arXiv:1404.5770v1 [cs.PL] 23 Apr 2014

Capture-Avoiding and Hygienic Program Transformations (incl. Proofs)

Sebastian Erdweg[1], Tijs van der Storm[2,3], and Yi Dai[4]

[1] TU Darmstadt, Germany [2] CWI, Amsterdam, The Netherlands
[3] INRIA Lille, France [4] University of Marburg, Germany

Abstract. Program transformations in terms of abstract syntax trees compromise referential integrity by introducing variable capture. Variable capture occurs when in the generated program a variable declaration accidentally shadows the intended target of a variable reference. Existing transformation systems either do not guarantee the avoidance of variable capture or impair the implementation of transformations.
We present an algorithm called *name-fix* that automatically eliminates variable capture from a generated program by systematically renaming variables. *name-fix* is guided by a graph representation of the binding structure of a program, and requires name-resolution algorithms for the source language and the target language of a transformation. *name-fix* is generic and works for arbitrary transformations in any transformation system that supports origin tracking for names. We verify the correctness of *name-fix* and identify an interesting class of transformations for which *name-fix* provides hygiene. We demonstrate the applicability of *name-fix* for implementing capture-avoiding substitution, inlining, lambda lifting, and compilers for two domain-specific languages.

1 Introduction

Program transformations find ubiquitous application in compiler construction to realize desugarings, optimizers, and code generators. While traditionally the implementation of compilers was reserved for a selected few experts, the current trend of domain-specific and extensible programming languages exposes developers to the challenges of writing program transformations. In this paper, we address one of these challenges: capture avoidance.

A program transformation translates programs from a source language to a target language. In doing so, many transformations reuse the names that occur in a source program to identify the corresponding artifacts generated in the target program. For example, consider the compilation of a state machine to a simple procedural language as illustrated in Figure 1. The state machine has three states `opened`, `closed`, and `locked`. For each state the compiler generates a constant integer function with the same name. Furthermore, for each state the compiler generates a dispatch function that takes an event and depending on the event returns the subsequent state. For example, the dispatch function for `opened` tests if the given event is `close` and either yields the integer constant representing the following

```
state opened
  close => closed

state closed
  lock => locked
  open => opened

state locked
  unlock => closed
```

(a) Door state machine.

```
1  fun opened() = 0;
2  fun closed() = 1;
3  fun locked() = 2;
4  fun opened-dispatch(event) =
5    if (event == "close") then closed() else error();
6  fun closed-dispatch(event) =
7    if (event == "open") then opened()
8    else if (event == "lock") then locked() else error();
9  fun locked-dispatch(event) =
10   if (event == "unlock") then closed() else error();
11 fun main-dispatch-next-event(state, event) =
12   if (state == opened()) then opened-dispatch(event)
13   else if (state == closed()) [...];
```

(b) Program generated for the door state machine.

Fig. 1. Many transformations reuse names from the source program in generated code.

state closed or a dynamic error. Finally, the compiler generates a main dispatch function that calls the dispatch function of the current state.

A naive implementation of such compiler is easy to implement, but also runs the risk of introducing variable capture. For example, if we consistently rename the state locked to opened-dispatch as shown in Figure 2(a), we expect the compiler to produce code that behaves the same as the code generated for the state machine without renaming. However, a naive compiler blindly copies the state names into the generated program, which leads to the incorrect code shown in Figure 2(b): The function definition on line 4 shadows the constant function on line 3 and thus captures the variable reference opened-dispatch on line 8 (we assume there is no overloading). For the example shown, the problem is easy to fix by renaming the dispatch function on line 4 and its reference on line 12 to a fresh name opened-dispatch-0. However, a general solution is difficult to obtain. Existing approaches either rely on naming conventions and fail to guarantee capture avoidance, or they require a specific transformation engine and affect the implementation of transformations.

We propose a generic solution called *name-fix* that guarantees capture avoidance and does not affect the implementation of transformations. *name-fix* compares the name graph of the source program with the name graph of the generated program to identify variable capture. If there is variable capture, *name-fix* systematically and globally renames variable names to differentiate the captured variables from the capturing variables, while preserving intended variable references among original variables and among synthesized variables, respectively. *name-fix* requires name analyses for the source and target languages, which often exists or are needed anyway (e.g., for editor services, error checking, or refactoring), and hence can be reused. *name-fix* treats transformations as a black box and is independent of the used transformation engine as long as it supports origin tracking for names [26].

```
state opened
  close  =>  closed

state closed
lock=>opened-dispatch
  open  =>  opened

state opened-dispatch
  unlock  =>  closed
```

(a) Consistently renaming door state machine.

```
 1  fun opened() =  0;
 2  fun closed() =  1;
 3  fun opened-dispatch() =  2;
 4  fun opened-dispatch(event) =
 5    if (event == "close") then closed() else error();
 6  fun closed-dispatch(event) =
 7    if (event == "open") then opened()
 8    else if (event == "lock") then opened-dispatch() else ...
 9  fun opened-dispatch-dispatch(event) =
10    if (event == "unlock") then closed() else error();
11  fun main-dispatch-next-event(state, event) =
12    if (state == opened()) then opened-dispatch(event)
13    else if (state == closed()) [...];
```

(b) Program generated for the renamed door state machine is incorrect: Variable capture of opened-dispatch.

Fig. 2. Variable capture can occur when original and synthesized names are mixed.

name-fix enables developers of program transformations to focus on the actual translation logic and to ignore variable capture. In particular, *name-fix* enables developers to use simple naming schemes for synthesized variables in the transformation and to produce intermediate open terms. For example, in Figure 1, we append `"-dispatch"` to a state's name to derive the name of the corresponding dispatch function. This construction occurs at two independent places in the transformation: When generating a dispatch function for a state, and when generating the main dispatch function. The connection between these is only established when assembling all parts of the generated program in the final step of the transformation. Using *name-fix*, it is safe to apply global naming schemes with intermediate open terms to associate generated variable references and declarations. Transformations of this kind fall into the class of transformations for which *name-fix* guarantees hygiene, that is, α-equivalent source programs are always mapped to α-equivalent target programs.

In summary, we make the following contributions:

- We studied 9 existing DSL implementations that use transformations and found that 8 of them were prone to variable capture.
- We present *name-fix*, an algorithm that automatically eliminates variable capture from the result of a program transformation.
- We state and verify termination and correctness properties for *name-fix* and show that *name-fix* produces α-equivalent programs for programs that are equal up to consistent but possibly capturing renaming.
- We propose a notion of hygienic transformations and identify an interesting class of transformations for which *name-fix* provides hygiene.
- We present an implementation of *name-fix* in the metaprogramming system Rascal. Our implementation supports capture avoidance for transformations that generate code as syntax trees or as strings.

– We demonstrate the applicability of *name-fix* in a wide range of scenarios: for capture-avoiding substitution, for optimization (function inlining), for desugaring of language extensions (lambda lifting), and for code generation (compilation of DSLs for state machines and for digital forensics).

2 Capture-avoiding transformations: What and why

Capture avoidance is best known from capture-avoiding substitution: When substituting an expression e_2 under a binder as in $\lambda\, x.\ (e_1[y := e_2])$, variable x may not occur free in e_2 otherwise the original binding of x in e_2 would be shadowed by the λ. To implement capture-avoiding substitution, we must rename x to a fresh variable $\alpha \notin \{y\} \cup FV(e_1) \cup FV(e_2)$ to avoid the capture: $\lambda\, \alpha.\ (e_1[x := \alpha][y := e_2])$. Ensuring capture avoidance is already relatively complicated for substitution in the λ-calculus. For larger languages and more complex program transformations, ensuring capture avoidance is a non-trivial and error-prone task.

2.1 Variable capture in the wild

To better understand the relevance of the problem of variable capture, we studied implementations of a DSL for questionnaires in 10 state-of-the-art language workbenches in the context of the Language Workbench Challenge 2013 [9].[1] The questionnaire DSL features named declarations of questions and named definitions of derived values. 9 of the 10 language workbenches translate a questionnaire into a graphical representation using either Java or HTML with CSS and JavaScript as target language. One workbench uses interpretation instead of transformation. In most cases, the implementation of the DSL was conducted by the developers of the workbench themselves.

The result of our study is shocking: The DSL implementations in 8 of the 9 language workbenches that use transformations fail to address capture avoidance and produce incorrect code even for minimal changes to the definition of a questionnaire. For example, some implementations fail when a question name is changed to container, questions, or SWTUtils, because these names are implicitly reserved for synthesized variables. Other implementations of the DSL use naming schemes similar to the one we illustrated in the state-machine example. If there is already a question called Q, these implementations fail when naming another question QBlock, calculated_Q, or grp_Q. Some of the variable captures result in compile-time errors of the generated Java code, others result in misbehaved code that, for example, silently skips some of the questions when storing answers persistently. Debugging such errors typically requires investigation of the generated code and can be very time-consuming.

Of the studied DSL implementations, only the transformation implemented in Más addressed variable capture. It uses global name mappings to generate

[1] We studied all workbenches of the previous study [9]: Ensō, Más, MetaEdit+, MPS, Onion, Rascal, Spoofax, SugarJ, the Whole Platform, and Xtext.

unique names from source-language variables for the generated code. The usage of these name mappings and similar approaches is cross-cutting and relies on the discipline of the developer; it is not enforced or supported by the framework. We seek a solution that provides stronger guarantees and has less impact on the implementation of a transformation.

