount (8)* to *getCount_0*. This however causes a name resolution result as seen in 5.13b, where the original call of *getCount (8)* now points to the synthesized definition of *getCount (11)*, and the identifier *getCount_0* is now bound to the renamed *getCount_0 (8)*.

In the first step to fix this situation, NameFix resolves the virtual name graph as seen in 5.13c to restore the original interface of the referenced module by using the reverse renaming map. This already fixes the unintended reference from *getCount_0 (17)*, but also causes an ambiguity for *getCount (14)*. This name graph is then fixed assuming that all identifiers can be renamed without any constraints. The resulting virtual name graph as seen in 5.13d is the structurally correct name graph with all intended references restored and unintended captures removed.

Resuming work on the actual transformed name graph, 5.13e shows the result after the intended reference to *getCount_0 (8)* is restored. As can be seen, this also caused additional unintended references between the local names. Finally, all unintended references are removed by renaming *getCount_0 (17)* to *getCount_1* as seen in 5.13f. As this graph is structurally equal to the fixed virtual name graph, it is guaranteed that capture avoidance of the applied transformation is preserved.

5.4 Avoidance of renaming exported identifiers

The solutions presented in the previous subsection are able to handle the most common situations that can occur when exported identifiers are renamed. Yet, there are situations that cannot be solved without a global algorithm that can retroactively modify already fixed modules. Examples for such situations are the following:

- If an identifier or a set of related identifiers is supposed to reference multiple external identifiers that originally shared the same name, but that were renamed to different new names, it is not possible to restore all intended references.
- If an external identifier is renamed to a name that also exists in the local module, new, possibly unfixable hygiene issues may be caused if local identifiers are renamed to restore references.
- If two exported identifiers from different modules share the same name after renaming, this might cause a conflict between them that cannot be represented in the modular name graph and therefore leads to an immediate failure of the name resolution algorithm.

Even if NameFix can handle the renaming by using the presented algorithms, the renaming of exported identifiers may can cause additional problems: If two modules are supposed to be interchangable as they share the same interface and module identifier, renaming their interfaces differently can break this interchangability. Even propagating the renaming to dependent modules can only restore compatibility to one of the modules. For this problem, there is no real solution except manual adaptions to the interfaces or the postponement of name fixing and renaming propagation up to the point when the actual module dependencies are known[3]. Additionally, the renaming of exported identifiers always requires NameFix or a compatible algorithm to be available and applied to all dependent modules, which may not be an acceptable requirement in many practical scenarios.

While for all these reasons, the renaming of exported names can be problematic, it often can be avoided in the first place. Thankfully, with knowledge of the correct name graph, there are often multiple renaming options to choose from, which were already used when finding a method of fixing inter-modular captures. The method *apply-virtual-graph*, later renamed to *remove-unintended-relations*, was introduced to select the most suitable option. In the sub-method *select-renaming*, it was checked if one of the sets to rename doesn't contain an external identifier, and the matching set was selected. Yet for local conflicts, both sets are suitable, and the decision for one of them is arbitrary. Here lies the potential to add a check for exported identifiers and select the set containing none or at least the lowest amount of them.

---

[3] While this might be a acceptable approach for transformations that are part of the compilation process, it is not applicable for transformations like Refactorings that are applied completely independent of module resolution.

```
$\textit{select-renaming}(\mathrm{rel}_1, \mathrm{rel}_2, \mathrm{fresh}, (V, \rho, \rho^{out}), \mathrm{meta}) = \{$
  if $((\mathrm{rel}_1 \cup \mathrm{rel}_2) \cap \textit{codom}(\rho^{out}) == \emptyset)$ {
    if $(|\mathrm{rel}_1 \cap \textit{export}(\mathrm{meta})| > |\mathrm{rel}_2 \cap \textit{export}(\mathrm{meta})|)$
      return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_2\}$;
    else
      return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_1\}$;
  }
  else {
    if $(\mathrm{rel}_1 \cap \textit{codom}(\rho^{out}) == \emptyset)$
      return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_1\}$;
    else if $(\mathrm{rel}_2 \cap \textit{codom}(\rho^{out}) == \emptyset)$
      return $\{(v \mapsto \mathrm{fresh}) \mid v \in \mathrm{rel}_2\}$;
    else
      fail ('Unable to fix module without renaming external identifiers!');
  }
}
```

Figure 5.14: Extended definition of *select-renaming* that minizes renamings of exported identifiers.

