Modular Capture Avoidance for Program Transformations

Nico Ritschel Sebastian Erdweg
TU Darmstadt, Germany

Abstract

The application of program transformations and refactorings involves the risk of capturing variables, which may break the intended semantics of the transformed code. One way to resolve variable capture is by renaming of the involved identifiers. However, in a modular context, the renaming of exported declarations is undesirable (affecting a module's clients), and the renaming of imported declarations is impossible (requiring changes to third-party modules).

We present an algorithm *name-fix* that detects and eliminates variable capture modularly. We extend a previous non-modular version of *name-fix* in order to (i) minimize renamings of exported declarations, (ii) propagate necessary renamings of exported declarations to clients, and (iii) avoid renamings of imported declarations altogether. Together with support for transitive name bindings and conflicting declarations, our extensions to *name-fix* enable the application to real-world languages that feature separate compilation. To demonstrate the applicability of *name-fix*, we use it to modularly resolve variable capture for optimizations, refactorings, and desugarings of Lightweight Java.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation; D.2.2 [*Design Tools and Techniques*]: Modules and interfaces

Keywords variable capture; program transformation; refactorings; modules; hygienic macros

1. Introduction

Program transformations and generative programming find wide applications in software development. Application scenarios include the compilation of domain-specific languages (DSLs) [11], language extensions [7], macros [2, 12], refactorings [13], and automated code migration [8]. In many scenarios, program transformations are explicitly triggered by users but only change part of the user-provided source code. A major challenge for the developer of a program transformation is to make sure that the transformed code and the code that is left unchanged (including used libraries) continue to interact correctly.

Specifically, this paper addresses the problem of avoiding *variable capture* when transforming programs. When a transformation moves, adds, or alters named references or declarations in a program, there is always the possibility of a modified declaration to unintentionally capture an unchanged reference, or a modified reference to be unintentionally captured by an unchanged declaration. Such variable capture can confuse programmers, break the intention of the code or the transformation, and lead to bugs that are hard to detect and eliminate. A previous study [10] revealed that 9 out of 10 language workbenches (tools for DSL compilation) are prone to variable capture: It was possible to construct DSL programs that, when compiled, yielded ill-behaved code due to variable capture. The other language workbench used conventions across all transformation code to avoid variable capture. We seek a solution to variable capture that does not rely on conventions but provides guarantees instead.

As example for a transformation that can capture variables, consider a refactoring that rewrites public fields into private fields with getter and setter methods. The refactoring also changes all references to the public field to use the getter and setter methods instead. Figure 1(a) shows a simple Java program with two classes used to represent points. The second class MirroredPoint inherits from class Point and provides a method getY that yields the mirrored y-coordinate. Figure 1(b) shows the program after applying the refactoring: Fields x and y in class Point have become private, getter methods have been added, and the reference to y in MirroredPoint has been rewritten to a call to getY. However, the last rewriting causes name capture because getY is bound by the method in MirroredPoint rather than the one in Point.

While this unintended result may be attributed to the naive implementation of the used program transformation, ensuring capture avoidance during transformation is difficult and may require systematic renaming of some variables. In our example, we need to rename one of the declarations of getY. However, since getY is externally accessible, a

SLE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3686-4/15/10...
http://dx.doi.org/10.1145/2814251.2814260

```
1 class Point {
2       public int x, y;
3 }
4
5 class MirroredPoint extends Point {
6       public int getY() { return -y; }
7 }
```

(a) Example program before transformation.

```
1 class Point {
2       private int x, y;
3       public int getX() { return x; }
4       public int getY() { return y; }
5 }
6
7 class MirroredPoint extends Point {
8       public int getY() { return −getY(); }
9 }
```

(b) Example program after making fields private and adding getters.

Figure 1. Refactoring that introduces name capture.

renaming needs to involve the method's callers, some of which may not be known yet because they reside in user code.

In this paper, we present an algorithm that automatically ensures capture avoidance while supporting modular transformations in the sense of Java-style separate compilation. We build on previous work for automated capture avoidance that used a global post-processing to restore proper scoping [10]. We significantly extend this work by adding support for processing modules individually, where the renaming of a declaration in one module entails a deferred renaming of corresponding references in other modules that may become known later on. To this end, we accommodate modules with renaming interfaces that keep track of renamings and allow us to adopt client modules accordingly. Moreover, we add support for transitive name bindings (for example, transitive method overriding) and for repairing declaration conflicts (such as illegal overloading) that result from a transformation. These extensions are necessary to support automated capture avoidance for real-world programming languages.

Importantly, our algorithm for modular capture avoidance retains the following benefits of the previous global post-processing [10]:

- Noninvasive: To preserve code readability, our algorithm renames as few variables as possible. In particular, a program is left unchanged if it does not contain any captured variables. This is particularly important for refactorings and in the modular case, where exported names are visible to clients.
- Language-parametric: Our algorithm operates on name-binding graphs and can handle variable capture for all source and target languages that support modular static name resolution.
- Transformation-parametric: Our algorithm compares the name bindings of a program before and after transformation to detect and fix captured variables. We support any transformation engine that provides origin tracking [21, 22] for variable names, such that we can distinguish source names from synthesized names.

After revisiting the previous non-modular solution in Section 2, we make the following contributions.

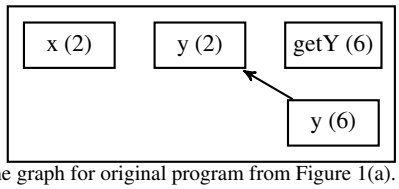
- We present *modular name graphs*, a generic representation for name bindings that models externally visible names through interfaces and distinguishes the bindings of individual modules.
- We describe a post-processing algorithm that operates on modular name graphs and modularly eliminates variable capture by propagating externally visible renamings to client modules.
- To minimize renaming of externally visible names, we developed a renaming optimization that, given a modular name graph and a renaming, tries to identify an equivalent renaming that is local to the module.
- We present an implementation of our algorithm in Scala and demonstrate its applicability to refactorings of programs written in Lightweight Java [20].

