Hygienic Resugaring of Compositional Desugaring

Justin Pombrio Shriram Krishnamurthi
Brown University (United States)
{justinpombrio, sk}@cs.brown.edu

Abstract

Syntactic sugar is widely used in language implementation. Its benefits are, however, offset by the comprehension problems it presents to programmers once their program has been transformed. In particular, after a transformed program has begun to evaluate (or otherwise be altered by a black-box process), it can become unrecognizable.

We present a new approach to *resugaring* programs, which is the act of reflecting evaluation steps in the core language in terms of the syntactic sugar that the programmer used. Relative to prior work, our approach has two important advances: it handles hygiene, and it allows almost arbitrary rewriting rules (as opposed to restricted patterns). We do this in the context of a DAG representation of programs, rather than more traditional trees.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords syntactic sugar, resugaring, hygiene, abstract syntax DAG

1. Introduction

Syntactic sugar has a venerable history in programming languages, starting with its use by Landin [10]. Desugaring is now actively used in many practical settings:

- In the definition of language constructs in many languages ranging from Python to Haskell.
- To extend the language, in languages ranging from the Lisp family to C++ to Julia.
- To shrink the semantics of large scripting languages with many special-case behaviors, such as JavaScript and Python, to small core languages that tools can more easily process.

Of course, once a program has been desugared, it is much harder for its programmer to recognize. Worse, when desugaring is followed by any phase that rewrites terms, such as evaluation, optimization, or theorem proving, there is typically no easy way to view the rewritten terms using their original pre-transformation syntax. This penalizes either the programmer who uses the sugar (who must contend with the details of desugaring) or the language designer (who must decide whether to forgo sugar and deal with a larger, more complex language). In short, it violates the abstraction that syntactic sugar ought to provide.

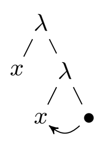
What we instead need is to lift an evaluation (or other reduction) sequence back to the surface language in terms of the original program. That is, we must *reconstruct* a source term that reflects what the intermediate term *would have been* had the reduction process been defined explicitly in terms of the source language (which, for practical reasons, it is not). We build on the idea of *resugaring* previously introduced by Pombrio and Krishnamurthi [13]. That work gives a method to reconstruct surface (i.e., pre-transformation) terms out of core (i.e., post-transformation) terms. This work improves upon that in two notable ways:

- The earlier work did not handle hygiene, which is a standard part of desugaring systems. Our work expressly handles hygiene.
- The earlier work handled only limited rewriting systems: ones where syntactic sugar could be expressed as a set of declarative rules in a very limited language (akin to the `syntax-rules` [15] macro system). This significantly limited the applicability of that work. This work permits the use of arbitrary *functions*, so long as they are compositional in the desugaring of their subterms (i.e., do not probe the content of the subterms). Our work can therefore handle the vast majority of complex desugaring rules used in real languages. For instance, the earlier paper could not handle some of the sugar used to implement Pyret (`pyret.org`), a new functional programming language, but the work in this paper can.

This work makes two additional contributions:

- Just like the prior work, we provide semantic guarantees about resugaring, so that a programmer gets output that is both meaningful and predictable. The previous work defined three goal properties: Emulation, Abstraction, and Coverage. We prove the same Emulation theorem (Theorem 1), prove a richer version of Abstraction (Theorem 2), and put Coverage — which was only evaluated empirically in the prior work — on a formal footing (Theorem 4).
- In defining this resugaring system, we shift from traditional abstract syntax trees to a different representation: abstract syntax *DAGs* (ASDs), whose back-edges represent references from bound to binding instances. Using this we are able to reconstruct a traditional hygiene theorem (Section 4.3) without having to assume that the desugaring algorithm is itself "hygienic".

An ASD is simply a tree that reflects binding structure. For instance, the ASD representation of the term $\lambda x.\, \lambda x.\, x$ is:



ASDs differ from typical AST representations in two ways: (i) their variable references unambiguously link to their declaration sites, and (ii) their nodes, including variable declarations, have identity. Thus, for instance, the two declarations of x above are *not* equal. Similarly, a second copy of this ASD would not be equal to the first, since its nodes would differ in identity. (It would, however, be isomorphic.)

Overall, then, our approach has the following shape. A program is initially converted from an abstract syntax tree to an ASD through a process called scope *resolution*, which makes the binding structure explicit. This program is then desugared. After each step of the resulting evaluation (or other transformation), our approach attempts to resugar it. If it can be resugared, the resulting ASD is then *unresolved* to produce a term in the source language; otherwise the step is skipped:

$$\begin{tikzcd}
\mathit{surf}\,\mathrm{AST}_1 \arrow[r, "\mathit{resolve}"] & \mathit{surf}\,\mathrm{ASD}'_1 \arrow[r, "\mathit{desugar}"] & \mathit{core}_1 \arrow[d, "\mathit{step}"] \\
 & \mathit{(skipped)} & \mathit{core}_2 \arrow[l, "\mathit{resugar\ fails}"'] \arrow[d, "\mathit{step}"] \\
\mathit{surf}\,\mathrm{AST}_2 & \mathit{surf}\,\mathrm{ASD}'_2 \arrow[l, "\mathit{unresolve}"'] & \mathit{core}_3 \arrow[l, "\mathit{resugar}"'] \arrow[d, "\mathit{step}"] \\
 & & \cdots
\end{tikzcd}$$

We discuss the structure of the ASD in Section 3, resolution in Section 3.2, unresolution in Section 3.3, resugaring (and desugaring) in Section 4, and how it applies to sequences of terms (including the skipping of terms) in Section 5.

Terminology

Throughout the paper, we will often make the following distinctions:

surface vs. core The *surface* is the language before desugaring, and the *core* is the language after.

declaration vs. reference A variable's *declaration* is the binding site that introduces it. A *reference* is a use of a variable, typically in expression position. We take this naming convention from Erdweg et al. [3].