In Figure 5.14, an extended definition of the function *select-renaming* is presented that explicitly handles the case that both sets of possible renamings that were determined by the function *remove-unintended-relations* don't contain any names from external interfaces. In this case, the one with the lowest number of exported names is selected to minimize the impact of the renaming.

While the presented code only attempts to minimize the impact of a renaming as a soft constraint, there might be situations that require the avoidance of interface renamings as a hard constraint that has to be satisfied. The simplest way to handle such an use case is a check of the final result of name fixing for renamed interfaces. As NameFix already minimized the number of required renamings, any remaining ones would be unavoidable and therefore require the algorithm to terminate with a failure. Of course, there could also be more complex real scenarios with different priorities when avoiding renamings. There could be integrated individually for each use case by directly modifying the criteria added in Figure 5.14.

In this chapter, the concepts of modular name resolution, capture avoidance and name fixing were introduced and discussed. Algorithms were presented that allow the application of the concepts defined for the original NameFix algorithm to a modular context and with limited control over external interfaces. Combined, they provide a strong foundation that can handle a wide range of module systems and name binding rules used in programming languages. This greatly extends the practical usability of NameFix, as will be demonstrated in the next chapter on the example of Lightweight Java.

6 Case study: Lightweight Java

To create test cases and scenarios for the algorithms and definitions presented in this thesis, they need to be applied on actual program transformations. One of the intentions of extending NameFix' capabilities was its integration into the *SugarJ* library, that allows code with an alternative syntax to be embedded into regular Java code [6]. Using a subset of Java is an ideal way for a first review of the applicability of the defined concepts to Java, and to compare their capabilities with the original definition of NameFix.

Lightweight Java (LJ) is a subset of Java defined 2010 by Rok Strniša as part of the development of an alternative module system for Java [7]. As LJ aims to reproduce a significant part of the Java language while using a substantially simplified syntax, it is ideal for studying program transformations and hygiene on a simple but realistic example. Additionally, the name lookup rules and type system for LJ are formally specified in great detail and proven to be sound, which makes it easy to create parsing, type checking and name resolution algorithms.

Scala was selected as host language for implementing LJ and NameFix as its advanced language features like *case classes* and *pattern matching* allow a fast implementation of syntax-based transformations. Additionally, an existing Scala implementation of the original NameFix algorithm by its authors could be used as a basis for all subsequent extensions.[1]

6.1 Implementation of Lightweight Java

The first step to use NameFix on Lightweight Java was the implementation of the language's *Abstract Syntax Tree (AST)*. For this task, the precise and well-documented definition of LJs grammar was of great support. Additionally, the usage of Scala's case classes to construct and pattern matching to process AST nodes provided an intuitive and effective way to work with LJ on a syntactical level. To allow an effective usage of functional programming patterns, all AST nodes were designed as immutable, meaning that they can't be modified after their instantiation. Instead, a fresh node needs to be created with the modified values.

Figure 6.1 shows excepts of the definitions of the AST nodes *Program* and *ClassDefinition* that demonstrate how type checking for LJ can be implemented in Scala. The parameters of the case classes, which both extend an abstract superclass *AST*, are equivalent to definitions of read-only class fields with the same name, while additional features, like the comparison of two objects based on their field's values, are automatically added to the definition. Both class definitions also make use of Scala's *star parameters*, which allow a variable number of parameters, meaning that an LJ program can contain a variable number of classes and an LJ class can contain an variable number of elements – which themselves can be fields or methods.

The definition of the method *findAllFields* shows how intuitive working on AST nodes in Scala can be: To get a set of all fields declared or inherited by a class, its inheritance path is calculated by another method *getInheritancePath* and the fields of all classes in the path are collected using the *flatMap*-method. The actual type checking is implemented using the predefined Scala method *require*, which checks a certain condition and raises an *IllegalArgumentException* if it is not fulfilled at runtime. For the *Program* class, the only condition that needs to be checked is that all class names are unique, which can be achieved by extracting the names, removing duplicates and comparing the resulting list's size with the number of classes defined in the program. The type checking is then continued by calling the classes' type checker for each class of the program.