2.2 Problem statement

The goal of this work is to provide a mechanism that avoids variable capture in code that is generated by program transformations. To this end, we seek a mechanism that satisfies the following design goals:

G1: Preserve reference intent: If a reference from the source program occurs in the target program, then the original declaration must also occur in the target program and the reference is still bound by it. In other words, source-program variables may neither be captured by synthesized declarations nor by other source-program declarations.

G2: Preserve declaration extent: If a declaration from the source program occurs in the target program, then only source-program references may be bound by it. In other words, synthesized variable references may not be captured by source-program declarations.

G3: Noninvasive: Avoidance of variable capture should not impact the readability of generated code. This is important in practice, where the generated code is often manually inspected when debugging a program transformation. In particular, a generated program should be left unchanged if it does *not* contain variable capture.

G4: Language-parametric: It should be possible to eliminate variable capture from virtually all source and target languages that feature static name resolution.

G5: Transformation-parametric: The mechanism should work with different transformation engines and should not impose a specific style of transforming programs. Ideally, the mechanism supports existing transformations unchanged.

In the following sections, we present our solution *name-fix*. It fully achieves the first three goals. In addition, *name-fix* is language-parametric provided the name analysis of source and target language satisfy modest assumptions. Finally, *name-fix* works with any transformation engine that provides origin tracking [26] for variable names, so that names originating from the source program can be distinguished from names synthesized by the transformation.

3 Graph-guided elimination of variable capture

The core idea of our solution is to provide a generic mechanism for the detection and elimination of variable capture based on name graphs of the source and target program. We use the term *name* for the string-valued entity that occurs in the abstract syntax tree of a program. Naturally, the same name may occur at

multiple locations of a program. To distinguish different occurrences of the same name, we assume names are labeled with a variable ID. In source programs, such IDs are unique. However, for target programs generated by some transformation, we do not require that variable IDs are unique, because the transformation may have copied and duplicated names from the input program to the output program.

We write x^v to denote that name x is labeled with variable ID v, and we write $p^{@v}$ to retrieve from program p the name corresponding to variable ID v. Nodes that share the same ID must have the same name so that $p^{@v}$ is uniquely determined. The nodes of a name graph are the variable IDs that occur in a program and the edges connect references to the corresponding declarations.

Definition 1. The *name graph* of a program p is a pair $G = (V, \rho)$ where
V is the set of variable IDs in p (references and declarations),
$\rho \in V \rightarrow V$ is a partial function from references to declarations,
and if $\rho(v_r) = v_d$, then reference and declaration have the same name $p^{@v_r} = p^{@v_d}$.

For example, Figure 3 displays the name graph of the state machine in Figure 1(a), where we use line numbers as variable IDs: ID 1 represents the declaration of `opened`, ID 2 represents the reference to `closed` in the transition on line 2, ID 4 represents the declaration of `closed`, and so on.

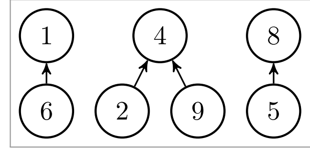


Fig. 3. Name graph of state machine in Figure 1(a).

We require that transformations preserve variable IDs when reusing names from the source program in the generated code. For example, when compiling the state machine of Figure 1(a) to the code in Figure 1(b), the compiler reuses the names of state declarations for the declaration of constant functions and for references to these constant functions in the main dispatch. Accordingly, in the generated code, these names must have the same variable ID as in the source program. Essentially, whenever a transformation copies a name from the source program to the target program, the corresponding ID must be copied as well and thus preserved. In contrast, names that are synthesized by the transformation should have fresh variable IDs.

For example, Figure 4 shows the name graph of the compiled state machine (we left out nodes of function parameters `event` and `state` for clarity). We use line numbers from the source program as variable IDs for reused variables, and *ticked* line numbers of the target program as variable IDs for synthesized variables. In addition, we depict nodes of synthesized variables with a darker background color. We have cycles in the name graph for source nodes 1, 4, and 8 because the transformation duplicated the names at these labels to generate constant functions and references to these constant functions.

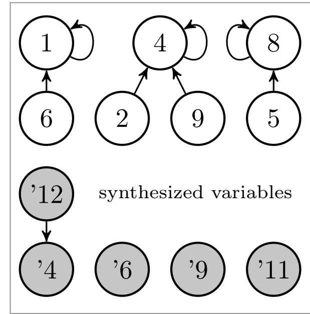


Fig. 4. Names of compiled state machine of Figure 1(b).

One important property of the name graph in Figure 4 is that the source nodes are disconnected from the synthesized nodes, and

all references from the original name graph in Figure 3 have been preserved. In contrast, consider the name graph in Figure 5 that displays result of compilation after renaming state `locked` to `opened-dispatch` as in Figure 2(b). The graph illustrates that a source variable has been captured (dashed arrow) during compilation: The variable at line 5 of the source program was intended to point to the state declared at line 8, but after compilation it points to the dispatch function at line 4 of the synthesized program.

Our solution identifies variable capture by comparing the original name graph of the whole program with the name graph of the generated code. Function *find-capture* in Figure 6 computes the set of edges that witness variable capture. In the state-machine example, *find-capture* finds only one edge $(5 \mapsto {}'4)$ as part of `notPresrvRef1`. We discuss the precise definition of variable capture in the subsequent section.

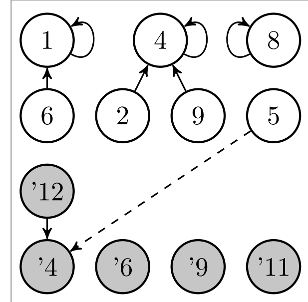


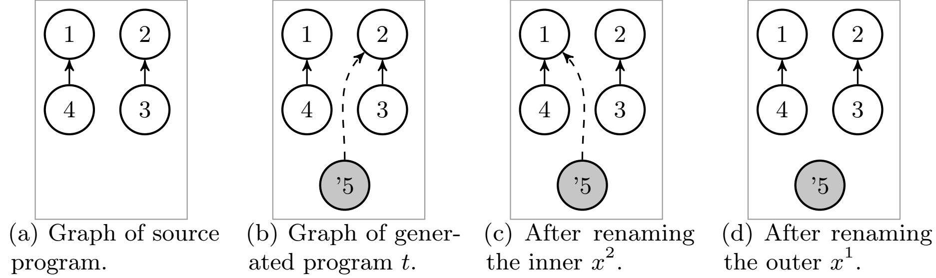
Fig. 5. Variable capture (dashed arrow) in the code of Figure 2(b).

If there are witnesses of variable capture, our solution computes a variable renaming that has two properties. First, for each witness of variable capture, the renaming renames the capturing variable to eliminate the witness. Second, the renaming ensures that intentional references to the capturing variable are renamed as well. This can be difficult because the name graph of the generated code is inaccurate due to variable capture. Therefore, our solution conservatively approximates the set of potential references by including all synthesized variables of the same name. Function *comp-renaming* in Figure 6 computes the renaming as a function from a variable ID to the variable's fresh name, computed by `gensym`. For the example, we get $\pi_{src} = \emptyset$ because $'4 \notin V_s$ and $\pi_{syn} = \{'4 \mapsto$ `"opened-dispatch-0"`, $'12 \mapsto$ `"opened-dispatch-0"`$\}$ because $\mathtt{t}^{@'4} = \mathtt{t}^{@'12}$. Function *rename* in Figure 6 visits all nodes in a syntax tree (represented as s-expression) and applies the renaming π to variables with the corresponding IDs. For the example, the renaming yields a capture-free program with the same name graph as shown in Figure 4.

Function *name-fix* in Figure 6 brings it all together and is the main entry point of our solution. It takes the name graph of the source program and the generated target program as input. First, it computes the name graph of the target program using the function $resolve^T$ that we assume to provide name resolution for the target language T. *name-fix* then calls *find-capture* to identify variable capture. If *find-capture* finds no capturing edges, *name-fix* returns the generated program unchanged. Otherwise, *name-fix* calls *comp-renaming* and *rename* to compute and apply the renaming that eliminates the witnessed variable capture. Since the name graph G_t of `t` may be inaccurate due to variable capture, *name-fix* recursively calls itself to repeat the search for and potential repair of variable capture. Note that *name-fix* applies a closed-world assumption to infer that all unbound variables are indeed free, and thus can be renamed at will.

```
Syntactic conventions:
  x^v          variable x labeled with variable ID v
  p^@v = x     name x that occurs in program p at variable ID v

find-capture((V_s, ρ_s), (V_t, ρ_t)) = {
  notPresrvRef1 = {(v ↦ ρ_t(v)) | v ∈ dom(ρ_t), v ∈ V_s, v ∈ dom(ρ_s), ρ_s(v) ≠ ρ_t(v)};
  notPresrvRef2 = {(v ↦ ρ_t(v)) | v ∈ dom(ρ_t), v ∈ V_s, v ∉ dom(ρ_s), v ≠ ρ_t(v)};
  notPresrvDef  = {(v ↦ ρ_t(v)) | v ∈ dom(ρ_t), v ∉ V_s, ρ_t(v) ∈ V_s};
  return notPresrvRef1 ∪ notPresrvRef2 ∪ notPresrvDef;
}

comp-renaming((V_s, ρ_s), (V_t, ρ_t), t, capture) = {
  π_src = ∅;
  π_syn = ∅;
  foreach v_d in codom(capture) {
    usedNames = {t^@v | v ∈ V_t} ∪ codom(π_src) ∪ codom(π_syn)
    fresh = gensym(t^@v_d, usedNames);
    if (v_d ∈ V_s ∧ v_d ∉ π_src)
      π_src = π_src ∪ {(v_d ↦ fresh)} ∪ {(v_r ↦ fresh) | v_r ∈ dom(ρ_s), ρ_s(v_r) = v_d};
    if (v_d ∉ V_s ∧ v_d ∉ π_syn)
      π_syn = π_syn ∪ {(v ↦ fresh) | v ∈ V_t \ V_s, t^@v = t^@v_d};
  }
  return (π_src, π_syn);
}

rename(t, π) = {
  return t match {
    case x^v if v ∈ dom(π)  =>  π(v)^v
    case x^v                =>  x^v
    case c                  =>  c
    case (t_1...t_n)        =>  (rename(t_1, π) ... rename(t_n, π));
  }
}

name-fix(G_s, t) = {
  G_t = resolve^T(t);

  capture = find-capture(G_s, G_t);
  if (capture == ∅) return t;

  (π_src, π_syn) = comp-renaming(G_s, G_t, t, capture);
  t' = rename(t, π_src ∪ π_syn);
  return name-fix(G_s, t');
}
```

Fig. 6. Definition of *name-fix* that guarantees capture-avoidance.