2. A Worked Example

We will motivate our term representation by showing two problems that arise during resugaring, and how representing terms as ASDs instead of ASTs fixes both problems. The first, which arises during desugaring, is the familiar hygiene problem (Section 2.1), and is solved by the fact that the ASD distinguishes identifiers that happen to share the same name. The second problem (Section 2.2) arises when resugaring, and is solved by the fact that the ASD distinguishes nodes that happen to represent the same syntactic construct.

2.1 Desugaring: Variable Capture

The first column of Fig. 1 shows the unhygienic desugaring of a program, leading to variable capture; we will describe it in detail.

The premise of the example is that a programmer, while developing an application involving TCP/IP connections, invokes a syntactic sugar that performs logging. The surface program the programmer wrote is shown in the first column. (`VERBOSE` is a predefined constant.)

The definition of this sugar is shown in the second row. The sugar `log` α `to` β `when` γ writes α to the file-system port β when the condition γ is true.

A naive, unhygienic expansion of the `log` sugar is shown in the third row. The highlighted code simply shows the instantiation of pattern variables α, β, and γ (to improve legibility), and the $[C_1 \Rightarrow C_2]$ tag can be momentarily ignored. Unsurprisingly, this unhygienic expansion causes the variable `port` to be captured. As a result, the program eventually fails with a runtime type error when `to_str` is called on a file-system port.

Of course there are many hygienic transformation systems that could be used here. However, if we first resolve terms to ASDs the problem does not arise and an otherwise naive desugaring suffices. In particular, in an ASD, each variable declaration in a term has an unique identity. Rows 1–3 of the second column show the desugaring and subsequent core evaluation of the program as represented as an ASD. Since the two `port` variables are now represented distinctly, capture no longer occurs and the program behaves correctly when evaluated. As would be expected, the first evaluation step evaluates the `let`, and the second evaluates the outer `if`.

2.2 Resugaring: Code Capture

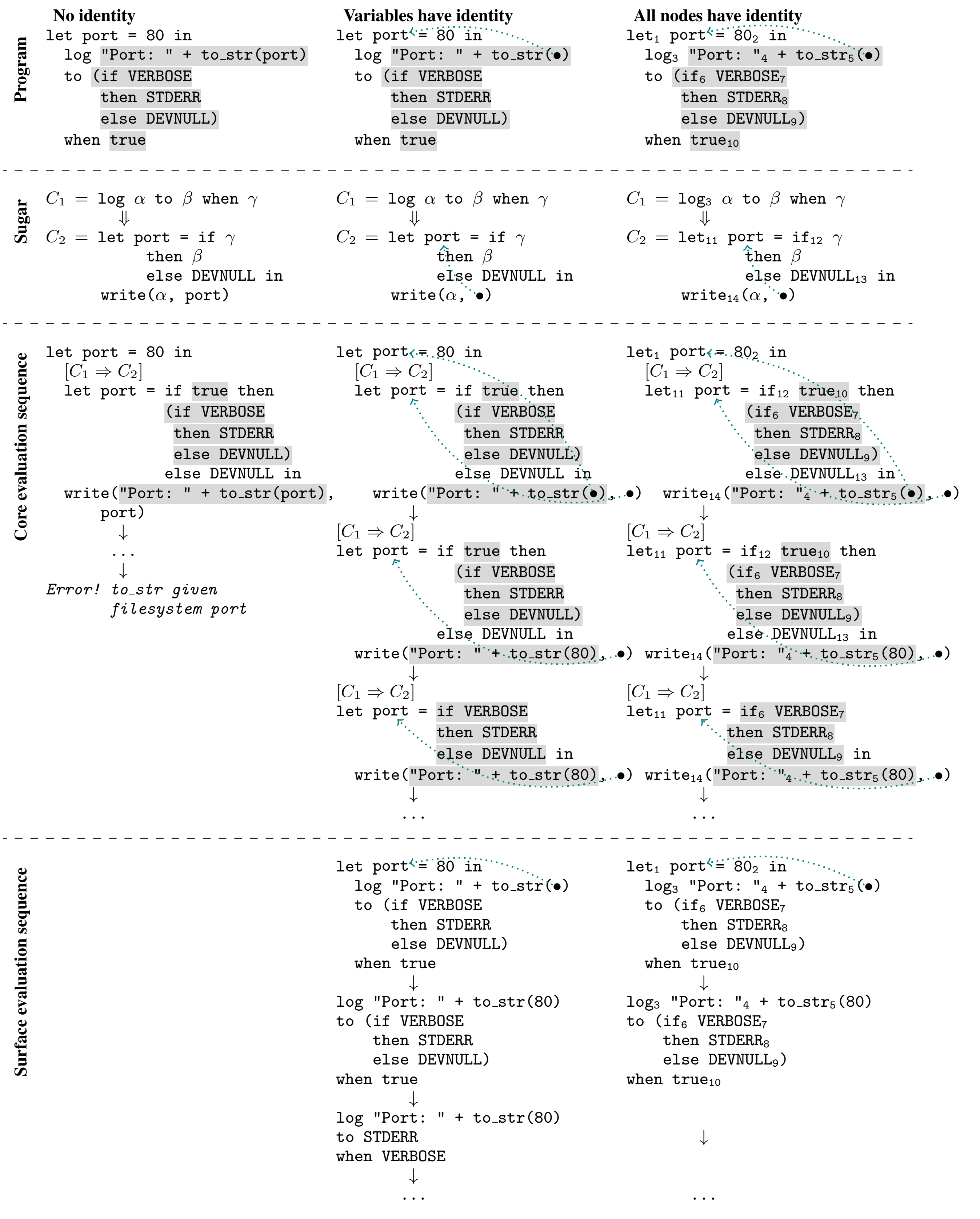
Let us see, however, what happens when this evaluation sequence is resugared. First of all, to be able to resugar, we must tag terms by the sugar they came from. This is necessary, for instance, to know whether the core code came from an invocation of `log`, or whether the programmer happened to write that code directly. Thus we put a *tag* $[C_1 \Rightarrow C_2]$ on the expansion of a sugar, where C_1 and C_2 are patterns representing the part of the term that was rewritten during desugaring. How this works in the face of arbitrary desugaring functions will be explained in Section 4.2. (There should also be a tag around the outer `let`; we have elided it for brevity.)