---

[1] The original Scala implementation of NameFix by Sebastian Erdweg can be found in a GitHub repository at the URL `https://github.com/seba--/hygienic-transformations`. A fork of this repository, containing the new implementation of Lightweight Java and the extended versions of NameFix, can be found at the URL `https://github.com/nritschel/hygienic-transformations`.

```
case class Program(classes: ClassDefinition*) extends AST {
  def findAllFields ( classDefinition : ClassDefinition ) =
    getInheritancePath( classDefinition ).flatMap(_. fields )

  def typeCheck = {
    require(_.className.name).distinct.size == classes.size)
    classes .foreach(_.typeCheckForProgram(this))
  }
}

case class ClassDefinition (className: ClassName, superClass: ClassRef,
        elements: ClassElement*) extends AST {

  def typeCheck(program : Program) = {
    val allFields = program.findAllFields(this)

    require( allFields .map(_.fieldType). forall {
      case className@ClassName(_) =>
        program.findClassDefinition(className).isDefined
      case ObjectClass => true
    })
    ...
    getMethods(program).forach(_.typeCheck(program, this))
  }
}
```

Figure 6.1: Excerpts of the definitions of the AST nodes *Program* and *ClassDefinition*

Type checking a *ClassDefinition* consists of a larger number of checks, of which only one typical example is seen in Figure 6.1: The check if there is a definition for each type referenced by the fields of the current class. The check calls the already explained method *findAllFields* to get a set of all fields available in the current class. The types of these fields are extracted and pattern matching is used to determine their kind in the AST.

Per definition of Lightweight Java, a type, represented in the AST as a *ClassRef* node, can either be a class name or a reference to the *Object* class, which, contrary to Java, has the semantics of an empty class definition. While for references to the *Object* class, no resolution is required, there needs to be a matching definition for each referenced class name, which is looked up by calling a helper function *findClassDefinition*. Like for *Program* nodes, type checking is then continued recursively for all methods defined by the class.

The existing implementation of NameFix already provided definitions for name graphs and names. While the first one was sufficient as a starting point for the modifications presented in the upcoming subsections, the latter one has been replaced by a new class that is closer to the theoretical definitions: A class *Identifier* is now used to represent a specific identifier used in a program. Identifiers have a string-based name property and are compared based on an internal ID, which is retained if an identifier is renamed. Consequently, they can be used in name graphs as an equivalent to the theoretical name IDs.

To allow an easy and uncomplicated resolution of name graphs for LJ, the *Identifier* class is also used as a representation of identifier naming in the LJ AST. Identifiers with additional features, like the

ClassName class shown in 6.1, are extending the *Identifier* class as well as additional *traits*[2] like *ClassRef* for class references. To simplify the creation of ASTs, the Scala feature of *implicit conversions* was also applied to implicitly transform strings to fresh identifiers where applicable.

A parser for LJ was also created to simplify the writing of test programs. It makes use of Scala's *parser combinators*, which allow an efficient implementation that abstracts from the actual parsing algorithms. Figure 6.2 shows an example of a LJ program and the parsed AST. In this example, the *Identifier* class as well as the derived *ClassName* and *VariableName* classes are explicitly shown, while they could also be implicitly converted from a simple string. A detail that is equivalent to the original LJ definition, but may be confusing at first, is the fact that the returned value is the child of the *MethodBody*-node, followed by the actual statements before the value is returned.

```
class Example {
  Example field;

  Object method(Example param)
  {
    param = this.field;
    param.field = this;
    return param;
  }
}
```

```
Program(ClassDefinition(
  ClassName(’Example’), ObjectClass,
  FieldDeclaration(
    ClassName(’Example’), Identifier(’Field’)),
  MethodDefinition(
    MethodSignature(ObjectClass, Identifier(’method’),
      VariableDeclaration(
        ClassName(’Example’), Identifier(’param’))),
    MethodBody(VariableName(’param’),
      FieldRead(VariableName(’param’),
        This, Identifier(’field’)),
      FieldWrite(VariableName(’param’),
        Identifier(’field’), This)))))
```

Figure 6.2: Example of a Lightweight Java program and the parsed AST.