(a) Graph of source program. (b) Graph of generated program t. (c) After renaming the inner x^2. (d) After renaming the outer x^1.

Fig. 7. Name graphs during execution of *name-fix* for $t = \lambda\, x^1.\, (\lambda\, x^2.\, x^3\, x^{'5})\, x^4$.

In the following, we present examples that illustrate two design choices of *name-fix* that may be somewhat unintuitive: Why are multiple rounds of renaming required, and why do we rename all synthesized variables of the same name. For the former property, consider the lambda expression $t = \lambda\, x^1.\, (\lambda\, x^2.\, x^3\, x^{'5})\, x^4$, where we use superscripts to annotate variable IDs and ticked IDs for synthesized variables. The first graph in Figure 7 shows the original binding structure of the hypothetical source program that t is generated from. The second graph shows the binding structure of t. The synthesized variable $x^{'5}$ is captured by the binding of x^2, which is illegal due to notPresrvDef in *find-capture*. Accordingly, *comp-renaming* initiates a renaming of x^2, also renaming x^3 to preserve the source reference. This yields expression $t' = \lambda\, x^1.\, (\lambda\, \alpha^2.\, \alpha^3\, x^{'5})\, x^4$ with binding structure as shown in the third graph. Indeed, x^2 no longer captures $x^{'5}$. However, now x^1 captures $x^{'5}$. Thus, by renaming x^1 and its reference x^4, we get $t'' = \lambda\, \beta^1.\, (\lambda\, \alpha^2.\, \alpha^3\, x^{'5})\, \beta^4$ with capture-free binding structure as shown in the last graph. The iterative renaming was necessary because the name graph of t did not indicate that $x^{'5}$ is eventually captured by x^1. We could have preemptively renamed x^1 together with x^2, but this contradicts our goal for minimal invasiveness.

To illustrate why *name-fix* renames all synthesized variables of the same name, consider the expression $t = \lambda\, x^{'3}.\, x^1 (\lambda\, x^2.\, x^{'4})$ in which $x^{'3}$ captures $x^{'1}$ and x^2 captures $x^{'4}$. Thus, *name-fix* needs to rename $x^{'3}$ and x^2. Because $x^{'3}$ and $x^{'4}$ are both synthesized and have the same name, renaming of $x^{'3}$ entails the renaming of $x^{'4}$ even though they are unrelated in the name graph of t. Thus, *name-fix* yields the correct result $t' = \lambda\, \alpha^{'3}.\, x^1 (\lambda\, \beta^2.\, \alpha^{'4})$. To see why $x^{'3}$ should bind $x^{'4}$, consider what happens had the source program consistently used y in place of x: $t_2 = \lambda\, x^{'3}.\, y^1 (\lambda\, y^2.\, x^{'4})$. This program has no variable capture and is returned unchanged by *name-fix*. Since we want the result of *name-fix* to be invariant under consistent renamings of the source variables, $x^{'3}$ must bind $x^{'4}$ in both t and t_2. By renaming all synthesized variables of the same name, *name-fix* ensures that no potential variable reference is truncated.

Both of the above examples also illustrate another point: *name-fix* does not guarantee valid name binding with respect to the target language. The final

result in both examples contains a free variable. Instead, *name-fix* guarantees that there is no variable capture. We state and verify the precise properties of *name-fix* in the next section.

4 Termination, correctness, and an equivalence theory

Our solution *name-fix* iteratively eliminates variable capture in a fixed-point computation. In this section we show three important properties of *name-fix*: *name-fix* terminates, *name-fix* eliminates variable capture, and *name-fix* yields α-equivalent outputs for inputs that are equal up to consistent (but possibly capturing) variable renaming.

We represent programs as s-expressions with constant symbols c, labeled variable names x^v, and compound terms $(t_1 \dots t_n)$. We shall frequently require two programs to be equal up to unconditional renaming:

Definition 2. Two programs are *label-equivalent* $p_1 \equiv_{\mathcal{L}} p_2$ iff they are equal up to variable names:

$$
\begin{aligned}
c_1 &\equiv_{\mathcal{L}} c_2 && \text{if } c_1 = c_2 \\
x_1^{v_1} &\equiv_{\mathcal{L}} x_2^{v_2} && \text{if } v_1 = v_2 \\
(t_1 \dots t_n) &\equiv_{\mathcal{L}} (t_1' \dots t_n') && \text{if } t_i \equiv_{\mathcal{L}} t_i' \quad \forall\, 1 \leq i \leq n
\end{aligned}
$$

To simplify our formalization, we do not consider bijective relabeling functions and assume label-equivalence instead. As first metatheoretical result we state that *name-fix* terminates.[2]

Theorem 1. *For any name graph G_s and any program t, name-fix(G_s, t) terminates in finitely many steps.*

4.1 Assumptions on name resolution

We present our framework for capture-avoiding transformations independent of any concrete source and target languages. Since our technique works on top of name graphs, we require functions $resolve^L$ that compute the name graph of a program of some language L by name analysis. However, instead of requiring a specific form of name analysis, we specify minimal requirements on the behavior of $resolve^L$ that suffice to show our technique is sound. The first assumption states that name analysis must produce a name graph.

Assumption 1. *Given a program p, $resolve^L(p)$ yields the name graph $G = (V, \rho)$ of p according to Definition 1.*

The second assumption requires $resolve^L$ to behave deterministically. First, given two programs p_1 and p_2 that are equal up to variable names, names that are references in p_1 must be references in p_2 if the declaration is available (but it can refer to another declaration). Second, given a reference with two potential declarations in p_1 and p_2, $resolve^L$ must deterministically choose one of them.

[2] Proofs of all theorems and additional lemmas appear in Appendix A.

Assumption 2. *Let* $p_1 \equiv_{\mathcal{L}} p_2$ *be label-equivalent with name graphs* $\mathit{resolve}^L(p_1) = (V, \rho_1)$ *and* $\mathit{resolve}^L(p_2) = (V, \rho_2)$*.*

(i) If $\rho_1(v_r) = v_d$ *and* $p_2^{@v_r} = p_2^{@v_d}$*, then* $v_r \in \mathit{dom}(\rho_2)$*.*

(ii) If $\rho_1(v_r) = v_d$*,* $\rho_2(v_r) = v'_d$*,* $p_1^{@v_d} = p_1^{@v'_d}$*, and* $p_2^{@v_d} = p_2^{@v'_d}$*, then* $v_d = v'_d$*.*

In addition to these assumptions, we require that the name graph (V, ρ) of the original source program satisfies $\mathit{dom}(\rho) \cap \mathit{codom}(\rho) = \emptyset$. We call such graphs *bipartite name graphs*. Note that $\mathit{resolve}^L$ often does not produce bipartite name graphs for generated code due to name copying as in Figure 4. We believe our requirements are modest and readily satisfied by name analyses of most languages.

4.2 *name-fix* eliminates variable capture

We define the notion of capture-avoiding transformations in terms of the name graph of the source and target programs, before we show that *name-fix* can turn any transformation into a capture-avoiding one.

Definition 3. A transformation $f : S \to T$ is *capture-avoiding* if for all $s \in S$ with $\mathit{resolve}^S(s) = (V_s, \rho_s)$ and $t = f(s)$ with $\mathit{resolve}^T(t) = (V_t, \rho_t)$:

1. Preservation of reference intent: For all $v \in \mathit{dom}(\rho_t)$ with $v \in V_s$,
   (i) if $v \in \mathit{dom}(\rho_s)$, then $\rho_s(v) = \rho_t(v)$,
   (ii) if $v \notin \mathit{dom}(\rho_s)$, then $v = \rho_t(v)$.
2. Preservation of declaration extent: For all $v \in \mathit{dom}(\rho_t)$, if $v \notin V_s$, then $\rho_t(v) \notin V_s$.

The first condition states that a capture-avoiding transformation must preserve references of the source program. That is, if a variable v occurs in the target program and this reference was bound in the source program, then the target program must provide the same binding for v. That is, the transformation must preserve the reference intent of the source program's author.