We will give a full account of resugaring in Section 4, but for now it suffices to say that to resugar a term t tagged by $[C_1 \Rightarrow C_2]$, undo the rewrite the sugar performed: check to see if t matches the pattern C_2, yielding a substitution that maps "pattern variables" to syntactic terms, and if so apply that substitution to C_1. Resugaring the core sequence above thus produces the surface evaluation sequence shown in the last row of column 2.

The first two steps are fine. The first term is the same as the original program (having been accurately reconstructed by resugaring), and the second shows that the `let` has been substituted properly. The third term is strange, however, and is a non sequitur with respect to the second.

What happened is that the *sugar's* `if` statement in C_2 was matched against the *programmer's* `if` statement, causing it to be "resugared". As a result, this surface term makes no sense as a follow-up to the previous surface term. We dub this "code capture", and it is somewhat analogous to variable capture. Just as renaming `port` to `tcp_port` would have changed the meaning of the program when unhygienically desugaring, refactoring the *surface* code `if VERBOSE then STDERR else DEVNULL` to `if not(VERBOSE) then DEVNULL else STDERR` would prevent this term from being resugared, changing the surface sequence.

In the third column of Fig. 1, each node in the term is given a unique identity. We represent their identity with numeric subscripts; these numbers have no further meaning and, e.g., do not represent an ordering. (One result of giving nodes identity is that each time a rule is applied it is freshly instantiated; thus the desugaring rule in the second row shows a particular instantiation of the `log` sugar.)



Program

No identity

```
let port = 80 in
  log "Port: " + to_str(port)
  to (if VERBOSE
      then STDERR
      else DEVNULL)
  when true
```

Variables have identity

```
let port = 80 in
  log "Port: " + to_str(•)
  to (if VERBOSE
      then STDERR
      else DEVNULL)
  when true
```

All nodes have identity

```
let_1 port = 80_2 in
  log_3 "Port: "_4 + to_str_5(•)
  to (if_6 VERBOSE_7
      then STDERR_8
      else DEVNULL_9)
  when true_10
```

Sugar

```
C_1 = log α to β when γ
        ⇓
C_2 = let port = if γ
           then β
           else DEVNULL in
      write(α, port)
```

```
C_1 = log α to β when γ
        ⇓
C_2 = let port = if γ
           then β
           else DEVNULL in
      write(α, •)
```

```
C_1 = log_3 α to β when γ
        ⇓
C_2 = let_11 port = if_12 γ
           then β
           else DEVNULL_13 in
      write_14(α, •)
```

Core evaluation sequence

```
let port = 80 in
  [C_1 ⇒ C_2]
  let port = if true then
                (if VERBOSE
                 then STDERR
                 else DEVNULL)
             else DEVNULL in
  write("Port: " + to_str(port),
      port)
        ↓
       ...
        ↓
Error! to_str given
       filesystem port
```

```
let port = 80 in
  [C_1 ⇒ C_2]
  let port = if true then
                (if VERBOSE
                 then STDERR
                 else DEVNULL)
             else DEVNULL in
    write("Port: " + to_str(•), •)
        ↓
[C_1 ⇒ C_2]
let port = if true then
              (if VERBOSE
               then STDERR
               else DEVNULL)
           else DEVNULL in
  write("Port: " + to_str(80), •)
        ↓
[C_1 ⇒ C_2]
let port = if VERBOSE
           then STDERR
           else DEVNULL in
  write("Port: " + to_str(80), •)
        ↓
       ...
```

```
let_1 port = 80_2 in
  [C_1 ⇒ C_2]
  let_11 port = if_12 true_10 then
                (if_6 VERBOSE_7
                 then STDERR_8
                 else DEVNULL_9)
             else DEVNULL_13 in
  write_14("Port: "_4 + to_str_5(•), •)
        ↓
[C_1 ⇒ C_2]
let_11 port = if_12 true_10 then
              (if_6 VERBOSE_7
               then STDERR_8
               else DEVNULL_9)
           else DEVNULL_13 in
  write_14("Port: "_4 + to_str_5(80), •)
        ↓
[C_1 ⇒ C_2]
let_11 port = if_6 VERBOSE_7
           then STDERR_8
           else DEVNULL_9 in
  write_14("Port: "_4 + to_str_5(80), •)
        ↓
       ...
```

Surface evaluation sequence

```
let port = 80 in
  log "Port: " + to_str(•)
  to (if VERBOSE
      then STDERR
      else DEVNULL)
  when true
        ↓
log "Port: " + to_str(80)
to (if VERBOSE
    then STDERR
    else DEVNULL)
when true
        ↓
log "Port: " + to_str(80)
to STDERR
when VERBOSE
        ↓
       ...
```

```
let_1 port = 80_2 in
  log_3 "Port: "_4 + to_str_5(•)
  to (if_6 VERBOSE_7
      then STDERR_8
      else DEVNULL_9)
  when true_10
        ↓
log_3 "Port: "_4 + to_str_5(80)
to (if_6 VERBOSE_7
    then STDERR_8
    else DEVNULL_9)
when true_10

        ↓

       ...
```

Figure 1. Desugaring and Resugaring Example

Crucially, in the last core step shown, since $\texttt{if}_6$ in the term does not match $\texttt{if}_{12}$ in the right-hand-side C_2 of the tag, this term cannot be resugared. As a result, it is correctly skipped in the surface evaluation sequence.