2. Non-modular Capture Avoidance

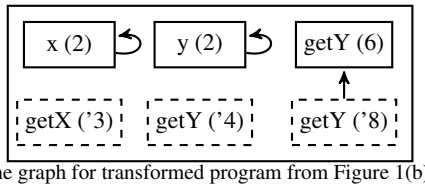
Our algorithm for modularly avoiding variable capture builds on previous work [10] for automatically avoiding variable capture through a global post-processing. In this section, we introduce the global algorithm as basis for our modular variant presented in the subsequent sections. We refer to the global algorithm as $name\text{-}fix\text{-}global$ and call our modular variant $name\text{-}fix$.

The first step to avoiding variable capture is to detect unintended capture. To this end, $name\text{-}fix\text{-}global$ compares the binding structure of the user program before and after transformation. That is, $name\text{-}fix\text{-}global$ requires name analyses of the original, non-transformed program as well as analysis of the transformed code.

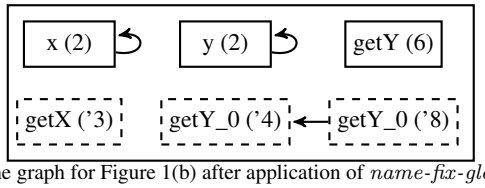
In order to be language-parametric, $name\text{-}fix\text{-}global$ uses its own simple representation of variable bindings, called *name graph*. For example, Figure 2(a) shows the name graph for the original program from Figure 1(a). Each node of a name graph corresponds to an occurrence of an identifier in the source code. In the graphs shown in this paper, we indicate the represented identifier through its name and line number in the source code. For example, Figure 2(a) shows that identifier y from Line 6 is a reference bound by identifier y from Line 2. For brevity, our example does not show nodes for class names. Technically, $name\text{-}fix\text{-}global$ assigns a unique



(a) Name graph for original program from Figure 1(a).



(b) Name graph for transformed program from Figure 1(b) with capture.



(c) Name graph for Figure 1(b) after application of *name-fix-global*.

Boxes represent references/declarations; arrows represent bindings. Numbers indicate line in source code where name occurs/originates. Dashed boxes represent names synthesized by a transformation.

Figure 2. Name graphs for Figures 1(a) and 1(b) with fixing.

ID to each identifier in the source code; we refer the interested reader to the original publication for details [10].

To be independent of the transformation engine, *name-fix-global* neither inspects nor instruments transformations. However, *name-fix-global* requires a transformation engine to support origin tracking [21, 22] for variable names such that it is possible to distinguish original identifiers from synthesized identifiers in generated code. For example, Figure 2(b) shows the name graph of the transformed program from Figure 1(b). We designate identifiers from the original program using solid boxes and their original line number whereas dashed boxes designate identifiers synthesized by a transformation. For synthesized identifiers, we use the line number in the generated program, with an additional tick mark ' for better readability. For example, the identifiers x and y originate from Line 2 in the original program, whereas the identifiers getX and getY in Line '3 and Line '4 of the generated code have been synthesized by the transformation. Note that the nodes for x and y represent both the field declarations in Line '2 of the generated code as well as the field accesses in Line '3 and Line '4 of the generated code, hence the self-reference in the name graph.

The name graph in Figure 2(b) shows the capture we discussed in the previous section clearly: The synthesized name getY ('8) accidentally references the original name getY (6) from class MirroredPoint instead of the synthesized definition getY ('4) from class Point. The problem is that the definition of getY (6) in class MirroredPoint shadows the definition in class Point. To fix this capture, one of the conflicting definitions has to be renamed. However, care needs to be taken to also rename all of the valid, non-captured references to the definition. In our example, we can either rename getY ('4) together with getY ('8) to a fresh name getY_0 as shown in Figure 2(c), or rename getY (6) to a fresh name (not shown).

Since *name-fix-global* needs to change all valid, non-captured references of a definition to soundly introduce fresh names, *name-fix-global* requires a global processing where it can manipulate the source code of all involved classes. This approach may be viable for small examples like the one above, but it does not scale to real software systems that involve many source files as well as libraries written by third parties. In particular, when developing an API used by others, some references to definitions reside with the clients of the API and are not available for a global processing. For real software systems, a modular solution for avoiding variable capture is needed.

3. Modular Name Graphs and the Problem of Modular Capture Avoidance

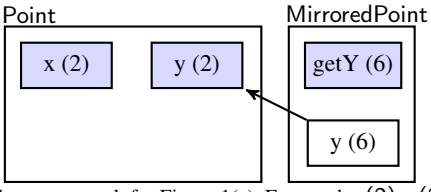
The concept of name graphs as introduced in the previous section is based on the assumption of a global, fully accessible name space. This assumption is at odds with good software-engineering principles, in particular with the principle of separating concerns into reusable components. Indeed, most real-world programs have external dependencies on libraries provided by others.

The problem of *name-fix-global* is that it disrespects module boundaries. As a first step toward modular *name-fix*, we define *modular name graphs*, which integrates modules boundaries into name graphs. Where a global name graph $G = (V, \rho)$ consists of a set of identifier occurrences $V \subset ID$ and a set of edges $\rho \subset V \times V$ that represent the binding structure of an entire program, a modular name graph only represents the binding structure of a single module. That is, the binding structure of a program with multiple modules is represented by a set of modular name graphs, one modular name graph for each module.

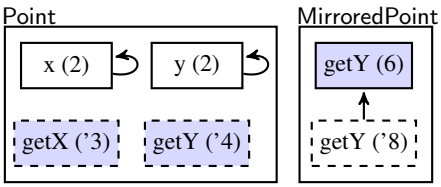
A module can export declarations to other modules and a module can refer to declarations defined in other modules. To allow for encapsulation and information hiding, we equip modular name graphs with interfaces that expose externally visible declarations. A module may only refer to those declarations of another module that are part of the interface.