If the source program does not contain v as a bound variable (but maybe as a declaration), v can only refer to itself in the target program. We specifically admit such self-references to allow transformations to duplicate names of source-program declarations in order to introduce additional delegation. For example, our compiler for state machines illustrated in Figure 1(a) uses names of state declarations to generate constant functions and references to these functions. Note that we also admit duplication of reference names, each of which has the same variable ID and thus must refer to the original declaration.

The second condition states that a capture-avoiding transformation must keep synthesized variable references separate from variables declared in the source program. We consider all variables of the source program V_s to be original and all variables of the target program that do not come from the source program $(V_t \setminus V_s)$ to be synthesized. This condition prevents synthesized variable references to be captured by original variable declarations, that is, synthesized variables can only be bound by synthesized declarations.

Function *find-capture* in Figure 6 implements the test for capture avoidance and collects witnesses in case of variable capture. Since *name-fix* only terminates when *find-capture* fails to find variable capture, the correctness of *name-fix* follows from its termination.

Theorem 2 (Capture avoidance). Given a transformation $f : S \rightarrow T$, *name-fix* yields a capture-avoiding transformation $\lambda s.\ \textit{name-fix}(\textit{resolve}^S(s), f(s))$.

4.3 Definitions of α-equivalence and sub-α-equivalence

It is not enough to ensure that *name-fix* eliminates variable capture, because, for example, a function that returns the empty program would satisfy this property. To ensure the usefulness of *name-fix*, we need to show that, given two programs that are equal up to possibly capturing renaming, it produces α-equivalent programs (and not just any programs). Two programs are α-equivalent if they are equal up to non-capturing renaming, that is, if they have the same syntactic structure and binding structure.

Definition 4. Two programs p_1 and p_2 with name graphs $\textit{resolve}^L(p_1) = (V_1, \rho_1)$ and $\textit{resolve}^L(p_2) = (V_2, \rho_2)$ are *α-equivalent* $p_1 \equiv_\alpha p_2$ iff $p_1 \equiv_{\mathcal{L}} p_2$ and $\rho_1 = \rho_2$.

Note that $p_1 \equiv_{\mathcal{L}} p_2$ entails $V_1 = V_2$. As expected, our definition of α-equivalence is independent of the concrete names that occur in the programs. The following examples illustrate our definition of α-equivalence.

Program	Name graph
$p_1 = \lambda\, x^1.\ (\lambda\, y^3.\ y^4\ y^5)\ x^2$	$G_1 = (\{1,2,3,4,5\}, \{(2 \mapsto 1), (4 \mapsto 3), (5 \mapsto 3)\})$
$p_2 = \lambda\, x^1.\ (\lambda\, x^3.\ x^4\ x^5)\ x^2$	$G_2 = (\{1,2,3,4,5\}, \{(2 \mapsto 1), (4 \mapsto 3), (5 \mapsto 3)\})$
$p_3 = \lambda\, x^1.\ (\lambda\, y^3.\ x^4 + y^5)\ x^2$	$G_3 = (\{1,2,3,4,5\}, \{(2 \mapsto 1), (4 \mapsto 1), (5 \mapsto 3)\})$
$p_4 = \lambda\, x^1.\ (\lambda\, x^3.\ x^4 + x^5)\ x^2$	$G_4 = (\{1,2,3,4,5\}, \{(2 \mapsto 1), (4 \mapsto 3), (5 \mapsto 3)\})$

Our definition correctly identifies $p_1 \equiv_\alpha p_2$, because they are label-equivalent and have the same name graphs. Indeed, p_2 can be derived from p_1 by consistently renaming all occurrences of the bound variable y to x. In contrast, $p_3 \not\equiv_\alpha p_4$ because the binding structure differs: x^4 is bound to x^1 in p_3, but to x^3 in p_4. All other combinations of above programs (modulo symmetry of $\equiv_\alpha$) are not α-equivalent because they fail the required label-equivalence. In particular, $p_2 \not\equiv_\alpha p_4$ in spite of having the same binding structure.

To relate programs that are equal up to possibly capturing renaming, we propose the following notion of sub-α-equivalence.

Definition 5. Two programs are *sub-α-equivalent* $p_1 \equiv_\alpha^G p_2$ under a name graph $G = (V, \rho)$ iff $p_1 \equiv_{\mathcal{L}} p_2$ and, given V_p is the set of labels in p_1 and p_2,

(i) for all $v_r, v_d \in V_p \cap V$ with $\rho(v_r) = v_d$, $\quad p_1^{@v_r} = p_1^{@v_d} \Leftrightarrow p_2^{@v_r} = p_2^{@v_d}$
(ii) for all $v_r, v_d \in V_p \setminus V$, $\quad p_1^{@v_r} = p_1^{@v_d} \Leftrightarrow p_2^{@v_r} = p_2^{@v_d}$

Sub-α-equivalence compares two programs based on the actual names occurring in them, and not based on the binding structure. The relation is parameterized over a name graph G. The first condition states that for each binding in this graph, p_1 and p_2 need to agree on whether reference and declaration share the same name or not. Even if the reference and declaration have the same name, it does not imply that there is a corresponding binding in either p_1 or p_2, because another declaration can also have this name and capture the reference. The second condition states that for all variables not in G, p_1 and p_2 need to agree on which variable occurrences share names. To illustrate sub-α-equivalence, let us consider $G = (\{1, 2, 3\}, \{(2 \mapsto 1), (3 \mapsto 1)\})$ and the following programs:

$[p_1]_{\equiv_\alpha^G}$	$p_1 = \lambda x^1.\ (\lambda y^{'4}.\ x^3 + y^{'5})\ x^2$ $p_3 = \lambda x^1.\ (\lambda z^{'4}.\ x^3 + z^{'5})\ x^2$	$p_2 = \lambda z^1.\ (\lambda y^{'4}.\ z^3 + y^{'5})\ z^2$ $p_4 = \lambda z^1.\ (\lambda z^{'4}.\ z^3 + z^{'5})\ z^2$
$\neg[p_1]_{\equiv_\alpha^G}$	$p_5 = \lambda \underline{z}^1.\ (\lambda y^{'4}.\ x^3 + y^{'5})\ x^2$ $p_7 = \lambda x^1.\ (\lambda \underline{z}^{'4}.\ x^3 + y^{'5})\ x^2$	$p_6 = \lambda x^1.\ (\lambda y^{'4}.\ \underline{z}^3 + y^{'5})\ x^2$ $p_8 = \lambda x^1.\ (\lambda y^{'4}.\ x^3 + \underline{z}^{'5})\ x^2$

The first four programs are sub-α-equivalent to p_1 under G. We have $p_1 \equiv_\alpha^G p_2$ because they agree on the name sharing at variable IDs 1, 2, and 3, which is required because of the bindings in G, and on the name sharing at variable IDs '4 and '5, which is required because these IDs are not in G. Similar analysis shows $p_1 \equiv_\alpha^G p_3$ and $p_1 \equiv_\alpha^G p_4$. Programs p_5 through p_8 are examples that are not sub-α-equivalent to p_1 under G. For p_5 and p_6 the first condition of sub-α-equivalence fails because there is no agreement on the name sharing at 1 and 3. For p_7 and p_8 the second condition fails because there is no agreement on the name sharing at '4 and '5.

Note that $p_1 \equiv_\alpha^G p_4$ illustrates that sub-α-equivalence is weaker than α-equivalence because $p_1 \not\equiv_\alpha p_4$. In the following subsection we use sub-α-equivalence to characterize programs that *name-fix* can repair to α-equivalent programs.

4.4 An equivalence theory for *name-fix*

We now turn to one of the main results of our metatheory: Function *name-fix* is noninvasive, preserves sub-α-equivalence, and is invariant under consistent (but possibly capturing) renaming of original and synthesized variables, as specified by sub-α-equivalence.

For capture-free programs, *name-fix* yields the input program unchanged, that is, *name-fix* is noninvasive:

Theorem 3. For any name graph $G_s = (V_s, \rho_s)$ and any program t with $\textit{find-capture}(G_s, \textit{resolve}^T(t)) = \emptyset$, $\textit{name-fix}(G_s, t) = t$.

Given a bipartite name graph of the source program, *name-fix* preserves sub-α-equivalence:

Theorem 4. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and any program t, $\textit{name-fix}(G_s, t) \equiv_\alpha^{G_s} t$.*

Given a bipartite name graph of the source program, *name-fix* maps sub-α-equivalent programs to α-equivalent ones:

Theorem 5. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and programs $t_1 \equiv_\alpha^{G_s} t_2$, name-fix$(G_s, t_1) \equiv_\alpha$ name-fix(G_s, t_2).*

5 Hygienic transformations

In the previous section, we demonstrated that for any transformation $f : S \to T$, *name-fix* provides a capture-avoiding transformation $\lambda\, s.\textit{name-fix}(G_s, f(s))$. However, for some transformations *name-fix* yields a transformation that adheres to the stronger property of hygienic transformations.

Definition 6. A transformation $f : S \to T$ is *hygienic* if it maps α-equivalent source programs to α-equivalent target programs:

$$s_1 \equiv_\alpha s_2 \implies f(s_1) \equiv_\alpha f(s_2).$$

This definition of hygiene for transformations follows Herman's definition of hygiene for syntax macros [10].