Using the presented implementation of Lightweight Java, it was possible to create a name resolution algorithm and adapt NameFix to support hygienic transformations of Lightweight Java programs. The resulting implementations will be shown and discussed in the upcoming subsections.

6.2 Hygienic transformations for Lightweight Java

The first task to enable hygiene for Lightweight Java transformations was to implement the interfaces required by NameFix. The interface functions required by NameFix are defined in a trait *Nominal* that needs to be implemented for all AST nodes:

- The method *allNames*, supposed to return a set of all names used in the AST, was trivial to implement by simply traversing all nodes of the AST and collecting all used names in its identifiers.
- The implementation of the method *rename* to apply renamings was also based on traversing the AST. But instead of collecting the names of the identifiers, they were renamed based on the given renaming function. As all nodes of the AST are designed to be immutable, a fresh AST is built using the new identifiers, and finally returned by the *rename*-method.
- The most complex method in the interface was the actual name graph resolution algorithm *resolveNames*, which will be discussed in detail.

---

[2] Traits in Scala are a similar concept to *interfaces* in Java, declaring an abstract set of class functionality, that needs to be implemented by classes extending the trait

Generating the name graph for an AST works similar to the implementation of *allNames*: The AST is traversed and each node's name resolution method is called. Calculating the references between IDs is however a less simple task: As it is not possible for identifier nodes to know the context it is used in, a scoping map needs to be provided to find the correct bindings of references.

In Lightweight Java, there are two rather separated scoping levels: On the one hand, there is the scoping for class names and the fields and methods of each class. On the other hand, there are local variable bindings inside each method. Yet, type information is also relevant for local variables as the bindings of field and method usages depend on it.

Since a non-modular name resolution with bindings for class names only makes sense when it is performed for a whole program, the generation of a global scoping map is only performed in the *resolveNames*-implementation of the node *Program*, which is the AST's root node. Here, the identifiers in the definitions of all classes and their elements are collected and a name-to-identifier mapping is created for them. To also allow the resolution the name graphs of invalid programs as presented in Chapter 4, names need to be mapped to not just one identifier but a set of all conflicting definitions with the same name. The generated environment is then passed down when traversing through the lower level AST nodes.

ClassDefinition-nodes can simply pass the already calculated environment to their childs and collect the resulting name graph nodes and edges. For local name bindings in methods however, an additional map needs to be generated that maps variable names to their definitions as method parameters. If method parameters share the same names and are therefore in a declaration conflict, additional references between them are also added to the name graph.

While the original definition of LJ doesn't allow local variable declarations, this feature was added as an example of a program transformation. To support a binding to local variables, the mapping needs to be updated after every statement of the method is processed, as additional declarations may have been added. This also allows an on-the-fly handling of illegal redefinitions of variables: If the name resolution of a declarative statement finds an already defined mapping for its declared name, a reference to the existing binding is added and the original mapping is overwritten. This way, all conflicting variable declarations and their references are related to each other in the resulting name graph.

The resolution of variables to their type is already defined as a part of type checking which can be reused: It simply maps each variable declaration node to the class name node of its declared type. Using the global mapping for classes, the relevant fields and methods can be looked up by first using this local variable-to-type-map and then the global class-name-to-elements map. A special handling is required for the identifier *this*, that needs to be added as a declared variable with the same type as the class containing the currently resolved method. The *this*-identifier and all references to it are however not added to the name graph themselves, as *this* is a keyword and therefore can't be considered as a regular identifier.

Using the presented environments, it is possible to create an extended name graph for the AST of any syntactically legal LJ program, independent if it is correctly typed or has semantically allowed name bindings. A non-extended name resolution was not implemented as the only difference to the extended resolution would be a failure of resolution for conflicting definitions. It is however possible to convert an extended name graph to a non-extended one, which fails if identifiers have multiple outgoing edges.