Definition 1. *A name-graph interface I defines a set of externally referable identifier occurrences $I^{exp} \subset ID$. For a sequence of name-graph interfaces $\sigma = (I_1, \ldots, I_n)$, we define $\sigma^{exp} = I_1^{exp} \cup \ldots \cup I_n^{exp}$.*

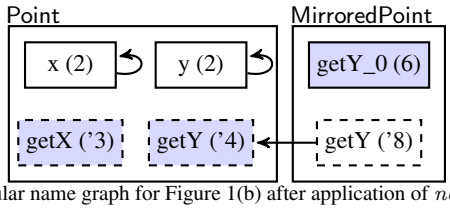
Later, we will extend the definition of name-graph interfaces to additionally propagate externally visible renamings to



(a) Modular name graph for Figure 1(a). Exported: x(2), y(2), and getY(6).



(b) Modular name graph for Figure 1(b). Synthesized exports: getX, getY('4).



(c) Modular name graph for Figure 1(b) after application of *name-fix*.

Boxes represent references/declarations; arrows represent bindings. Dashed boxes represent names synthesized by a transformation. Shaded boxes represent declarations exported by a module.

Figure 3. Modular name graphs for Figure 1(a) and 1(b).

client modules. The definition of modular name graphs is unaffected by this extension.

Definition 2. *A modular name graph is a tuple* $G = (V, \sigma, \rho, I)$ *where*

$V \subset ID$	*is the set of locally occurring identifiers,*
$\sigma \subset I \times \cdots \times I$	*is the sequence of interfaces the name graph is linked against,*
$\rho \subset V \times (V \cup \sigma^{exp})$	*is the set of edges that bind locally occurring references to local or external declarations, and*
I	*is the interface of the name graph.*

For example, consider again the original program from Figure 1(a) and the transformed program from Figure 1(b). Whereas Figure 2 shows the global name graph for both modules, we can use modular name graphs to inspect the binding structure of each module individually, making cross-module dependencies explicit. We show the modular name graphs of the original and transformed program in Figure 3.

Figure 3(a) shows the modular name graphs of Point and MirroredPoint as defined in the original program. We draw each modular name graph as a box containing the locally occurring identifiers and their bindings. We mark exported declarations of a module as name nodes with a colored background. Bindings to the interface of another module appear as edges between modular name graphs. Figure 3(b) shows the modular name graphs after transformation. As before, we mark names that are synthesized by a transformation using dashed boxes. A synthesized name that is exported accordingly shows up in a dashed box with colored background.

```
1 class PointUtil {
2     public int compareY(MirroredPoint a, MirroredPoint b) {
3         return a.getY() - b.getY();
4     }
5 }
```

Figure 4. Client of MirroredPoint's method getY.

Since modular name graphs contain the same edges as global name graphs, the variable capture of getY('8) is still easy to detect. But, in contrast to global name graphs, modular name graphs allow us to distinguish local declarations from exported ones as well as local references from references to declarations in other modules. This is important in our example. Given the capture in class MirroredPoint, *name-fix-global* renamed getY('4) in class Point to getY_0('4) as shown in Figure 2(c) of the previous section. That is, an exported declaration was renamed due a conflict in a client module. However, renaming declarations in modules provided by others is not possible in general and needs to be avoided by *name-fix*.

Fortunately, an alternative renaming is possible: We can rename getY(6) in class MirroredPoint to getY_0(6) as shown in Figure 3(c). This renaming fixes the variable capture without imposing changes to class Point. However, this renaming may in turn affect clients of MirroredPoint because MirroredPoint exports the renamed declaration getY_0(6). Of course, it would have been preferable to select a renaming that does not change any exported declaration. But, in general and in our example, such renaming does not exist and we have to rename exported declarations.

Consider class PointUtil in Figure 4 that contains calls to MirroredPoint's method getY. When renaming the declaration of getY, these references will fail to resolve. Note that the renaming in MirroredPoint was performed modularly, that is, without knowledge of the client class PointUtil. Therefore, we need to propagate renamings of exported declarations to a module's clients and clients must adopt interface renamings in their own code.

In the following section, we present an algorithm *name-fix* that modularly eliminates variable capture while retaining the interoperability of clients. Specifically, *name-fix* solves the following *problem of modular capture avoidance*:

- Eliminate variable capture between original names and names synthesized by a transformation.

- Support separate compilation such that variable capture in one module does not necessitate change of unrelated modules or of modules imported by the module.
- Avoid renaming exported declarations when possible.
- Preserve capture-free cross-module bindings when the renaming of exported declarations is unavoidable.

4. Modular Name-Fix

In the previous section, we defined the problem of modular capture avoidance. In this section, we present our solution *name-fix*. Our solution takes a possibly transformed module, its original modular name graph, and list of possibly transformed interfaces the module is linked against. Our solution operates in two phases. First, we eliminate any local and intermodular variable capture of the module assuming the interfaces of imported modules were not transformed and assuming we can apply renamings to those interfaces. This phase yields a modular name graph that is structurally correct but possibly uses incorrect interfaces for imported modules. In the second phase, *name-fix* uses the possibly transformed interfaces and applies interface-preserving renamings to the module in order to establish the binding structure discovered in the first phase.

4.1 Detecting and Eliminating Capture within Modules

Before we can eliminate variable capture, we must first detect it. Like *name-fix-global* (Section 2), we identify captured identifiers by inspection of the name graph. There are three kinds of bindings that represent variable capture in the transformed program [10]:

- Bindings from original to synthesized identifiers.
- Bindings from original identifiers to other original identifiers not targeted before the transformation.
- Bindings from synthesized to original identifiers.

The previous algorithm *name-fix-global* requires name graphs to be bipartite and non-ambiguous, that is, each identifier is either a reference or a declaration but not both and each reference uniquely refers to a declaration. This rules out transitive, cyclic, and ambiguous bindings and prevents support for language constructs such as overloading (multiple targets for a single reference) and overriding (transitive reference to a virtual method). In fact, our running example from Figure 1(b) contains a transitive binding because method getY in MirroredPoint overrides getY in Point. We ignored this dependency in our discussion above; Figure 5 shows the actual name graph for Figure 1(b). Note that in this example the overriding is also a variable capture because it is a reference from an original to a synthesized identifier.

To support arbitrarily complex transitive and ambiguous binding structure, we define a generalized notion of variable capture based on the connectivity of identifiers.