Transformations can inspect the names of variables and can generate structurally different code for α-equivalent inputs. For example, a transformation may decide to produce thread-safe accessors for variables with names prefixed by sync_. Accordingly, a consistent renaming from sync_foo to foo in the source program leads to generated programs that are not structurally equivalent, let alone α-equivalent. However, there is an interesting class of transformations for which *name-fix* provides hygiene:

Definition 7. A transformation $f : S \to T$ is *sub-hygienic* if it maps α-equivalent source programs $s_1 \equiv_\alpha s_2$ to sub-α-equivalent target programs $f(s_1) \equiv_\alpha^{G_s} f(s_2)$ under the name graph G_s of s_1 (or s_2).

The class of sub-hygienic transformations includes some common transformation schemes. First, it includes transformations that transform a source program solely based on the program's structure but independent of the concrete variable names occurring in it. In such transformations, synthesized variable names are constant and the same for any source program. Second, for a source language without name shadowing (such as state machines), sub-hygienic transformations include those that derive synthesized variable names using an injective function $g : \mathsf{string} \to \mathsf{string}$ over the corresponding source variable names. For example, in Figure 1, we derived the name of a dispatch function by appending -dispatch to the corresponding state name. In both cases *name-fix* eliminates all potential variable capture and yields a fully hygienic transformation:

Theorem 6. *For any sub-hygienic transformation $f : S \to T$, transformation $\lambda\, s.\textit{name-fix}(G_s, f(s))$ is hygienic.*

```
fun zero() = 0;
fun succ(x) = let n = 1 in x + n;
let n = x + 5 in
  succ(succ(n + x + zero()))
```

(a) Program with free variable x.

```
fun zero() = 0;
fun succ(x) = let n = 1 in (x + n);
let n0 = 2*n + 5 in
  succ(succ(n0 + 2*n + zero()))
```

(b) Result of substituting 2*n for x.

Fig. 8. *name-fix* yields a capture-avoiding substitution that renames local variables.

6 Case studies

To evaluate the applicability of capture-avoiding program transformation in practice, we have successfully applied *name-fix* in three different scenarios:

- Optimization: Function inlining via substitution in a procedural language.
- Desugaring of language extensions: Lambda lifting of local functions.
- Code generation: Compilation of state machines and of Derric, an existing DSL for digital forensics, to Java.

We have implemented all case-studies in Rascal, a programming language and environment for source code analysis and transformation [13]. The source code of our implementation and all case studies are available online: `http://github.com/seba--/hygienic-transformations`.

6.1 Preservation of variable IDs with string origins in Rascal

As described in Section 3, a transformation must preserve variable IDs of the source program when reusing these names in the target program. While it is possible for a developer of a program transformation to manually preserve variable IDs via copying, it is easier and safer if the transformation engine does it automatically. We extended Rascal to preserve variable IDs automatically via a new Rascal feature called *string origins* [24]. Every string value (captured by the **str** data type) carries information about its origin. A string can either originate from a parsed text file, from a string literal in a metaprogram, or from a string computation such as concatenation, slicing, or substitution.

String origins allow us to obtain precise offsets and lengths for known substrings (e.g., names) so that it is possible to replace substrings. We use this feature to support *name-fix* for transformations that produce a target program as a string instead of an abstract syntax tree. Despite the higher fragility of string-based transformations, they are common in practice. In our case studies, we use string-based transformations to generate Java code.

6.2 Capture-avoiding substitution and inlining

Substitution and inlining are program transformations that may introduce variable capture. Using *name-fix*, the definition of capture-avoiding versions of these

transformations becomes straight-forward because *name-fix* takes over the responsibility for avoiding variable capture. Figure 8 illustrates the application of capture-avoiding substitution to a program of a simple language with global first-order functions and local *let*-bound variables. In the example, we use substitution to replace free occurrences of variable x by 2*n. To prevent capture, our capture-avoiding substitution function renames the locally bound variable n.

Substitution is a program transformation where the source and the target language coincide. Capture-avoiding substitution must retain the binding structure of the original (source) program. Since this requirement is part of our definition of capture-avoiding transformations, we can use *name-fix* to get a capture-avoiding substitution function from a capturing substitution function. This simplifies the definition of substitution for our procedural language to the following:

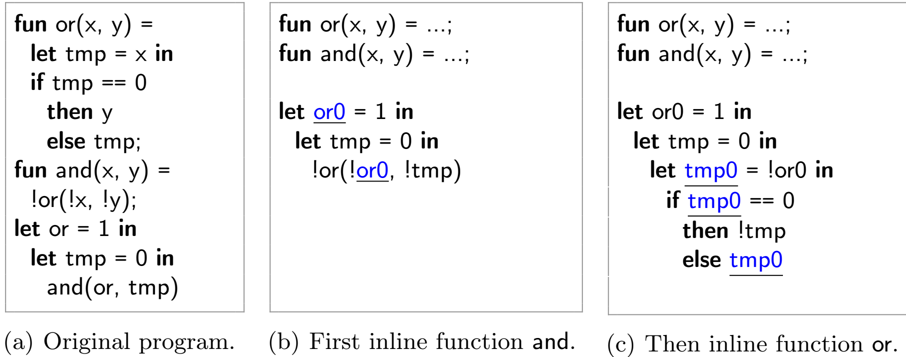
```
subst(p, x, e) = name-fix(resolve(p), substP(p, x, e));
substP(p, x, e) = prog([substF(f, x, e) | f ← p.fdefs], [substE(e2, x, e) | e2 ← p.main]);
substF(fdef(f, params, b), x, e) = fdef(f, params, x in params ? b : substE(b, x, e));

substE(var(y), x, e)          = x == y ? e : var(y);
substE(let(y, e1, e2), x, e) = let(y,  substE(e1, x, e),  x == y ? e2 : substE(e2, x, e));
substE(e1, x, e)              = for (Exp e2 ← e1) insert substE(e2, x, e);
```

Function substP takes a program p and substitutes e for x in all function definitions and expressions of the main routine using substF and substE, respectively. Function substF substitutes e for x in the body of a function only if x does not occur as parameter name of the function, that is, only if x is indeed free in the function body. Function substE proceeds similarly for *let*-bound variables. The final case of substE uses Rascal's generic-programming features [13] to provide a default implementation: We substitute e for x in each direct subexpression of e1 and insert the corresponding result in place of the subexpression.

Function subst ensures capture avoidance, but function substP does not: When pushing expression e under a binder, the bound variable may occur free in e, in which case the bound variable should be renamed. By using *name-fix*, we can omit checking and potentially renaming the bound variable both for function definitions and for *let* expressions and still get a capture-avoiding substitution function subst that behaves as illustrated in Figure 8.

Inlining of functions is a common program-optimization technique used by compilers. We illustrate our implementation of capture-avoiding inlining in Figure 9. The left column shows a simple program using two logical functions or and and. The central column shows the program after inlining and. Note that our language uses a single namespace for functions and *let*-bound variables. We avoid capture of the reference to or by renaming the local variable or to or0. The right column shows the result of inlining or in the central program. The local variable tmp in the definition of or is renamed to tmp0 since otherwise it would capture the reference to the variable tmp of the main body.

Based on our implementation of substitution, we can easily implement inlining by calling substE to substitute all arguments of a function call into the body of the function. Like for substitution, it suffices to call *name-fix* *after* function



(a) Original program. (b) First inline function and. (c) Then inline function or.

Fig. 9. Capture-avoiding function inlining is similar to hygienic macro expansion.

inlining is complete. Intuitively, this is because *name-fix* only renames bound variables, which are ignored by substE anyway. A detailed investigation of *when* to call *name-fix* is part of our future work.

6.3 Lambda lifting

Language extensions augment a base language with additional language features. Many compilers first *desugar* a source program to a core language. Extensible languages like SugarJ [8] enable regular programmers to define their own extensions via custom desugaring transformations. Such desugaring transformations should preserve the binding structure of the source program. In fact, the lack of capture-avoiding and hygienic transformations in extensible languages was a major motivation of this work.

Exemplary, to show that *name-fix* supports language extensions, we implemented an extension of our procedural language for local function definitions that we desugar by lifting them into the global toplevel function scope [12]. The left column of Figure 10 shows an example usage of the extension, where we have a global function f that is shadowed by a local function f, which is used in another local function g. When lifting the two local functions, we get two toplevel functions named f, where the originally local f captures a call to the originally global f in the definition of y. Accordingly, *name-fix* renames the lifted function f and its calls, both in the main program and the lifted version of g.

We implement lambda lifting by recursively (i) finding local functions, (ii) adapting calls to the local function to pass along variables that occur free in the function body, and (iii) lifting the function definition to the toplevel. To identify calls of a local function, we use the name graph of the non-lifted program. A single call to *name-fix* after desugaring suffices to eliminate potential name shadowing between functions in the toplevel function scope.

```
fun f(x) = x + 1;
let y = f(10) in
  let fun f(x) = f(x + y) in
    let fun g(x) = f(y + x + 1) in
      f(1) + g(3)
```

(a) Example with local functions f and g.

```
fun f(x) = x + 1;
fun f0(x, y) = f0(x + y, y);
fun g(x, y) = f0(y + x + 1, y);
let y = f(10) in
  f0(1, y) + g(3, y)
```

(b) Desugaring of local functions.