As not all Lightweight Java programs contain multiple outgoing edges for identifiers or transitive name bindings, it is possible to apply the original NameFix algorithm as implemented by its authors to many LJ programs. Yet, to handle the full capabilities of LJ, especially the method overriding feature, and fix name graphs with declaration conflicts, the extended version of NameFix as presented in Chapter 3 is required. The necessary modifications for this version are implemented in Scala as *NameFixExtended* almost identical to the theoretical definitions.

To allow the new version of NameFix to be tested on actual program transformations, two example transformations are implemented, which add some Java features not originally defined as part of Lightweight Java:

- Lightweight Java only allows variables to be declared as method parameters. To remove this restriction, a new AST node for local declarations was added and a transformation was implemented to convert programs with local declarations to regular LJ programs. To achieve this, the transformation adds a new helper method with the suffix "_ldt" to the class, which has the locally declared variables added as additional parameters. The original method body is then replaced by a call of the helper method, passing its original parameters and using the value *null* for the added ones.
- Loops are also not a supported feature of Lightweight Java. As recursion is however possible, loops can be replaced by calls to a recursive helper method and the addition of some helper fields to save and restore state over each recursion step. As for local declarations, an AST node is added that then gets replaced by original LJ elements using a program transformation. As it needs to support nested loops and handle all changes to state correctly, the actual transformation is too complex to be described in detail here. An important detail is however that the transformation needs to keep an internal counter for added fields and methods, as collisions between names added by the same transformation cannot be fixed by NameFix.

For both implemented transformations, hygiene issues can occur if existing methods or fields with the same names as the added helper elements exist. Figure 6.3 shows an example of such a hygiene issue resulting from the local declaration transformation. While there should be a bottom-up chain of method calls in the program on the right, the method name *method_ldt* is declared twice and the resulting program is therefore invalid.

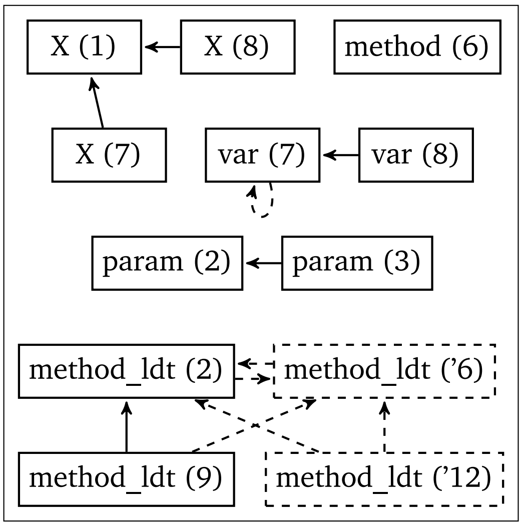
```
class X {
  Object method_ldt(Object param) {
    return param;
  }

  Object method() {
    X var;
    var = new X();
    return this.method_ldt(var);
  }
}
class
```

```
class X {
  Object method_ldt(Object param) {
    return param;
  }

  Object method_ldt(X var) {
    var = new X();
    return this.method_ldt(var);
  }

  Object method() {
    return this.method_ldt(null);
  }
}
```

Figure 6.3: Example Lightweight Java program with local variable declaration (left) and result of the unhygienic program transformation to regular LJ code (right).

The left side of Figure 6.4 shows the resulting name graph of the transformed program. As already explained in Chapter 2, it is important that original identifiers that are moved – or duplicated as the variable name *var* – keep their original ID. As a result, their connections to other original names are not seen as unintended captures. The actual captures are however seen at the bottom of the graph, as the conflicting method definitions are referencing each other and the references to them are ambiguous.

To fix the program, NameFix renames the original method and the intended reference to it, which separates the two method definitions and their references. The resulting Lightweight Java program can



```
class X {
  Object method_ldt_0(Object param) {
    return param;
  }

  Object method_ldt(X var) {
    var = new X();
    return this.method_ldt_0(var);
  }

  Object method() {
    return this.method_ldt(null);
  }
}
```

Figure 6.4: Name graph of the unhygiencially transformed program from Fig. 6.3, and the resulting Lightweight Java program after NameFix has been applied.

be seen on the right side of Figure 6.4. Similar examples of fixing for the transformation of loops can be found as tests in the actual Scala source code.