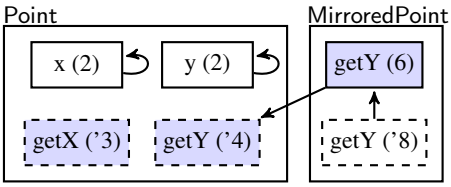


Figure 5. Graph for Figure 1(b) with transitive reference.

Definition 3. *Given a modular name graph $G = (V, \sigma, \rho, I)$, two identifiers v_1 and v_2 are connected $v_1 \sim_G v_2$ if $v_1 = v_2$, $(v_1, v_2) \in \rho$, $(v_2, v_1) \in \rho$, or there is a v_3 such that $v_1 \sim_G v_3$ and $v_3 \sim_G v_2$.*

Connectivity is an equivalence relation as it is reflexive, symmetric, and transitive. We write $[v]_G$ for the equivalence class of v under $\sim_G$. Using the connectivity of identifiers, we can detect all kinds of variable capture from above as follows.

Definition 4. *Let s be a source program that was transformed into a target program t, and let G_s and G_t be their respective modular name graphs. An identifier $v \in G_s$ induces capture in G_t if $[v]_{G_t} \not\subseteq [v]_{G_s}$.*

That is, an identifier v from the source program may only be connected to those identifiers in the target program it was connected to in the source program already. Since connectivity treats declarations and references alike, this definition covers all kinds of variable capture listed above, while additionally supporting transitive and ambiguous bindings. Function *find-capture* in Figure 6 implements this definition and yields the set of identifiers involved in variable capture. For example, the modular name graphs of Point from Figure 5 contains four equivalence classes with a single element each; the modular name graph of MirroredPoint contains a single equivalence class $[\text{getY (6)}]_{G_t}$ that contains all three identifiers named getY. In contrast, $[getY(6)]_{G_s}$ is a singleton set in the source graph for MirroredPoint of Figure 3(a). Accordingly, because $[\text{getY (6)}]_{G_t} \not\subseteq [\text{getY (6)}]_{G_s}$, *find-capture* finds as variable capture the identifier getY (6).

After detecting all captured identifiers in a module, the next step is to eliminate the capture through systematic renaming. To this end, function *comp-renaming* in Figure 6 produces a renaming function π. For each captured identifier v in capturedVars, *comp-renaming* generates a fresh name and renames all identifiers connected to v in the source name graph G_s from before the transformation. This way the renaming produced by *comp-renaming* renames the captured identifier and all identifiers that are *intended* to be connected according to the source program. Since the new name of v and its connections is fresh, the renaming eliminates the capture: After the renaming of m_t to m'_t we have $[v]_{G_{t'}} \subseteq [v]_{G_s}$ because only identifiers in $[v]_{G_s}$ carry the fresh name. For example, for MirroredPoint in Figure 5, *comp-renaming* produces a renaming $\pi = \{\text{getY (6)} \mapsto \text{"getY\_0"}\}$. If getY (6) was re-

```
Syntactic conventions for all pseudo-code figures:
  m^@v = n                          name n of identifier v in module m
  G_x  = (V_x, σ_x, ρ_x, I_x)       modular name graph G_x

Required functions
  resolve : m_x × σ_x → G_x         name resolution of module m_x
                                    against interfaces σ_x
  gensym : Name × 2^Name → Name     generate fresh name
  rename : m × (ID → Name) → m      rename module
  rename : σ × (ID → Name) → σ      rename interfaces


find-capture(G_s, G_t) = {
  return {v | v ∈ (V_s ∪ σ_s^exp), [v]_G_t ⊈ [v]_G_s};
}

comp-renaming(G_s, G_t, m_t, captureVars) = {
  π = ∅;
  usedNames = {m_t^@v | v ∈ V_t ∪ σ_t^exp};
  capturedClasses = {[v_0]_G_s | v_0 ∈ captureVars};
  foreach [v]_G_s in capturedClasses {
    fresh = gensym(m_t^@v, usedNames);
    π = π ∪ {(v_0 ↦ fresh) | v_0 ∈ [v]_G_s};
    usedNames = usedNames ∪ {fresh}
  }
  return π;
}

name-fix-virtual(G_s, m_t, σ_t) = {
  G_t = resolve(m_t, σ_t);
  captureVars = find-capture(G_s, G_t);

  if (captureVars = ∅) return G_t;

  π = comp-renaming(G_s, G_t, m_t, captureVars);
  m'_t = rename(m_t, π)
  σ'_t = rename(σ_t, π)
  return name-fix-virtual(G_s, m'_t, σ'_t);
}
```

Figure 6. Capture detection and elimination (first phase).

lated to other identifiers in the source graph, *comp-renaming* would rename those identifiers as well.

While the renaming produced by *comp-renaming* correctly eliminates variable capture, it ignores the existence of modules and interfaces. In particular, the renaming produced by *comp-renaming* may require the renaming of externally declared identifiers. Thus, *comp-renaming* allows us to construct a name graph that is capture-free but possibly uses renamed interfaces.

To this end, we define an algorithm that only *simulates* renamings of externally declared identifiers. While these renamings cannot actually be applied to imported modules, the simulated renaming of interfaces enables the generation of a *virtual name graph* where the module is linked against the virtually renamed interfaces. Since the binding structure in the virtual name graph is correct, the virtual name graph serves as an oracle in the second phase of *name-fix*, where we try to achieve a structurally equivalent graph by only applying renamings to local names.

Function *name-fix-virtual* in Figure 6 implements the extraction of virtual name graphs and takes as input a modular source graph G_s, a transformed module m_t, and the interfaces of linked transformed modules σ_t. The algorithm uses the language-specific function *resolve* to retrieve a modular name graph for m_t given the interfaces σ_t. Note that *name-fix-virtual* requires the interfaces σ_t exactly as they were generated by the transformation, that is, without renamings that *name-fix* may apply in these modules. Otherwise, name resolution fails to resolve references from m_t to externally declared identifiers. After name resolution, *name-fix-virtual* applies the previously defined functions *find-capture* and *comp-renaming* to detect captures and to compute renamings for fixing them. Then, *name-fix-virtual* applies the renaming to module m_t and to the interfaces σ_t, thus simulating the renaming of externally declared identifiers. Since the elimination of some captured references can cause other, previously hidden variable capture to appear, it is necessary to detected and eliminate variable capture recursively; the termination proof of *name-fix-global* [10] carries over to *name-fix-virtual*.