Fig. 10. Lambda lifting of local functions f and g requires renaming to avoid capture.

```
list[FDef] compile(list[State] states) =
  map(state2const, states) + map(state2dispatch, states) + mainDispatch(states)

FDef state2const(State s, int i) =
  fdef(s.name, [], val(nat(i)));
FDef state2dispatch(State s) =
  fdef("<s.name>-dispatch", ["event"], transitions2cond(s.transitions, val(error())));
Exp transitions2cond([t, *ts], Exp els) =
  cond(equ(var("event"), val(string(t.event)))
       , call(t.state, [])
       , transitions2cond(ts, els));
FDef mainDispatch(states) =
  fdef("main", ["state","event"], mainCond(states, val(error())))
Exp mainCond([s, *ss], Exp els) =
  cond(equ(var("state"), call(s.name, []))
       , call("<s.name>-dispatch", [var("event")])
       , mainCond (ss, els));
```

Fig. 11. Implementation of compiler from state machines to our procedural language.

6.4 State machines

In Section 1, we introduced a language for state machines to illustrate the problem of inadvertent capture in program transformation. The *name-fix* algorithm can be used to repair the result of the transformation without changing the transformation itself. As a result, developers can structure transformations in almost arbitrary ways. In the case of the state-machine compiler, a simple naming convention suffices to link generated references to declarations. In our case study, the conventions are that state names become constants and state names suffixed with -dispatch become dispatch functions.

We believe the increased liberty of using naming conventions simplifies the implementation of program transformations. We illustrate the main part of the compiler of state machines to our procedural language in Figure 11. In contrast to approaches based on explicit binders such as HOAS [18] or FreshML [22], generated references do not have to literally occur below their binders in the transformation itself. For example, function compile independently generates state

```
state current
  close => closed
end

state closed
  open => current
  lock => token
end

state token
  unlock => closed
end
```

(a) Renamed door state machine.

```
public class Door {
  final int current = 0, closed = 1, token = 2;
  void run(...) {
    int current0 = current;  String token0 = null;
    while ((token0 = input.nextLine()) != null) {
      if (current0 == current)
       {if (close(token0)) current0 = closed; else continue;}
      if (current0 == closed)
       {if (open(token0)) current0 = current;
        else if (lock(token0)) current0 = token; else continue;}
      if (current0 == token)
       {if (unlock(token0)) current0 = closed; else continue;}
}}}
```

(b) Renaming of local variables current and token to preserve the references of the state machine (exemplarily highlighted).

Fig. 12. Application of *name-fix* for generated Java code with JDT name resolution.

constants, state dispatch functions, and the main dispatch function (by mainCond), even though the main dispatch function refers to both generated constants and state dispatch functions via naming conventions.

Compilation to Java. To exercise capture-avoiding transformation in a more realistic setting, we also applied *name-fix* on the result of compiling state machines to Java. To obtain a name graph for Java, we used Rascal's M^3 source code model of Java, which provides accurate name and type information extracted from the Eclipse JDT [11]. The compiler from state machines to Java generates Java code as structural strings (cf. Section 6.1). It generates a constant for each state and a single dispatch loop in a run method.

We illustrate the application of the compiler and the use of *name-fix* on the generated Java code in Figure 12. The left column shows the state machine from Figure 1(a) where we consistently renamed states opened and locked to current and token, respectively. The right column shows the compiled Java program. Since the dispatch loop in run uses current to store the current state and token to save the last-read token, the compilation introduces variable capture. Note that even without using *name-fix*, the generated code compiles fine but is ill-behaved because current==current in the first *if* would always succeed. *name-fix* repairs the variable capture by renaming the local variables. This case study shows that *name-fix* and our implementation are not limited to simple languages, but are applicable for generating capture-free programs of languages like Java.

6.5 Digital forensics with Derric

Derric is a domain-specific language for describing (binary) file formats [25]. Such descriptions are used in digital forensic investigations to recover evidence

```
format Bad

sequence S1 S2

structures
S1 { x: 0x0;  y: S2.x; }
S2 { x; }
```

```
public class Bad {
  private long x;
  private boolean S1() {
    markStart();
    long x0 = ...;  ValueSet vs2 = ...;
    vs2.addEquals(0);
    if (!vs2.equals(x0)) return noMatch();
    long y = ...;  ValueSet vs5 = ...;
    vs5.addEquals(x);
    if (!vs5.equals(y)) return noMatch();
    addSubSequence("S1");
    return true;
}...}
```

(a) A Derric format. (b) The local variable shadows the field and must be renamed.

Fig. 13. *name-fix* eliminates variable capture for existing DSL compiler of Derric.

from (possibly damaged) storage devices. Derric descriptions consist of two parts. The first part describes the high-level structure of a file format by listing sequence constraints on basic building blocks (called structures) of a file. The second part describes each structure by declaring fields, their type, and inter-structure data dependencies. From these descriptions, the Derric compiler generates high-performance validators in Java that check whether a byte sequence matches the declared format.

We show a minimalist, artificial Derric format description in the left column of Figure 13. The format declares two structures (S1 and S2), which must occur in sequence. S1 contains two fields: x, which must be 0, and y, which should be equal to field x of S2, which is not further constrained. We show an excerpt of the code generated by the Derric compiler in the right column of Figure 13. The main issue is in method S1, which handles format recognition of structure S1. Field x, which Derric uses to communicate S2's field x to method S1 is shadowed by the local variable x which corresponds to S1's field x. Without going into too much detail, it is instructive to note that the Java code compiles fine even without any renaming, but it behaves incorrectly: Instead of checking S1.y = S2.x, it checks S1.y = S1.x. Such scenario occurs whenever two structures have a field of the same name and one structure access this field of the other structure in a constraint. *name-fix* restores correctness by consistently renaming the local variable in case of capture.

The Derric case study illustrates the flexibility and power of *name-fix*. Derric is a real-world DSL compiling to a mainstream programming language (Java). The compiler consists of multiple transformations for desugaring and optimization. The result of these transformations is an intermediate model of a validator, which is then pretty printed to Java. Nevertheless, we did not have to modify the Derric compiler in any significant way to be able to repair

inadvertent captures, nor was the compiler designed with *name-fix* in mind. This is shows that our approach is readily applicable in realistic settings.

7 Discussion

We reflect on the problem statement of this work, explain how *name-fix* supports breaking hygiene, and point out open issues and future work.

Problem statement. In section 2.2, we postulated five design goals for *name-fix*, all of which it satisfies. In Section 4, we have verified that *name-fix* preserves reference intent (G1) and declaration extent (G2) of the source program. Moreover, we have established an equivalence theory for *name-fix* that at least supports noninvasiveness (G3). In the previous section, we have shown how *name-fix* can be applied in a wide range of scenarios using different languages: state machines, a simple procedural language, DERRIC, and Java. These results support our claim that capture elimination with *name-fix* is language-parametric (G4).

Although the case studies are all implemented in Rascal, any transformation engine that propagates the unique labels of names is suited for *name-fix*. Similar to our encoding, one could easily imagine representing names as tagged strings Name = (String,Int). A structural representation of strings or compound identifiers are not necessary. Moreover, we do not require that transformations are written in any specific style to support capture elimination. In particular, our transformations make use of sophisticated language features such as intermediate open terms or generic programming. We conclude that a mechanism like *name-fix* is transformation-parametric and realizable in other transformation engines (G5).

Breaking hygiene. Some transformations require that source programs refer to names synthesized by a transformation. Such breaking of hygiene often occurs with implicitly declared variables. In other words, intended capture implies that there is a source reference that is not bound by a declaration in the source program. Consider, *anaphoric conditionals* which are like normal *if*-expressions but allow reference to the result of the condition using a special variable it [1]. For instance, in the expression **aif** c **then** !it **else** it, the variable it implicitly refers to a local variable generated by the desugaring of **aif**. Applying *name-fix*, however, resolves the capture which in this case is intended: **let** it0 = c **in if** it0 **then** !it **else** it. To break hygiene in such cases, the transformation must mark the source occurrences of it when they are carried over to the result: aif(c, t, e) $\Rightarrow$ let("it", c, cond(var("it"), *mark*("it", t), *mark*("it", e))). In our implementation, $mark(s, t)$ sets a synthesized=true attribute on the ID of any string s in t. Effectively this means that such names are treated as synthesized names instead of source names. As a result, *name-fix* does not rename the binder, and the result of desugaring the above expression will be **let** it = c **in if** it **then** !it **else** it.

Future work. Theorem 6 shows that *name-fix* turns sub-hygienic transformations into hygienic transformations. However, there is currently no decision procedure for whether a transformation is sub-hygienic or not. For a Turing-complete metalanguage, a static analysis can only approximate this property. Nevertheless,

a conservative analysis would be useful as it can *guarantee* that a transformation is sub-hygienic. For example, all transformations of our case studies except substitution are sub-hygienic, but we have not formally ensured that. We expect a type system that checks sub-hygiene to provide guidance to transformation developers similar to FreshML [22], but without reducing the flexibility.

Another open issue is *when* to apply *name-fix*. This is important when building transformations on top of other transformations or composing transformations sequentially into transformation pipelines. After every application of a transformation, there could be inadvertent variable capture that *name-fix* can eliminate. For our case studies we used informal reasoning to decide whether the call to *name-fix* can be delayed, but more principled guidance would be useful. For example, a simple class of transformations that commutes with applications of name-fix is the class of name-insensitive transformations, such as constant propagation. More generally, care has to be taken whenever a transformation compares two names for equality, because intermediate variable capture may yield inaccurate equalities. Since name-fix is the identity on capture-free programs (Theorem 3), applying name-fix more than necessary is at most inefficient, but not unsafe.

name-fix renames not only synthesized names but also names that originate from the source program. This may break the expected interface of the generated code. Accordingly, *name-fix* currently is a whole-program transformation that does not support linking of generated programs against previously generated libraries, because names in these libraries cannot be changed. Therefore, *name-fix* is currently ill-suited for separate compilation. We have experienced this problem in the Derric compiler, where a Derric field named `BIG_ENDIAN` will shadow a constant with the same name that occurs in Derric's precompiled run-time system. We leave the investigation of a modular *name-fix* for future work.