Thus changing the term representation from ASTs to ASDs prevented both variable capture and code capture. Variable capture was prevented because variables in an ASD have identity and variable references point directly to their declarations; while code capture was prevented because other nodes in an ASD also have identity.

3. Terms

We will now begin to describe our resugaring system formally, beginning with the definition of ASD terms. While ASDs are DAGs, the sharing present in them is limited to only their (variable) leaves, allowing us to use a simple textual representation: variables and nodes will be given subscripts that identify them. Thus we use subscripts to *represent* the DAG structure of terms. As an example, the desugared program from the previous section would be written:

```
let₁₀ port₁ = 80₁₁ in
  [C₁ ⇒ C₂] let₁₂ port₂ = if₁₃ true₁₄ then
      (if₁₅ VERBOSE₁₆ then STDERR₁₇ else DEVNULL₁₈)
      else DEVNULL₁₉ in
    write₁₀("Using port "₂₁ + to_str₂₂(port₁),
            port₂)
```

Our formal definition of terms is inspired by Gabbay and Pitts' Nominal Logic [5]. We start by defining two kinds of *atoms*: atoms that provide identity to nodes are taken from a set $\mathbb{A}$, and atoms that represent variables are pairs of a variable *name* from a set $\mathbb{X}$ and a unique *subscript* taken from $\mathbb{U}$:

$$\begin{array}{lcll} \textit{atom} & ::= & x_u & \text{where } x \in \mathbb{X} \text{ and } u \in \mathbb{U} \\ & | & a & \text{where } a \in \mathbb{A} \end{array}$$

In the case that a term's binding structure has not yet been resolved, a unique identifier $u \in \mathbb{U}$ will not have been chosen for its variables. In this case, we will write x for $x_{\textit{free}}$ where *free* is a distinguished element of $\mathbb{U}$. Likewise, let *free* also be a distinguished element of $\mathbb{A}$ for nodes that lack identity. (The two *frees* will be distinguished by context.)

Next we define terms over some fixed set of node types $\mathcal{N}$ as follows:

$$\begin{array}{lcll} t & ::= & \texttt{decl}(x_u) & \\ & | & \texttt{ref}(x_u) & \\ & | & \texttt{val}(val) & \\ & | & \texttt{node}_a(n, \overrightarrow{t_i}) & \text{where } n \in \mathcal{N} \\ & | & \texttt{tag}_{C_1 \Rightarrow C_2}\ t & \end{array}$$

Declarations `decl` represent variables in binding position, while references `ref` represent variables in use position. Nodes `node` represent both compound terms that have subterms, and constants. Tags $\texttt{tag}_{C_1 \Rightarrow C_2}$ record how a sugar was expanded so that it may be reversed later (patterns C will be defined momentarily). Values `val` represent runtime values. We are agnostic to the representation of values, and never inspect or modify val.

We do not assume that values have identity (i.e., subscripts), since this would require expensive runtime tagging. This introduces a problem, however: code capture could occur in part of a sugar that expanded to a value, since there would be no way to distinguish between, e.g., a $\texttt{val}(6)$ introduced by the sugar and a $\texttt{val}(6)$ introduced by the programmer. Thus the syntactic term `6` (that, when evaluated, produces the value 6) should be formally represented with a node such as $\texttt{node}_a(\texttt{int}, \texttt{val}(6))$.[1]

[1] This term/value distinction is also the reason that the term `80` loses its subscript after being evaluated to a value in Fig. 1.

Comparison to Traditional Hygiene

At first glance, our approach appears very similar to traditional approaches to hygiene, such as the original time-stamping algorithm by Kohlbecker et al. [8]. We will detail the similarities and differences here; our relationship to other work is given in Section 7. Their technique works by *coloring* all of the syntax with a fresh color (a syntax can have more than one color) at each expansion step. The set of unique colors that a variable has then serves to distinguish distinct identifiers that happen to share the same name. This would seem akin to our subscripts. However, our technique differs in three respects:

1. First, our variable subscripts *uniquely* determine identity, while theirs only determines identity up to the phases of expansion. For instance, if a macro expanded to $\lambda x.\, \lambda x.\, x$, their approach would color it $\lambda x_{\textit{phase1}}.\, \lambda x_{\textit{phase1}}.\, x_{\textit{phase1}}$. We, however, would resolve this term to $\lambda x_1.\, \lambda x_2.\, x_2$, distinguishing between the two xs introduced by the same phase of expansion.
2. Second, we give identity to all nodes, not just variables.
3. Third, scope for them is defined by the desugaring, whereas we define it explicitly for the surface language.

These technical differences reveal a philosophical difference: inasmuch as they assign unique colors to variables, it ends up *implicitly* reconstructing DAGs, whereas we do so directly and completely.

Desugaring and resugaring will also make use of *patterns*, which are terms with *holes* α_i (i.e. "pattern variables") in them:

$$\begin{array}{lcll} C & ::= & \texttt{decl}(x_u) & \\ & | & \texttt{ref}(x_u) & \\ & | & \texttt{val}(val) & \\ & | & \texttt{node}_a(n, \overrightarrow{C_i}) & \text{where } n \in \mathcal{N} \\ & | & \alpha_i & \text{for } i \in \mathbb{N} \end{array}$$

Holes may occur at most once in a pattern.