6.3 Modular hygiene for Lightweight Java

While the existing implementation of NameFix could be reused to enable hygiene for Lightweight Java, it did not contain any definition of modular interfaces or name resolution. To model the new requirements to the programming language's interface, the existing trait *Nominal* is extended by a new trait *NominalModular*, adding new interface methods. Additionally, a trait *MetaInterface* was defined to model the required meta interface to apply modular name resolution for a module. As the modular name resolution of each programming language is allowed to define its own meta interfaces, generic types are used to allow such a binding of a module type to a meta interface type. Figure 6.5 shows the Scala definitions for both traits. The intended semantics match the definitions from Chapter 5.

```
trait MetaInterface {
  val moduleID: Identifier
  def export: Set[ Identifier ]
}

trait NominalModular[T <: MetaInterface] extends Nominal {
  val moduleID: Identifier
  def dependencies: Set[Name]
  override def rename(renaming: Renaming): NominalModular[T]

  def resolveNamesModular(metaDependencies: Set[T]): (NameGraphModular, T)
  def resolveNamesVirtual(metaDependencies: Set[T], renaming: Renaming):
    NameGraphModular
}
```

Figure 6.5: Definitions of the traits *MetaInterface* and *NominalModular* in Scala.

The implementation of the modular NameFix algorithms in their final definition as presented in Chapter 5 is straight-forward. To give an example, Figure 6.6 shows the implementation of the function *add-intended-relations* from modular NameFix as a Scala method. It only differs from the pseudo-code definition in Figure 5.12 by the usage of Scala's methods to access and manipulate sets and maps. If the algorithm fails, an *IllegalArgumentException*, which also contains an error message in the actual code, is raised, as this allows test cases to explicitly check for this behavior.

```scala
def addIntendedRelations[S <: MetaInterface, T <: NominalModular[S]]
        (m: T, metaDep: Set[S], gVirtual: NameGraphModular) = {

  val (gM, _) = m.resolveNamesModular(metaDep)
  var renaming: Map[Identifier, Name] = Map()

  for (v <- gM.V) {
    val relM = findRelations(v, gM)
    val relV = findRelations(v, gVirtual)

    if ((relV -- relM).nonEmpty && relV.intersect(renaming.keySet).isEmpty) {
      val externalNames = gVirtual.EOut.values.flatten.toSet
      val propagatedNames = relV.intersect(externalNames).map(_.name)
      if (propagatedNames.size == 1) {
        val propagatedName = propagatedNames.head
        renaming ++= (relV -- externalNames).map(r => (r -> propagatedName))
      }
      else
        throw new IllegalArgumentException()
    }
  }
  m.rename(renaming).asInstanceOf[T]
}
```

Figure 6.6: Scala implementation of the function *add-intended-relations* as originally defined in Figure 5.12.

Before actually implementing the required modular name resolution methods for Lightweight Java, a concept for modules needs to be defined for it in general. While the original dissertation about Lightweight Java presented a concept for a module system, called *Lightweight Java Module System (LJAM)*, this system differs greatly from the concepts used in regular Java code. Therefore, implementing it would not allow a meaningful comparison with real Java behavior. The actual Java system of packages however is too complex to be actually implemented in the context of this thesis. A simplified but sufficient attempt to emulate modularity in name resolution is to resolve names not on program level using a global name-environment, but separately for each defined class. This way, access modifiers as in Java can be used to compute the exported members of a class, while references to external class names are sufficient to infer a dependence on this classes exported members.

While Lightweight Java doesn't natively support access modifiers, their addition to the AST is trivial: A flag indicating the used modifier can be added to the relevant AST nodes, namely the signatures of method definitions and field declarations. For class-based modularity, the relevant Java access modifiers are *public* and *private*. The *public* modifier indicates that a class element is part of a classes interface, which is semantically equal to Java [2]. While the *protected* modifier from Java is not implemented, package internal members can be treated as public because no separation of packages is implemented for LJ. This also means that the *public* modifier can be selected as the default modifier by the parser if no explicit choice is made.