Finally, the current implementation of *name-fix* requires repeated execution of the name analysis of the target language. As a result, *name-fix* can be expensive in terms of run-time performance. When a compiler is run continuously in an IDE, this penalty can be an impediment to usability. Fortunately, incremental name analysis is a well-studied topic (e.g., [19,27]) that is likely to yield benefits for *name-fix* because (i) we know the delta induced by *name-fix* (renamed variables) and (ii) new variable capture can only occur in references that have changed.

8 Related work

Various approaches to ensuring capture avoidance have been studied in previous work. Many of them represent a program not as a syntax tree, but use the syntax tree as a spanning tree for a graph-based program representation with additional links from variable references to the corresponding variable declarations. The advantage of graph-based representations is that variable references are unambiguously resolved at all times, which can guide developers of transformations. For example, nameless program representations such as de Bruijn indices [5] encode the graph structure of variable bindings via numeric values; Oliveira and

Löh directly encode recursion and sharing in the abstract syntax of embedded DSLs [16] via structured graphs. The disadvantage of these techniques is that they require explicit handling of graphs (updating indices, redirecting edges) and do not support open terms well.

In higher-order abstract syntax (HOAS) [18] variable references and declarations are encoded using the binding constructs of the metalanguage. Thus, developers of transformations inherit name analysis and capture-avoiding substitution from the metalanguage and work with fully name-resolved terms. It is well-known that HOAS has a number of practical problems [21]. For instance, the use of metalevel functions to encode binders makes them opaque; it is not possible to represent open terms or to pattern match against variable binders inside constructs such as let.

FreshML [22] uses types to describe the binding structure of object-language variable binders. This enables deconstruction of a variable binder via pattern matching, which yields a fresh name and the body as an open term in which the bound variable has been renamed to the fresh one. Due to using fresh variables, accidental variable capture cannot occur but intentional variable capture is possible. FreshML is limited by using types for declaring variable scope, because this is only possible for "declare-before-use" lexical scoping and not, for example, for the scoping of methods in an object-oriented class.

In model-driven engineering it is common to describe abstract syntax using class-based metamodels [17]. Syntactic categories correspond to classes, parent-child relations and cross-references are encoded using associations. Metamodels are expressive enough to model programs with each name resolved to its declaration using direct references (pointers). As a result, a large class of model-transformation formalisms are based on graph rewriting [4]. However, we are unaware of any work in this area that addresses capture avoidance. Especially, in the case of model-to-text (M2T) transformations, names have to be output and all guarantees about capture avoidance (if any) are lost.

Seminal work on hygiene has been performed in the context of syntax macros [14,3]. Like *name-fix*, hygienic macro expansion automatically renames bound variables to avoid variable capture. In related work, a number of approaches to hygienic macro expansion have been proposed [2,3,7,10]. Closest to our work is the expansion algorithm proposed by Dybvig, Hieb, and Bruggeman [7] in that they also associate additional contextual information to identifiers in syntax objects, similar to our string origins. However, in their work renamings appear during macro expansion (modulo lazy evaluation), whereas we perform renamings after transformation. Moreover, since for macros the role of an identifier only becomes apparent after macro expansion, they have to track alternative interpretations for a single identifier. In contrast, we require name analysis for the source language, which enables a completely different approach to hygienic transformations.

Marco [15] is a language-agnostic macro engine that detects variable capture by parsing error messages produced by an off-the-shelve compiler of the base language. Marco checks whether any of the free names introduced by a macro is

captured at a call-site of the macro. While Marco does not require name analysis, it has to rely on the quality of error messages of the base compiler, provides no safety guarantees, and can only detect but not fix variable capture.

Generation environments [23] are metalanguage values that allow the scoping of variable names generated by a program transformation. A program transformation can open a generation environment to generate code relative to the encapsulated lexical context. Since generation environments can be passed around as metalanguage values, different transformations can produce code for a shared a lexical context. While generation environments simplify the implementation of transformations, they rely on the discipline of developers and do not provide static guarantees.

Another area where capture avoidance is important is rename refactorings. In particular, previous work on rename refactoring for Java [20] omits checking preconditions and instead tries to fix the result of a renaming through qualified names so that reference intent is preserved. De Jonge et al. generalize this approach to support name-binding preservation in refactorings for other languages [6]. In contrast to our work, rename refactorings are a limited class of transformations that do not introduce any synthesized names.

9 Conclusion

We presented *name-fix*, a generic solution for eliminating variable capture from the result of program transformations by comparing name graphs of the transformation's input and output. This work brings benefits of hygienic macros to the domain of program transformations. In particular, *name-fix* relieves developers of transformations from manually ensuring capture avoidance, and it enables the safe usage of simple naming conventions. We have verified that *name-fix* terminates, is correct, and yields α-equivalent programs for inputs that are equal up to possibly capturing renaming. As we demonstrated with case studies on program optimization, language extension, and DSL compilation, *name-fix* is applicable to a wide range of program transformations and languages.

Acknowledgement. We thank Mitchel Wand, Paolo Giarrusso, Justin Pombrio, Atze van der Ploeg, and the anonymous reviewers for helpful feedback.

References

1. E. Barzilay, R. Culpepper, and M. Flatt. Keeping it clean with syntax parameters. In *Scheme*, 2011.
2. A. Bawden and J. Rees. Syntactic closures. In *LFP*, pages 86–95. ACM, 1988.
3. W. Clinger and J. Rees. Macros that work. In *POPL*, pages 155–162. ACM, 1991.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
5. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

6. M. de Jonge and E. Visser. A language generic solution for name binding preservation in refactorings. In *LDTA*. ACM, 2012.
7. R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.
8. S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.
9. 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 *SLE*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.
10. D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, Massachusetts, 2012.
11. A. Izmaylova, P. Klint, A. Shahi, and J. Vinju. M^3: An open model for measuring source code artifacts. arXiv:1312.1188, 2013. BENEVOL'13.
12. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA)*, pages 190–203. Springer, 1985.
13. P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.
14. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP*, pages 151–161. ACM, 1986.
15. B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley. Marco: Safe, expressive macros for any language. In *ECOOP*, volume 7313 of *LNCS*, pages 589–613. Springer, 2012.
16. B. C. d. S. Oliveira and A. Löh. Abstract syntax graphs for domain specific languages. In *PEPM*, pages 87–96. ACM, 2013.
17. R. F. Paige, D. S. Kolovos, and F. A. C. Polack. Metamodelling for grammarware researchers. In *SLE*, volume 7745 of *LNCS*, pages 64–82. Springer, 2012.
18. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988.
19. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *TOPLAS*, 5(3):449–477, 1983.
20. M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA*, pages 227–294. ACM, 2008.
21. T. Sheard. Accomplishments and research challenges in meta-programming. In *SAIG*, pages 2–44. Springer, 2001.
22. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ICFP*, pages 263–274. ACM, 2003.
23. Y. Smaragdakis and D. S. Batory. Scoping constructs for software generators. In *GCSE*, volume 1799 of *LNCS*, pages 65–78. Springer, 1999.
24. P. I. Valdera, T. van der Storm, and S. Erdweg. Tracing model transformations with string origins. In *ICMT*. Springer, 2014. to appear.
25. J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *ICSE*, pages 671–680. ACM, 2011.
26. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Symbolic Computation*, 15:523–545, 1993.
27. G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In *SLE*, volume 8225 of *LNCS*, pages 260–280. Springer, 2013.

Appendix

A Proofs of theorems from Section 4 and Section 5

Theorem 1. *For any name graph G_s and any program t, name-fix(G_s, t) terminates in finitely many steps.*

Proof. The depth of the recursion of *name-fix* is bound by the number of variable declarations in t. Each variable declaration v_d can at most occur once in the result of *find-capture* because it is immediately renamed to a fresh name. The renamed variable declaration cannot occur in *find-capture* again because (i) if $v_d \in V_s$, then only references $v_r \in V_s$ with $\rho(v_r) = v_d$ share the fresh name and (ii) if $v_d \notin V_s$, then only references $v_r \in V_t \setminus V_s$ share the fresh name but $v_r \in dom(\textit{find-capture})$ entails $\textit{find-capture}(v_r) \in V_s$. Hence *name-fix* terminates after at most all variable declarations in t have been renamed once. □

Theorem 2 (Capture avoidance). Given a transformation $f : S \to T$, *name-fix* yields a capture-avoiding transformation $\lambda s.\ \textit{name-fix}(\textit{resolve}^S(s), f(s))$.

Proof. When *name-fix* terminates, $\textit{find-capture} = \emptyset$ and thus all reference intent and declaration extent is preserved from the name graph of s to the name graph of the resulting program. □

Lemma 1. *For any graph G, sub-α-equivalence under G is an equivalence relation, that is, it is reflexive, symmetric, and transitive.*

Proof. Follows directly from the definition of sub-α-equivalence and the fact that $\equiv_{\mathcal{L}}$ is an equivalence relation. □

Theorem 3. For any name graph $G_s = (V_s, \rho_s)$ and any program t with $\textit{find-capture}(G_s, \textit{resolve}^T(t)) = \emptyset$, $\textit{name-fix}(G_s, t) = t$.