We will distinguish between terms (and patterns) whose binding structure has been resolved, and those whose binding structure is still unresolved. Unresolved terms are traditional ASTs, while resolved terms are our ASDs. In an *unresolved* term, then, every atom has the form x (that is, $x_{\textit{free}}$). In a *resolved* term, however, every declaration atom has a unique name x_u, so there can be no confusion between two variables that happen to share the same name. Later, in Section 3.2, we will show how to *resolve* the binding structure of a term, given scoping rules for the language.

3.1 Permutations

We will use *permutations* both to define α-equivalence and to resolve terms' scope. Permutations can act both on atoms directly, or on terms. A permutation applied to a term will act on the atoms of the term, leaving its overall shape unchanged. Permutations are defined as follows, and their action is shown in Fig. 2:

$$\sigma ::= \epsilon \mid (a \leftrightarrow b) \mid \sigma_1 \circ \sigma_2$$

Permutations form a group where ϵ is the identity, $\circ$ is group multiplication, and σ^{-1} is given by $(a \leftrightarrow b)^{-1} = (a \leftrightarrow b)$ and $(\sigma_1 \circ \sigma_2)^{-1} = \sigma_2^{-1} \circ \sigma_1^{-1}$. The *domain* of a permutation is the set of elements it permutes: $dom(\sigma) = \{a \mid \sigma \bullet a \neq a\}$.

It will be useful to compute a *union* of permutations $\sigma_1 + \sigma_2$ that has the same action as either of them over their domains. More precisely, let $\sigma_1 \subseteq \sigma_2$ mean that for all $a \in dom(\sigma_1)$, $\sigma_1 \bullet a = \sigma_2 \bullet a$. Then $\sigma_3 = \sigma_1 + \sigma_2$ is the least permutation such that $\sigma_3 \supseteq \sigma_1$ and $\sigma_3 \supseteq \sigma_2$, and can be computed using the

$$
\begin{array}{lcll}
\boxed{\sigma \bullet a \mapsto a} & & & \\
(a \leftrightarrow b) \bullet a & = & b & \\
(a \leftrightarrow b) \bullet b & = & a & \\
(a \leftrightarrow b) \bullet c & = & c & \text{when } c \notin \{a, b\} \\
 & & & \\
\boxed{\sigma \bullet t \mapsto t} & & & \\
\epsilon \bullet t & = & t & \\
(\sigma_1 \circ \sigma_2) \bullet t & = & \sigma_1 \bullet \sigma_2 \bullet t & \\
\sigma \bullet \alpha_i & = & \alpha_i & \\
\sigma \bullet \texttt{decl}(x_u) & = & \texttt{decl}(\sigma \bullet x_u) & \\
\sigma \bullet \texttt{ref}(x_u) & = & \texttt{ref}(\sigma \bullet x_u) & \\
\sigma \bullet \texttt{val}(val) & = & \texttt{val}(val) & \\
\sigma \bullet \texttt{node}_a(n, \overrightarrow{t_i}) & = & \texttt{node}_{\sigma \bullet a}(n, \overrightarrow{\sigma \bullet t_i}) & \\
\sigma \bullet \texttt{tag}_{C \Rightarrow C'}\ t & = & \texttt{tag}_{\sigma \bullet C \Rightarrow \sigma \bullet C'}\ \sigma \bullet t &
\end{array}
$$

Figure 2. Permuting

following rules (and is undefined when none apply):

$$
(\sigma_1 + \sigma_2) \bullet a = \begin{cases} \sigma_1 \bullet a & \text{if } a \notin dom(\sigma_2) \\ \sigma_2 \bullet a & \text{if } a \notin dom(\sigma_1) \\ b & \text{if } \sigma_1 \bullet a = \sigma_2 \bullet a = b \end{cases}
$$

3.2 Resolution, Informally

As we argued earlier, it is best to think of terms as DAGs. It is then intuitively clear that capture will not be a problem. Our intuition relies on the fact that each variable declaration in the term is unique.

We will show how to *resolve* a term t that does not initially have this property by making each of its declarations fresh. We call the resolution operator $\mathcal{R}$. There are two situations in which this resolution will be necessary:

1. First, the initial program written by the programmer must be resolved.
2. Second, when a piece of sugar is expanded, the code introduced by the sugar must be resolved.

To give an example of resolution, consider a simplified version of the initial program from Section 2:

```
let port = 80 in
  log port to STDERR when true
```

Roughly speaking, $\mathcal{R}$ chooses a fresh identity x_u for each variable declaration x, and then permutes x with x_u within the scope of that declaration. At the same time, nodes are assigned fresh identities. In this example, `port` would be assigned a fresh subscript $\texttt{port}_1$, and the permutation $(\texttt{port} \leftrightarrow \texttt{port}_1)$ would be applied in its scope, producing:

$$
\begin{array}{l}
\texttt{let}_{10}\ \texttt{port}_1\ \texttt{=}\ \texttt{80}_{11}\ \texttt{in} \\
\quad (port \leftrightarrow port_1)\ \ \bullet \\
\qquad \texttt{log}_{12}\ \texttt{port to STDERR}_{13}\ \texttt{when true}_{14} \\
= \texttt{let}_{10}\ \texttt{port}_1\ \texttt{=}\ \texttt{80}_{11}\ \texttt{in} \\
\qquad \texttt{log}_{12}\ \texttt{port}_1\ \texttt{to STDERR}_{13}\ \texttt{when true}_{14}
\end{array}
$$

3.3 Unresolution, Informally

While having fresh declarations is helpful to ensure properties like hygiene, the user of the language should not be exposed to them. Often their subscripts can simply be dropped, but other times this would result in variable capture. Thus we will give an *unresolution* algorithm that renames variables as necessary to avoid capture. (This is left implicit in many other hygiene algorithms that either (i) perform spurious renaming or (ii) color variables but do not say how to present them.) Our algorithm for doing so tries to use variables' original names, and renames a variable only when it is threatened with capture, as shown in Lemma 2.