4.2 Eliminating Capture across Modules

Function *name-fix-virtual* detects and eliminates variable capture, but it has two limitations. First, it renames external declarations that are exported by other modules. Second, it does not handle renamings that occur in imported modules and instead assumes the linked interfaces to be unchanged. For this reason, we only use the fixed name graph provided by *name-fix-virtual* as an oracle for eliminating capture from the transformed module in a way that retains imported variable declarations and propagates any renaming applied to them.

As shown in Figure 7, our algorithm *name-fix* first computes a module's virtual name graph. Since *name-fix-virtual* cannot handle interface renamings, we call it with the original interfaces σ_t^{orig} that are extracted and stored after transformation but before name-fixing. Our algorithm *name-fix* now modifies a transformed module in two steps. First, *name-fix* computes and applies renamings that ensure correct bindings from local references to interface declarations (function *restore-interface-bindings*). In particular, this step takes care that local references are correctly adapted to interface renamings. Second, we compute and apply renamings that eliminate unintended bindings (function *remove-capture*). Both steps use the virtual name graph as an oracle for determining intended and unintended bindings.

Function *restore-interface-bindings* iterates over the classes of names connected in the virtual name graph. For

```
name-fix(G_s, m_t, σ_t) = {
  G_virt = name-fix-virtual(G_s, m_t, σ_t^orig);
  m'_t = restore-interface-bindings(m_t, σ_t, G_virt);
  return remove-capture(m'_t, σ_t, G_virt);
}

restore-interface-bindings(m_t, σ_t, G_virt) = {
  G_t = resolve(m_t, σ_t);
  π = ∅;
  virtClasses = {[v]_G_virt | v ∈ V_virt}
  foreach [v]_G_virt in virtClasses {
    lost = {v_0 | v_0 ∈ σ_t^exp, v_0 ∈ [v]_G_virt, [v]_G_virt ⊈ [v_0]_G_t};
    if (lost ≠ ∅) {
      extNames = {m_t^@v_0 | v_0 ∈ σ_t^exp, v_0 ∈ [v]_G_virt};
      if (|extNames| = 1) {
        π = π ∪ {(v_0 ↦ n) | v_0 ∈ [v]_G_virt, n ∈ extNames};
      }
      else fail('Inconsistent renaming of interface names.');
    }
  }
  return rename(m_t, π);
}

remove-capture(m_t, σ_t, G_virt) = {
  G_t = resolve(m_t, σ_t);
  π = ∅;
  usedNames = {m_t^@v | v ∈ V_t ∪ σ_t^exp};
  virtClasses = {[v]_G_virt | v ∈ V_virt}
  foreach [v]_G_virt in virtClasses {
    capture = {v_1 | v_0 ∈ [v]_G_virt, v_1 ∈ [v_0]_G_t, v_1 ∉ [v_0]_G_virt};
    if (capture ≠ ∅) {
      fresh = gensym(m_t^@v, usedNames);
      π = π ∪ select-renaming([v]_G_virt, capture, fresh, G_t);
      usedNames = usedNames ∪ {fresh}
    }
  }
  if (π = ∅) return m_t;
  m'_t = rename(m_t, π);
  return remove-capture(m'_t, σ_t, G_virt);
}

select-renaming(V_1, V_2, fresh, G_t) = {
  if (V_1 ∩ σ_t^exp = ∅)
    return {(v ↦ fresh) | v ∈ V_1};
  else if (V_2 ∩ σ_t^exp = ∅)
    return {(v ↦ fresh) | v ∈ V_2};
  else fail('Unable to find fix without renaming interfaces.');
}
```

Figure 7. Eliminate capture across modules (second phase).

each class, it checks if there is an external name from an interface $v_0 \in \sigma_t^{exp}$ with references in the virtual name graph $v_0 \in [v]_{G_{virt}}$ that are missing in the name graph of the transformed program $[v]_{G_{virt}} \not\subseteq [v_0]_{G_t}$. If an external identifier with lost references exists, we compute the set names of all external declarations referred to by $[v]_{G_{virt}}$. Name fixing fails if there is more than one name, which means that two or more external declarations originally shared the same name but were renamed inconsistently to different names, even though there is a shared reference according to G_{virt}. If instead there is a unique new name for all external declarations, we rename all intended local references to use that name.

Let us exemplify how *restore-interface-bindings* propagates interface renamings. As discussed in the previous subsection, *name-fix* renames the method declaration getY (6) in class MirroredPoint to getY_0. When applying *name-fix* to a client of MirroredPoint such as class PointUtil in Figure 4, *name-fix-virtual* will determine that PointUtil refers to getY (6) in MirroredPoint. However, when resolving the name binding of PointUtil in the transformed interface of MirroredPoint, the method references end up being unbound, because the method declaration was renamed. Therefore, we get lost = {getY_0 (6)} in *restore-interface-bindings*, because the intended bindings from G_{virt} are missing in G_t. Function *restore-interface-bindings* finds the unique external name getY_0 to rename the references in PointUtil. This way, the interface renaming of class MirroredPoint is propagated to its clients.

After restoring interface bindings, we remove variable capture as determined by the virtual name graph. To this end, function *remove-capture* iterates over the classes of names connected in the virtual name graph. We eliminate the binding of any identifier v_1 that is connected to an identifier v_0 in the transformed module $v_1 \in [v_0]_{G_t}$ but is not connected to that identifier in the virtual name graph $v_1 \notin [v_0]_{G_{virt}}$.