As the *MetaInterface* class is only an abstract interface, an actual implementation is required for LJ. While the *moduleID* field can simply be filled with the corresponding class name, the set of exported identifiers returned by the method *export* needs to be divided into exported fields and exported methods internally to allow the correct resolution of all references. As LJ doesn't support method overloading, the number and types of parameters for each method are not relevant for name resolution. The resulting implementation of the class *ClassInterface* can be seen in Figure 6.7.

```scala
case class ClassInterface(className: ClassName, exportedFields: Set[Identifier],
    exportedMethods: Set[Identifier]) extends MetaInterface {

  override val moduleID = className

  override def export: Set[Identifier] = exportedFields ++ exportedMethods
}
```

Figure 6.7: Definition of *ClassInterface* in Scala, representing the meta data for name resolution generated for each Lightweight Java class.

Using the previous definitions, the final step of implementing modular name resolution is the implementation of the interface methods defined in the *NominalModular* trait:

- The *moduleID* field is set to the class name identifier, which also makes it compatible with the ID used in the *ClassInterface* implementation.
- As already mentioned, the set of *dependencies* for a class can be generated by collecting all references to external class names in the AST of the class.
- The function *rename* doesn't require a new implementation, as the only difference to the interface of *Nominal* is the requirement to return an instance of *NominalModular*. As *rename* always returns a renamed version of the current class definition, which is now a member of *NominalModular*, this requirement is automatically satisfied.
- To avoid a duplication of original name resolution code for the method *resolveNamesModular*, the resolution can be divided into three steps: First, the interfaces given as parameters are used to build an environment mapping that contains the classes and their elements. Then, the non-modular name resolution method *resolveNames* can be called to generate a non-modular name graph using this environment. Finally, the references in this name graph can be separated into internal and external references by checking if the referenced identifiers are declared in the current AST. This results in a modular name graph which can be used by NameFix.
- The method *resolveNamesVirtual*, which may seem to be the most difficult to implement, can take a great benefit from the already explained implementations: It can simply create a wrapper around the given class interfaces that applies the given renaming if it is defined for an identifier. Then, it can reuse the method *resolveNamesModular* by calling it with these interfaces.

Altogether, the implementation of the required interfaces to add modular name resolution support for Lightweight Java is straight-forward. This shows that the requirements to apply modular name fixing to an actual programming language are still modest and don't require a great effort if the general structures for non-modular name resolution are already present. Advanced language features like overloading can of course complicate the resolution process as type information may be required for a sufficient resolution. Yet, such information is also required to allow the actual execution of a module, and already available structures to store the meta data can likely be re-used for the presented tasks.

7 Related Work

Many of the oldest and most significant approaches to hygiene were developed in the context of syntax macro expansion [3, 8]. Especially the work by Herman [8] provided a formal foundation that was adapted for the original NameFix algorithm by its authors. However, algorithms that ensure hygiene for syntax macros have two significant advantages: On the one hand, they can interact directly with the macro engine and rename identifiers defined by the macro on-the-fly during expansion. On the other hand, as the AST generated by a macro usually has only one root node that is added to the surrounding program's AST, the ways in which captures can occur are limited and therefore more predictable than the results of a program transformation.

Modularity, especially in non-functional languages like Java, is still a topic of ongoing research. Dynamic linking as shortly introduced in Chapter 5 not only makes it difficult to reason about name bindings, but also about variable typing. Ancona et. al. presented a new modularity concept for Java named *compositional compilation* that allows a flexible modular resolution while still ensuring type safety [5]. As compositional compilation allows the resolution of inter-modular references to distinct identifiers in contrast of Java's dynamic linking, it can make the application of modular name fixing easier. However, it doesn't solve the resolution in Refactoring situations, where no fixed compilation environment can be assumed at all.

An alternative approach to remove unintended captures without renaming identifiers was presented by Schäfer et. al. [9]. They demonstrated that for captures resulting from renaming refactorings in Java, the addition of qualifiers can be sufficient to restore the intended variable bindings. While this approach was generalized to be language-independent by de Jonge and Visser [10], it is still limited to the handling of renamed original identifiers and doesn't support new, synthesized ones. Yet, especially in the context of inter-modular captures, qualifiers could be the next possible step to handle captures that cannot be fixed by the algorithms presented in this thesis.