We present an example of this algorithm using the term from Section 2 that threatened variable capture. We will make a few changes for expository purposes: we simplify the program to focus on its binding structure and introduce one extra `let` binding to better show the behavior of unresolution. We also ignore the identities of nodes (which are removed during unresolution in a straightforward way) and focus just on variables. Here is the term we wish to unresolve:

$$
\begin{array}{l}
\texttt{let msg}_1\ \texttt{= "Port: " in} \\
\quad \texttt{let port}_2\ \texttt{= 80 in} \\
\qquad \texttt{let port}_3\ \texttt{= STDERR in} \\
\qquad\quad \texttt{write(msg}_1\ \texttt{+ to\_str(port}_2\texttt{), port}_3\texttt{)}
\end{array}
$$

Unresolution proceeds in two phases. The first phase, *findThreats*, safely but conservatively estimates the set of variable references at risk of capture. Specifically, it estimates that a variable x_u is at risk of being captured iff it is in scope of a *different* variable $x_{u'}$ of the same name.

In this case, *findThreats*:

- correctly concludes that $\texttt{msg}_1$ is not at risk of capture, since it is not in scope of any other variable of the same name
- correctly concludes that $\texttt{port}_2$ is at risk of capture, since it is in scope of $\texttt{port}_3$
- over-conservatively concludes that $\texttt{port}_3$ is at risk of capture, since it is in scope of $\texttt{port}_2$

Thus the final set of threats returned is $\{\texttt{port}_2, \texttt{port}_3\}$

The second phase, *renameThreats*, begins by picking a fresh variable name for each threatened variable, perhaps producing the map $\texttt{port}_2 \mapsto \texttt{portA},\ \texttt{port}_3 \mapsto \texttt{portB}$. It then renames all variables in the term: threatened variables are looked up in the map, while others simply have their suffix removed, producing:

```
let msg = "Port: " in
  let portA = 80 in
    let portB = STDERR in
      write(msg + to_str(portA), portB)
```

Combining these two phases will give an *unresolution* operator $\mathcal{U}$ that turns ASDs back into ASTs.

3.4 Resolution and Unresolution, Formally

We have given examples of scope resolution and unresolution, and now present them formally. To begin, we need a language-agnostic algebra for expressing the scoping rules of a language. We will use the *binding combinators* defined in the Romeo expansion system [16]. (It is worth noting, however, that the rest of our system relies only on term resolution and unresolution; thus a different scope resolution mechanism could be substituted in place of Romeo's.) In Romeo's scoping algebra, terms can export bindings to be used by other terms, and a term that has subterms can choose which of its subterms' exported bindings should be imported into which of its other subterms. The combinators β for expressing binding imports and exports are:[2]

$$\beta ::= \epsilon \mid i \mid \beta_1 \circ \beta_2 \mid \beta_1 + \beta_2$$

Here ϵ is the empty binding, i denotes the bindings exported by the i'th subterm, $\circ$ denotes left-biased union, and $+$ denotes disjoint union. The meaning $[\![\beta]\!]\ (\overrightarrow{\sigma_i})$ of these combinators is given by how they act on a list of permutations $\overrightarrow{\sigma_i}$ (specifically, the permutations exported by the nodes of their children). They can also act on sets

[2] In Romeo, the combinators $\circ$ and $+$ are written $\triangleright$ and $\uplus$ respectively, and their action is defined differently, but they behave the same.

$$\begin{array}{lcl}
\boxed{[\![\beta]\!]\,(\overrightarrow{\sigma}) \mapsto \sigma} & & \\
[\![\epsilon]\!]\,(\overrightarrow{\sigma_i}) & = & \epsilon \\
[\![j]\!]\,(\overrightarrow{\sigma_i}) & = & \sigma_j \\
[\![\beta_1 \circ \beta_2]\!]\,(\overrightarrow{\sigma_i}) & = & [\![\beta_1]\!]\,(\overrightarrow{\sigma_i}) \circ [\![\beta_2]\!]\,(\overrightarrow{\sigma_i}) \\
[\![\beta_1 + \beta_2]\!]\,(\overrightarrow{\sigma_i}) & = & [\![\beta_1]\!]\,(\overrightarrow{\sigma_i}) + [\![\beta_2]\!]\,(\overrightarrow{\sigma_i})
\end{array}$$

$$\begin{array}{lcl}
\boxed{[\![\beta]\!]\,(\overrightarrow{\{x_u,...\}}) \mapsto \{x_u,...\}} & & \\
[\![\epsilon]\!]\,(\overrightarrow{S_i}) & = & \emptyset \\
[\![j]\!]\,(\overrightarrow{S_i}) & = & S_j \\
[\![\beta_1 \circ \beta_2]\!]\,(\overrightarrow{S_i}) & = & [\![\beta_1]\!]\,(\overrightarrow{S_i}) \cup [\![\beta_2]\!]\,(\overrightarrow{S_i}) \\
[\![\beta_1 + \beta_2]\!]\,(\overrightarrow{S_i}) & = & [\![\beta_1]\!]\,(\overrightarrow{S_i}) \cup [\![\beta_2]\!]\,(\overrightarrow{S_i})
\end{array}$$

Figure 3. Binding Combinators

of variables. Their action in either case is shown in Fig. 3. The pun of using ϵ, $\circ$ and $+$ both for permutations and as the binding combinators is on purpose, as each is just the lifted form of the other.

These binding combinators are used to give a *binding signature* $sign(n) = (\overrightarrow{\beta_i}) \uparrow \beta$ to each node constructor n, where β_i are the imports of its children, and β are its exports. (The up-arrow is merely notation for a pair.)