Apart from how capture is determined, *remove-capture* is similar to *comp-renaming*. One additional significant difference is that *comp-renaming* is allowed to rename externally declared identifiers because it is only used to construct a virtual name graph. In contrast, *remove-capture* may only rename identifiers that are local to the current module m_t. Here, *remove-capture* has two choices: Rename the identifiers $[v]_{G_{virt}}$ that validly refer to each other, or rename the identifiers capture that invalidly refer to the ones in $[v]_{G_{virt}}$. We use function *select-renaming* to select one of the two possibilities such that no externally declared identifier gets renamed, if possible. Name fixing fails in *select-renaming* if the renaming of externally declared identifiers is unavoidable. This only occurs when two or more external declarations had different names but were renamed to the same name. In practice, this is easy to avoid by ensuring that fresh names do not overlap between modules.

4.3 Avoiding Interface Renamings

Algorithm *name-fix* as presented so far solves most requirements of the *problem of modular capture avoidance* (Section 3): *name-fix* eliminates capture, supports separate compilation, and preserves cross-module bindings. However, *name-fix* currently ignores which declarations get exported by the transformed module m_t. If possible we want to avoid renaming exported declarations in order to confine the impact

```
select-renaming(V1, V2, fresh, Gt) = {
  if ((V1 ∪ V2) ∩ σt^exp = ∅) {
    if (|V1 ∩ It^exp| < |V2 ∩ It^exp|)
      return {(v ↦ fresh) | v ∈ V1};
    else
      return {(v ↦ fresh) | v ∈ V2};
  }
  else if (V1 ∩ σt^exp = ∅)
    return {(v ↦ fresh) | v ∈ V1};
  else if (V2 ∩ σt^exp = ∅)
    return {(v ↦ fresh) | v ∈ V2};
  else
    fail('Unable to find fix without renaming interfaces!');
  }
}
```

Figure 8. Minimize renamings of exported declarations.

```
type Renaming = Identifier => Identifier
trait NameInterface {
  def export: Set[Identifier]
  def original: NameInterface
  def rename(renaming: Renaming): NameInterface
}
trait NominalModular[I <: NameInterface] {
  def allNames: Set[String]
  def rename(renaming: Renaming): NominalModular[I]
  def interface: I
  def link(dependencies: Set[I]): NominalModular[I]
  def resolveNamesModular: NameGraphModular[I]
}
```

Figure 9. Interfaces for language-specific functionality.

of a transformation and the required propagation of renamings as much as possible. This is also in accordance with *name-fix*'s original design goal of non-invasiveness, that is, renaming as few identifiers as possible.

As shown in Section 4.2, for each occurrence of variable capture, there are two possible renamings to choose from. Function *select-renaming* selects one of the renamings such that no externally declared variable gets renamed. This criteria is of highest priority because renaming external declarations is invalid. If, however, both renamings are valid and only rename local identifiers, we are free to choose the less invasive renaming.

Figure 8 shows an extended version of *select-renaming* which explicitly handles cases where both renamings are valid. To minimize the impact of *name-fix* on clients of a module, we select the renaming that changes fewer declarations exported by the current module. To this end, we look up the interface I_t of the modular name graph G_t and extract the exported identifiers occurrences I_t^{exp}.

Depending on the used programming language and module system, there may be more fine-grained nuances regarding exported names. For example, in Java it is favorable to rename package-internal identifiers rather than publicly visible ones and maybe even to rename non-final declarations rather than final ones. Such extensions can be integrated easily into *select-renaming* by altering the condition that leads to the selection of one or the other renaming.

5. Case Studies

To evaluate our approach for modular capture avoidance, we have implemented modular *name-fix* as presented in Section 4 in Scala and applied it to different types of program transformations. As target languages, we have used a language based on lambda calculus extended with support for simple Haskell-like modules, and Lightweight Java [20], a stateful subset of the programming language Java extended with Java-like access modifiers.

Our implementation is language-parametric. As shown in Figure 9, we provide interfaces for the definition of language-specific functionality required by *name-fix*. Trait NameInterface provides the necessary information for interfaces: The exported declarations, the original non-renamed interface, and a renaming function for simulated interface renaming as performed by *name-fix-virtual*. The instances of trait NominalModular represent modules of the target language. We require methods to retrieve all names that occur in the module (to generate fresh names), to rename the identifiers in a module, to extract a module's interface, to link a module against a set of interfaces, and to modularly resolve the bindings of a module against the previously linked interfaces. Classes that implement NameInterface or NominalModular can freely add further language-specific functionality not directly required by *name-fix*. For example, it is possible to store types in an interface and to check well-typedness of a module against them. This enables support also for those languages where name analysis requires additional context information such as typing.

Our implementation is transformation-parametric. The only requirement of *name-fix* is that the transformation engine tracks the origin of identifiers from the original to the transformed program. To simplify identifier tracking name graph generation, we have implemented a generic representation of identifiers (class Identifier) that uses the identity of JVM objects to distinguish identifiers of the same name. To track the origin of identifiers in our target languages, we embed Identifier objects directly into syntax trees and copy references to them into the transformed program, thus preserving object identity. However, our implementation of *name-fix* is not tight to this form of origin tracking; other techniques such as string origins [21] can be used just as well.

In our functional target language, we applied *name-fix* to repair the result of transformations implementing an *optimization* that partially evaluates programs and a *compiler phase*

that desugars first-class functions into first-order functions through lambda lifting. Through conflicting local definitions as well as the shadowing of imported identifiers, we have encountered and successfully fixed both local and inter-modular variable capture using *name-fix*.

For our case studies on Lightweight Java, we considered each class as a separate module that implicitly imports all other classes it contains references to. In Lightweight Java, the required usage of Java's dot-notation to access object members prevents some of the most obvious variable-capture scenarios. However, variable capture still occurs as the result of inheritance-related shadowing caused by the addition or modification of class members. To induce and fix such shadowing, we have applied *name-fix* on a *refactoring* that adds getter and setter methods as presented in the example from Figure 1, as well as a *language extension* that adds local variable declarations inside methods through a desugaring.

The source code of our implementation and all case studies is available online:
`http://github.com/seba-/hygienic-transformations`.