8 Conclusion and Future Work

While Erdweg, van der Storm and Dai laid the theoretical foundation to enable language-independent program transformations, the actual NameFix algorithm presented by them lacked several features that are essential for an actual application to a broad field of programming languages. This thesis outlined some of these shortcomings and presented extensions and conceptual modifications to address them while retaining the original underlying concepts of name fixing.

The fully extended version of NameFix lifted almost all restrictions of the structure of name graphs and, as a consequence, on the name binding schemes of the target languages. It was demonstrated that even situations that prevent an unambiguous name resolution can be handled using the defined extensions. Furthermore, it supports the handling of partitioned, interconnected name graphs and the staged application of name fixing on separate modules. The algorithms make it possible to add individual extensions to select and prioritize renamed identifiers and therefore adapt NameFix for individual use cases.

As demonstrated on the example of the programming language Lightweight Java, the presented extensions are sufficient to enable hygiene in both a local and a modular context for a realistic sample of widely used language features. While some more advanced features of Java like method overloading or generic types are not covered by this example, we are confident that that support for them and even further language features can be added by making use of the presented concepts.

A task that remains however is to formally prove the correctness of the presented algorithms. While some of the proofs presented in the context of the original NameFix algorithm may still be valid or can be trivially adapted, especially the modified definition of capture avoidance requires additional formal elaboration. The presented extensions also have an effect on other unsolved problems of NameFix: While modularity can reduce the complexity and consequently improve the performance of name resolution, it may not be as easy to integrate into existing compiler or transformation environments. Adapting each module system that needs to be supported to the presented concepts requires more language-specific work and reduces the flexibility of the algorithm.

Modular name resolution also increases the relevance of the open question when to actually apply NameFix: While the originally intention was to apply NameFix after each transformation step to ensure correct results for further transformations, this may be problematic as not every intermediate language may support modular name resolution. Additionally, as the actual, distinct identifiers of the referenced module interfaces must be known to apply modular NameFix, this can further complicate its implementation for Java. Without further enhancements to Java's module system, NameFix can only assume its application environment to be the same as the later execution environment, which may be a misjudgement in many real use cases.

For the presented modular algorithms, there is also room for further improvement and extensions to cover more of the not yet fixable cases. As already mentioned in Chapter 7, the addition of qualifiers can be used as a less invasive alternative to renamings and open up new name fixing possibilities. However, to remain independent of specific module systems and qualifier schemes, additional interfaces need to be defined so that the actual selection of suitable qualifiers can be performed by the programming language. In the context of the increasingly complex interfaces used by the original NameFix algorithm an the extensions presented in this thesis, it might be a reasonable step to develop a categorization of programming languages or transformation engines. Similar to the trait hierarchy used in Chapter 6, such categories could allow an easier identification which features of NameFix are supported in which scenario.

Bibliography

[1] Sebastian Erdweg, Tijs van der Storm and Yi Dai: Capture-Avoiding and Hygienic Program Transformations. In *Proceedings of European Conference on Object-Oriented Programming*, pages 489-514. Springer, 2014.

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha and Alex Buckley: The Java® Language Specification (Java SE 8 Edition), pages 163-170. 2015

[3] William Clinger and Jonathan Rees: Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1991.

[4] Daniel Weise and Roger Crew: Programmable syntax macros In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. ACM, 1993.

[5] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou and Elena Zucca: Polymorphic bytecode: compositional compilation for Java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–37. ACM, 2005.

[6] Sebastian Erdweg, Tillmann Rendel, Christian Kästner and Klaus Ostermann: SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 391–406. ACM, 2011.

[7] Rok Strniša: *Formalising, improving, and reusing the Java Module System*. PhD thesis, University of Cambridge, 2010.

[8] David Herman: *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University Boston, 2012.

[9] Max Schäfer, Torbjoörn Ekman and Oege de Moor: Sound and Extensible Renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 227–294. ACM, 2008.

[10] Maartje de Jonge and Eelco Visser: A Language Generic Solution for Name Binding Preservation in Refactorings. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, Article no. 2. ACM, 2012.