The algorithms for scope resolution $\mathcal{R}$ and unresolution $\mathcal{U}$ are given in Fig. 4. In the figure, $new\,u.\,\_$ generates a globally unique fresh name or id u, and $t\,//\,C$ is used to copy the fresh ids chosen for t onto C. In $\mathcal{R}$, recursive calls return a pair of a term t and the permutation σ that it exports; this pair is written $t \uparrow \sigma$. We will slightly abuse notation by using term/permutation pairs, like $\mathcal{R}(t) = t' \uparrow \sigma$, in situations where terms are expected; in this case we mean for the permutation to be ignored.

The $\mathcal{U}$ function uses three helper functions: (i) *exports*(t) finds the set of variable declarations provided by a term t, (ii) *findThreats*(t, S) recursively finds threatened variables in term t (S is the set of variables "in scope" at t), and (iii) *renameThreats*(t, f) renames variables in t according to f.

Scope resolution and unresolution are approximately inverses of one another. To make this formal, say that two terms t_1 and t_2 are *isomorphic* $t_1 \simeq t_2$ when they differ only up to a permutation:

Definition 1 (isomorphism). $t_1 \simeq t_2$ when $\exists \sigma.\, \sigma \bullet t_1 = t_2$

Then resolution and unresolution obey the rule:

Lemma 1. $\mathcal{R}(\mathcal{U}(\mathcal{R}(t))) \simeq \mathcal{R}(t)$

Proof sketch. We aim to show that performing $\mathcal{U}$ and then $\mathcal{R}$ on a term $\mathcal{R}(t)$ is the identity up to permutation. Neither $\mathcal{R}$ nor $\mathcal{U}$ change the shape of the term, so we only need consider how they modify variables. Consider first the variable declarations, then references.

A variable declaration $\texttt{decl}(x_u)$ in $\mathcal{R}(t)$ will get mapped by f in $\mathcal{U}$ to some $\texttt{decl}(y)$, and then to some $\texttt{decl}(y_v)$ for fresh v in $\mathcal{R}$. This is fine.

Now consider references. The only concern is that some reference $\texttt{ref}(x_u)$ in $\mathcal{R}(t)$ might get mapped to $\texttt{ref}(y)$ by f (as it must) but then get mapped to some $\texttt{decl}(y_{v'})$ for $v' \neq v$ by $\mathcal{R}$. Since the reference $\texttt{ref}(x_u)$ in $\mathcal{R}(t)$ obtained the subscript u via $\mathcal{R}$ in the first place, it must have been acted on by the permutation $(x \leftrightarrow x_u)$ from $\texttt{decl}(x_u)$. Thus, in the second $\mathcal{R}$ step, it will be acted on by the permutation $(y \leftrightarrow y_v)$ from $\texttt{decl}(y_v)$. The only remaining concern is that it may also be acted on by a *different* permutation $(y \leftrightarrow y_{v'})$ with $v' \neq v$. But any variable in danger of causing this would have been found by *findThreats* and renamed during $\mathcal{U}$. □

$$\begin{array}{lcll}
\boxed{\Sigma \bullet C \mapsto t} & & & \\
\Sigma \bullet \texttt{val}(v) & = & \texttt{val}(v) & \\
\Sigma \bullet \alpha_i & = & t & \text{when } \alpha_i \to t \in \Sigma \\
\Sigma \bullet \texttt{decl}(x_u) & = & \texttt{decl}(x_u) & \\
\Sigma \bullet \texttt{ref}(x_u) & = & \texttt{ref}(x_u) & \\
\Sigma \bullet \texttt{node}_a(n, \overrightarrow{C_i}) & = & \texttt{node}_a(n, \overrightarrow{\Sigma \bullet C_i}) &
\end{array}$$

Figure 5. Substitution

Once terms have been resolved, it is easy to compare them for equality up to renaming: two resolved terms are *α-equivalent* when they are identical up to a permutation of their variables. We will write $t_1 =_\alpha t_2$ to mean that t_1 and t_2 are α-equivalent.

Definition 2 (α-equivalence). $t_1 =_\alpha t_2$ when $\mathcal{R}(t_1) \simeq \mathcal{R}(t_2)$

Lemma 2. *$\mathcal{U}(t)$ will only rename a variable reference x_u in t if it is in scope of a declaration $x_{u'}$ with $u' \neq u$.*

Proof. The only variables which are renamed by *renameThreats* are those in the set returned by *findThreats*, so we just need to argue that *findThreats* only finds threatened variables. The only nonempty base case for *findThreats* is that for variable references, given by:

$$findThreats(\texttt{ref}(x_u), S) = \{x_u\} \quad \text{if } \{x_{u'} \in S \mid u \neq u'\} \neq \emptyset \quad \text{else } \emptyset \text{ blah}$$

The set S of variables it passes along recursively is precisely the set of variables in scope at that point, so *findThreats* will only produce $\{x_u\}$ when some $x_{u'}$ "threatens" to capture x_u. □

4. Desugaring and Resugaring

In this section, we introduce the primary algorithms of our resugaring system: the algorithms for desugaring and resugaring individual terms. They can then be used to resugar an evaluation sequence via the pseudo-code algorithm:

```
def showSurfaceSequence($s$):
  let $c$ = $desugar(s)$
  while $c$ can take a reduction step:
    let $s'$ = $resugar(c)$
    if $s'$ was successful: print($s'$)
    $c$ := step($c$)
```

4.1 Matching and Substitution

During desugaring and resugaring, our system will *match* terms against patterns, producing a *substitution* from holes α_i to subterms, and then apply this substitution to another pattern.

$$\Sigma ::= \epsilon \mid \alpha_i \to t \mid \Sigma_1 \circ \Sigma_2$$

We will overload the notations $\bullet$ and $\circ$ to also refer to substitution, and will define symmetric composition the same way as for permutations:

$$(\Sigma_1 + \Sigma_2) \bullet \alpha = \begin{cases} \Sigma_1 \bullet \alpha & \text{if } \alpha \notin dom(\Sigma_2) \\ \Sigma_2 \bullet \alpha & \text{if } \alpha \notin dom(\Sigma_1) \\ t & \text{if } \Sigma_1 \bullet \alpha = \Sigma_2 \bullet \alpha = t \end{cases}$$

Substitution is defined in Fig. 5. Unlike permutations, substitutions do not form a group because they typically do not have inverses.