5.1 Modular Lambda

For our case study on a modular lambda language, we have extended a previously existing, non-modular implementation of the lambda calculus with a simple Haskell-like module system. A module has a name, can import the scope of other modules using their name, and defines a list of top-level definitions, all of which are exported by default. Local definitions shadow imported definitions. An import statement also represents an edge in the name graph: It refers to the identifier of the imported module. This way, *name-fix* eliminates variable capture for modules, definitions, and variables bound by lambda expressions.

Our definition of modular lambda does not support explicit qualifiers and having the same name exported by multiple imported modules leads to ambiguous references. We model such ambiguous references by making it refer to all potential definitions. If the ambiguity results from a transformation because a definition was added or renamed, *name-fix* will eliminate the ambiguity. Top-level definitions within a module have no precedence and consequently are in conflict if they share the same name.

We have implemented two transformations with modular lambda as the target language. First, we developed an optimization that partially evaluates program fragments through normalization. When the transformation encounters a beta redex $(\lambda x.b)e$, it replaces all occurrences of x in b by e. However, the optimization employs a capturing substitution function that disregards scoping and potentially produces variable capture involving references to variables and local and external definitions. A call to *name-fix* after the optimization finishes is sufficient to restore correct scoping. A detailed example of this transformation its definition was presented in the context of *name-fix-global* [10].

```
1 module M {
2   import Base; // defines "fun"
3   def calc = 2 + fun ((λx.x+x) 1)
4 }
```

(a) Module with local anonymous function definition.

```
1 module M {
2   import Base // defines "fun"
3   def fun0(x) = x+x
4   def calc = 2 + fun (fun0 1)
5 }
```

(b) Module with function lifted and capture fixed by *name-fix*.

Figure 10. Lambda lifting and application of *name-fix*.

As a second transformation, we implemented lambda lifting. Lambda lifting desugars anonymous first-class functions into top-level first-order functions by lifting the function definition and by introducing references to the newly created top-level definitions. Our transformation ensures that each lifted function is assigned a different name. However, our transformation does neither consider pre-existing top-level definitions in the local module nor pre-existing top-level definitions imported from other modules. Consequently, lambda lifting can cause local and inter-modular variable capture between definitions.

We show an example in Figure 10 where we lift the anonymous function (λx.x+x) shown in 10(a) to a top-level function definition with the generated name fun. This however causes the imported function fun to be shadowed and the local usage of the function to be captured. When *name-fix* is applied, it computes the virtual name graph and determines the necessary renamings. As the import involved in the conflict is externally bound, it can not be renamed. Instead, the lifted function is renamed to fun0.

5.2 Lightweight Java

Lightweight Java (LJ) is a reduced but stateful subset of the Java programming language. While a distinct module system was designed for LJ [20], it is different from the concepts used in Java and more similar to the module system we implemented for modular lambda. To evaluate *name-fix* for a module system that is closer to actual Java, we assume that each Lightweight Java class is a module and resolve references to external classes on demand. Moreover, the original definition of Lightweight Java doesn't provide support for access modifiers to control the visibility of class members. Although custom visibility is not essential for modularity, we have added Java-like *public* and *private* modifiers to facilitate testing of *name-fix*'s capabilities for avoiding interface renamings.

As all references to fields and methods have to be qualified in LJ, some common variable-capture scenario cannot occur. In particular, in LJ, fields cannot be hidden by local variables

```
1  class Base {
2    public int method(MyObject a) {
3      String b;
4      b = a.toString();
5      b = b.concat(b);
6      return this.methodHelper(b);
7    }
8
9    public int methodHelper(String s) {
10     return s.size();
11   }
12 }
```

(a) Class with local variable declaration.

```
1  class Base {
2    public int method(MyObject a) {
3      return this.methodHelper0(a, null);
4    }
5
6    private int methodHelper0(MyObject a, String b) {
7      b = a.toString();
8      b = b.concat(b);
9      return this.methodHelper(b);
10   }
11
12   public int methodHelper(String s) {
13     return s.size();
14   }
15 }
```

(b) Transformed class with added helper method.

Figure 11. Desugaring of local variable declaration.

or method parameters. However, shadowing can occur in the context of overrides when a class inherits from another class: The subclass can accidentally override and thus shadow a method from the superclass. As method overrides can alter the semantics of a class, they need to be handled like regular references when in comes to capture avoidance. An example modular name graph with an override reference was shown in Figure 5 in Section 4.1.

We implemented two transformations with LJ as the target language. First, we implemented the refactoring that we used as a running example throughout the paper. When modularly applying *name-fix* to classes Point, MirroredPoint, and PointUtil, *name-fix* behaves correctly: Point does not get changed; the method declaration getY in MirroredPoint gets renamed to getY_0, and the method references to getY in PointUtil both get renamed to getY_0.

As second transformation, we implemented a desugaring that extends LJ with an additional language feature, namely local variables. In LJ, variables can only be bound as parameters through method declarations, that is, it is necessary to provide all used variable values as method arguments. Our transformation replaces local variable declarations by a method call that refers to a newly created helper method. This helper method uses an additional parameter to bind the variable value.

For example, consider the LJ program in Figure 11(a) that contains a local variable declaration of name b, initialized to a.toString(). The method concatenates b with itself and calls methodHelper to compute its return value. The result of our desugaring appears in Figure 11(b). In place of the local variable declaration, the desugaring inserts a call to a generated method methodHelper and passes the variable value as argument. The generated method (Line 6) contains the code that followed the local variable declaration. However, since there was already a method named methodHelper in the original program, a variable capture occurred. We used *name-fix* to eliminate that variable capture. To this end, *name-fix* had to rename either of the declarations of methodHelper. As explained in Section 4.3, *name-fix* chose to rename the generated method because it is private and thus not part of the interface.

To summarize, we successfully applied *name-fix* to eliminate variable capture produced by four different transformations. The transformations implemented optimizations, language extensions, and refactorings. An evaluation of *name-fix* on an existing code base it outside the scope of this paper and will be the focus of future work.