Proof. By definition of *name-fix*. □

Lemma 2. *For any renaming π and program t, $\textit{rename}(t, \pi) \equiv_{\mathcal{L}} t$.*

Proof. By induction on the structure of t □

Lemma 3. *For any name graph $G_s = (V_s, \rho_s)$ and sub-α-equivalent programs $t_1 \equiv_\alpha^{G_s} t_2$ under name graph G_s, if $\textit{find-capture}(G_s, \textit{resolve}^T(t_1)) = \emptyset$ and $\textit{find-capture}(G_s, \textit{resolve}^T(t_2)) = \emptyset$, then $t_1 \equiv_\alpha t_2$.*

Proof. Let $(V_i, \rho_i) = \textit{resolve}^T(t_i)$ and $\mathsf{capture}_i = \textit{find-capture}(G_s, \textit{resolve}^T(t_i))$. By definition $t_1 \equiv_\alpha t_2$ if $\rho_1 = \rho_2$, which holds if $dom(\rho_1) = dom(\rho_2)$ and $\rho_1(v) = \rho_2(v)$ for all $v \in dom(\rho_1)$. Let $v \in dom(\rho_1)$ (analogously for $v \in dom(\rho_2)$). We distinguish 3 cases:

1. If $v \in V_s$ and $v \in dom(\rho_s)$, then $\rho_1(v) = \rho_s(v)$ because otherwise $\rho_s(v) \neq \rho_1(v)$ entails $(v \mapsto \rho_1(v)) \in \mathsf{notPresrvRef1} \subseteq \mathsf{capture}_1$, contradicting $\mathsf{capture}_1 = \emptyset$. By Assumption 1, $t_1^{@v} = t_1^{@\rho_1(v)}$, which implies $t_2^{@v} = t_2^{@\rho_1(v)}$ by the first condition of $t_1 \equiv_\alpha^{G_s} t_2$. Thus by Assumption 2-(i), $v \in dom(\rho_2)$. Then $\rho_2(v) = \rho_s(v)$ because otherwise $(v \mapsto \rho_2(v)) \in \mathsf{notPresrvRef1} \subseteq \mathsf{capture}_2$, contradicting $\mathsf{capture}_2 = \emptyset$. By Assumption 2-(ii), $\rho_1(v) = \rho_2(v)$.
2. If $v \in V_s$ and $v \notin dom(\rho_s)$, then $\rho_1(v) = v$, because otherwise $(v \mapsto \rho_1(v)) \in \mathsf{notPresrvRef2} \subseteq \mathsf{capture}_1$, contradicting $\mathsf{capture}_1 = \emptyset$. We trivially have $t_2^{@v} = t_2^{@\rho_1(v)}$ and thus by Assumption 2-(i), $v \in dom(\rho_2)$. Then $\rho_2(v) = v$, by because otherwise $(v \mapsto \rho_2(v)) \in \mathsf{notPresrvRef2} \subseteq \mathsf{capture}_2$, contradicting $\mathsf{capture}_2 = \emptyset$. By Assumption 2-(ii), $\rho_1(v) = v = \rho_2(v)$.
3. If $v \notin V_s$, then $\rho_1(v) \notin V_s$ because otherwise $(v \mapsto \rho_1(v)) \in \mathsf{notPresrvDef} \subseteq \mathsf{capture}_1$, contradicting $\mathsf{capture}_1 = \emptyset$. By Assumption 1, $t_1^{@v} = t_1^{@\rho_1(v)}$, which implies $t_2^{@v} = t_2^{@\rho_1(v)}$ by the second condition of $t_1 \equiv_\alpha^{G_s} t_2$. By Assumption 2-(i), $v \in dom(\rho_2)$. We have $\rho_2(v) \notin V_s$ because otherwise $(v \mapsto \rho_2(v)) \in \mathsf{notPresrvDef} \subseteq \mathsf{capture}_2$, contradicting $\mathsf{capture}_2 = \emptyset$. By Assumption 1, $t_1^{@v} = t_1^{@\rho_1(v)}$ and thus $\rho_1(v) = \rho_2(v)$ by Assumption 2. $\square$

Lemma 4. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and program t with $\mathsf{capture} = \mathit{find\text{-}capture}(G_s, \mathit{resolve}^T(t)) \neq \emptyset$, renaming preserves sub-α-equivalence $t \equiv_\alpha^{G_s} \mathit{rename}(t, \pi_{src} \cup \pi_{syn})$ given π_{src} and π_{syn} as in name-fix.*

Proof. Let $(V_t, \rho_t) = \mathit{resolve}^T(t)$ and $t' = \mathit{rename}(t, \pi_{src} \cup \pi_{syn})$. First we have $t \equiv_\mathcal{L} t'$ by Lemma 2. By the definition of *rename* and since $dom(\pi_{src}) \cap dom(\pi_{syn}) = \emptyset$, $t'^{@v}$ becomes either $\pi_{src}(v)$ if $v \in dom(\pi_{src})$, $\pi_{syn}(v)$ if $v \in dom(\pi_{syn})$, and remains unchanged otherwise. We separately show that both conditions of sub-α-equivalence are satisfied:

1. For all $v_r, v_d \in V_t \cap V_s$ with $\rho_s(v_r) = v_d$ we have $v_r \notin dom(\pi_{syn})$, $v_d \notin dom(\pi_{syn})$, $v_r \in dom(\pi_{src}) \Leftrightarrow v_d \in dom(\pi_{src})$ because G_s is bipartite, and if $v_r \in dom(\pi_{src})$, then $\pi_{src}(v_r) = \pi_{src}(v_d)$. Thus, $t^{@v_r} = t^{@v_d} \Leftrightarrow t'^{@v_r} = t'^{@v_d}$.
2. For all $v_r, v_d \in V_t \setminus V_s$ we have $v_r \notin dom(\pi_{src})$ and $v_d \notin dom(\pi_{src})$. If $t^{@v_r} \neq t^{@v_d}$, then $t'^{@v_r} \neq t'^{@v_d}$ because π_{syn} maps distinct names to distinct fresh names. If instead $t^{@v_r} = t^{@v_d}$, we have $v_r \in dom(\pi_{syn}) \Leftrightarrow v_d \in dom(\pi_{syn})$ and if $v_r \in dom(\pi_{syn})$, then $\pi_{syn}(v_r) = \pi_{syn}(v_d)$. Thus, $t^{@v_r} = t^{@v_d} \Leftrightarrow t'^{@v_r} = t'^{@v_d}$. $\square$

Theorem 4. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and any program t, $\mathit{name\text{-}fix}(G_s, t) \equiv_\alpha^{G_s} t$.*

Proof. By induction on $\mathit{name\text{-}fix}(G_s, t)$ using Theorem 3 and Lemma 4. $\square$

Lemma 5. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and programs $t_1 \equiv_\alpha^{G_s} t_2$, if $\mathit{find\text{-}capture}(G_s, \mathit{resolve}^T(t_1)) = \emptyset$, then $t_1 \equiv_\alpha \mathit{name\text{-}fix}(G_s, t_2)$.*

Proof. By induction on $\mathit{name\text{-}fix}(G_s, t_2)$. Base case: $\mathit{find\text{-}capture}(G_s, \mathit{resolve}^T(t_2))$ $= \emptyset$ and $\mathit{name\text{-}fix}(G_s, t_2) = t_2$. Then $t_1 \equiv_\alpha t_2$ by Lemma 3. Step case: $\mathit{find\text{-}capture}(G_s, \mathit{resolve}^(t_2)) \neq \emptyset$ and $\mathit{name\text{-}fix}(G_s, t_2) = \mathit{name\text{-}fix}(G_s, t_2')$. Then $t_1 \equiv_\alpha^{G_s} t_2'$ by Lemma 4, and $t_1 \equiv_\alpha \mathit{name\text{-}fix}(G_s, t_2')$ by the induction hypothesis. □

Theorem 5. *For any bipartite name graph $G_s = (V_s, \rho_s)$ and programs $t_1 \equiv_\alpha^{G_s} t_2$, $\mathit{name\text{-}fix}(G_s, t_1) \equiv_\alpha \mathit{name\text{-}fix}(G_s, t_2)$.*

Proof. By induction on $\mathit{name\text{-}fix}(G_s, t_1)$. Base case by Lemma 5. Step case: $\mathit{find\text{-}capture}(G_s, \mathit{resolve}^(t_1)) \neq \emptyset$ and $\mathit{name\text{-}fix}(G_s, t_1) = \mathit{name\text{-}fix}(G_s, t_1')$. Then $t_1' \equiv_\alpha^{G_s} t_1$ by Lemma 4 and $t_1' \equiv_\alpha^{G_s} t_2$ by transitivity. Thus, $\mathit{name\text{-}fix}(G_s, t_1') \equiv_\alpha^{G_s}$ $\mathit{name\text{-}fix}(G_s, t_2)$ by the induction hypothesis. □

Theorem 6. *For any sub-hygienic transformation $f : S \to T$, transformation $\lambda\, s.\mathit{name\text{-}fix}(G_s, f(s))$ is hygienic.*

Proof. For any $s_1 \equiv_\alpha s_2$, $f(s_1) \equiv_\alpha^{G_s} f(s_2)$ by the definition of sub-hygiene. Then $\mathit{name\text{-}fix}(G_s, f(s_1)) \equiv_\alpha \mathit{name\text{-}fix}(G_s, f(s_2))$ by Theorem 5. □