6. Related Work

Transforming modular programs. While the application of modularity on program compilation has been a subject of previous work [1, 3], the separate application of program transformations has not been in the focus of research so far. Rule-based program transformation systems like Stratego/XT [23] or Rascal [15] implicitly provide basic support for transforming modular programs as rules or strategies can be separately applied on subsets of a program's syntax. However, this support is not comprehensive enough to allow the propagation of transformation effects to dependent modules that are unavailable when the transformation is applied. As a result, additional work is required by developers to allow a modular application of cross-module transformations like the refactoring presented in Figure 1(b). The usage *name-fix* can reduce this amount of work by providing a solution for modular transformation hygiene that is independent of the actual transformation logic.

Transformation hygiene. Extensive research on hygiene has been performed for specific types of program transformations. Especially the area of hygienic macro expansion has originated a large number of approaches [4, 6, 14, 16] and concepts similar to *name-fix* were researched in this area [6]. However, a key difference between *name-fix* and most hygiene approaches for macros is that *name-fix* is applied as a post-processing after the transformation was applied, while macro-based algorithms are usually applied as part of the expansion process. This is important to allow *name-fix* to

be independent of the transformed language and the transformation engine. Technically, because *name-fix* is a post-processing and not integrated into a language's compiler, *name-fix* cannot rely on a graph-based program representation used by the compiler internally [18]. Instead, *name-fix* needs to perform explicit renamings to make the fixed binding structure available for subsequent tools.

The *name-fix* algorithm presented in this paper is based on a previous version with the same name developed by Erdweg et. al. [10]. Our extended version of *name-fix* removes two key limitations of its predecessor, namely its inability to handle modular programs and transitive references. If applied on a program without external references or transitivity, the results of the previous and the current version of *name-fix* are similar, although not necessarily identical. The reason why the results may differ is that the *select-renaming* function we presented may select references to be renamed, while the original version always renamed definitions. The correctness of the original version of *name-fix* is formally proven, which is an open task for the algorithms presented in this paper.

An alternative approach to eliminating variable capture is the addition of qualifiers instead of variable renaming. Automated approaches for this have been presented for Java in particular [19] and for other languages generically [5]. However, adding qualifiers is limited to refactorings which do not introduce new identifiers. Moreover, these systems assume that a transformation always intends to preserve the original behavior of the target program. Yet, the idea of combining identifier qualification with renaming seems promising for future research and will be shortly discussed in Section 7.

Modelling name bindings. Neron, et. al. [17] described a more advanced representation of name graphs that provides a more expressive, yet still language-independent model of name relations in a program. The *scope graphs* they introduce allow an analysis of the consequences of renamings without re-resolving the underlying program. If combined with modularity features, scope graphs could support the incrementalization of *name-fix* and even a non-recursive implementation that is able to determine all required renamings in a single iteration.

7. Conclusion and Future Work

We presented *name-fix*, a generic, modular solution for eliminating variable capture resulting from program transformations. With *name-fix*, the effort of manually ensuring hygiene can be lifted from designers of transformations and replaced by a fully automated post-processing step. With our case studies on refactoring, optimization, and language extension for different programming languages, we demonstrated the versatility of *name-fix*. In particular, *name-fix* is non-invasive, language-parametric, and transformation-parametric.

An issue of modular capture avoidance compared to global approaches is the possibility of unsolvable situations. In our definition of *name-fix*, we have identified two problematic situations that could only be resolvable through renaming of external declarations. First, if an identifier refers to multiple external declarations and one of them was renamed, then it is impossible to rename the reference such that it again refers to all external declarations. Second, if there is variable capture between external declarations that are imported into the same module, then no local renaming can resolve this capture. As we have illustrated, there is no possible solution to solve these scenarios by only renaming local variables.

An alternative approach to solve variable capture that could be integrated as part of *name-fix* is the adoption of qualifiers and other program changes that circumvent the standard scoping of the underlying language. For example, some module systems allows import statements to bring only some definitions into the local, hiding the other definitions. Using such features, it may be possible to support the problematic situations that currently fail and the elimination may be even less invasive. On the downside, using advanced module-system features makes the elimination language-specific as not all languages provide the same module-system features.

In ongoing work, we explore the application of *name-fix* to existing Java code. We use Oracle's Java compiler to compute modular name graphs. We hope that this work will provide insights into the scalability and applicability of *name-fix* on large-scale code bases and will enable support for capture avoidance in the SugarJ framework [9].

Acknowledgments

We thank the reviewers for helpful feedback.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 609–636. Springer, 2002.

[2] E. Burmako. Scala macros: Let our powers combine! In *Proceedings of Scala Workshop*, pages 3:1–3:10. ACM, 2013.

[3] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 266–277. ACM, 1997.

[4] W. Clinger and J. Rees. Macros that work. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.

[5] M. de Jonge and E. Visser. A language generic solution for name binding preservation in refactorings. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*. ACM, 2012.

[6] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[7] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.

[8] S. Erdweg, S. Fehrenbach, and K. Ostermann. Evolution of software systems with extensible languages and DSLs. *IEEE Software*, 31(5):68–75, 2014.

[9] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM, 2013.

[10] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 489–514. Springer, 2014.

[11] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[12] M. Flatt. Creating languages in Racket. *Communication of the ACM*, 55(1):48–56, 2012.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

[14] D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, Massachusetts, 2012.

[15] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *Proceedings of Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177, 2009.

[16] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of Conference on LISP and Functional Programming (LFP)*, pages 151–161. ACM, 1986.

[17] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *Proceedings of European Symposium on Programming (ESOP)*, pages 205–231. Springer, 2015.

[18] J. Pombrio and S. Krishnamurthi. Hygienic resugaring of compositional desugaring. In *Proceedings of International Conference on Functional Programming (ICFP)*, 2015.

[19] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 277–294. ACM, 2008.

[20] R. Strniša. *Formalising, improving, and reusing the Java module system*. PhD thesis, University of Cambridge, 2010.

[21] P. I. Valdera, T. van der Storm, and S. Erdweg. Tracing model transformations with string origins. In *Proceedings of International Conference on Model Transformations (ICMT)*, pages 154–169. Springer, 2014.

[22] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Symbolic Computation*, 15:523–545, 1993.

[23] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.