A term can be *matched* against a pattern to produce a substitution, as shown in Fig. 6. Matching and substitution are nearly

$$\begin{array}{lcll}
\boxed{\mathcal{R}(t) \mapsto t} & & & \\
\mathcal{R}(t) & = & \mathit{fst}(\mathcal{R}_1(t)) & \text{(where } \mathit{fst}(t \uparrow \sigma) = t\text{)}
\end{array}$$

$$\begin{array}{lcll}
\boxed{\mathcal{U}(t) \mapsto t} & & & \\
\mathcal{U}(t) & = & \mathit{renameThreats}(t, f) & \text{where } S = \mathit{findThreats}(t, \emptyset) \\
 & & & \text{and } f(x_u) = \mathit{new}\, y.\, y \text{ for } x_u \in S \\
 & & & \text{and } f(x_u) = x \text{ otherwise}
\end{array}$$

$$\begin{array}{lcll}
\boxed{\mathcal{R}_1(t) \mapsto t \uparrow \sigma} & & & \\
\mathcal{R}_1(\mathtt{val}(\mathit{val})) & = & \mathtt{val}(\mathit{val}) \uparrow \epsilon & \\
\mathcal{R}_1(\mathtt{ref}(x_u)) & = & \mathtt{ref}(x_u) \uparrow \epsilon & \\
\mathcal{R}_1(\mathtt{decl}(x_u)) & = & \mathit{new}\, u'.\, \mathtt{decl}(x_{u'}) \uparrow (x_u \leftrightarrow x_{u'}) & \\
\mathcal{R}_1(\mathtt{tag}_{C \Rightarrow C'}\ t) & = & \mathtt{tag}_{C \Rightarrow (t' // C')}\ t' & \text{where } t' = \mathcal{R}_1(t) \\
\mathcal{R}_1(\mathtt{node}_a(n, \overrightarrow{s_i})) & = & \mathit{new}\, b.\, \mathtt{node}_b(n, \overrightarrow{[\![\beta_i]\!]\ (\overrightarrow{\sigma_j}) \bullet t_i}) \uparrow [\![\beta]\!]\ (\overrightarrow{\sigma_j}) & \text{when } \overrightarrow{\mathcal{R}_1(s_i) = t_i \uparrow \sigma_i} \\
 & & & \text{and } \mathit{sign}(n) = \overrightarrow{(\beta_i)} \uparrow \beta
\end{array}$$

$$\begin{array}{lcl}
\boxed{t \,//\, C \mapsto C} & & \\
t \,//\, \alpha_i & = & \alpha_i \\
\mathtt{val}(v) \,//\, \mathtt{val}(v) & = & \mathtt{val}(v) \\
\mathtt{decl}(x_u) \,//\, \mathtt{decl}(x_v) & = & \mathtt{decl}(x_u) \\
\mathtt{ref}(x_u) \,//\, \mathtt{ref}(x_v) & = & \mathtt{ref}(x_u) \\
\mathtt{node}_a(n, \overrightarrow{t_i}) \,//\, \mathtt{node}_a(n, \overrightarrow{C_i}) & = & \mathtt{node}_a(n, \overrightarrow{t_i \,//\, C_i})
\end{array}$$

$$\begin{array}{lcll}
\boxed{\mathit{findThreats}(t, \{x_u, ...\}) \mapsto \{x_u, ...\}} & & & \\
\mathit{findThreats}(\mathtt{val}(\mathit{val}), S) & = & \emptyset & \\
\mathit{findThreats}(\mathtt{ref}(x_u), S) & = & \{x_u\} & \text{when } \{x_{u'} \in S \mid u \neq u'\} \neq \emptyset \\
\mathit{findThreats}(\mathtt{ref}(x_u), S) & = & \emptyset & \text{otherwise} \\
\mathit{findThreats}(\mathtt{decl}(x_u), S) & = & \emptyset & \\
\mathit{findThreats}(\mathtt{node}_a(n, \overrightarrow{t_i}), S) & = & \bigcup \overrightarrow{\mathit{findThreats}(t_i, S \cup [\![\beta_i]\!]\ (\overrightarrow{\mathit{exports}(t_j)}))} & \text{when } \mathit{sign}(n) = \overrightarrow{(\beta_i)} \uparrow \beta
\end{array}$$

$$\begin{array}{lcl}
\boxed{\mathit{renameThreats}(t, x_u \mapsto x_u) \mapsto t} & & \\
\mathit{renameThreats}(\mathtt{val}(\mathit{val}), f) & = & \mathtt{val}(\mathit{val}) \\
\mathit{renameThreats}(\mathtt{ref}(x_u), f) & = & \mathtt{ref}(f(x_u)) \\
\mathit{renameThreats}(\mathtt{decl}(x_u), f) & = & \mathtt{decl}(f(x_u)) \\
\mathit{renameThreats}(\mathtt{node}_a(n, \overrightarrow{t_i}), f) & = & \mathtt{node}(n, \overrightarrow{\mathit{renameThreats}(t_i, f)})
\end{array}$$

$$\begin{array}{lcll}
\boxed{\mathit{exports}(t) \mapsto \{x_u, ...\}} & & & \\
\mathit{exports}(\mathtt{val}(\mathit{val})) & = & \emptyset & \\
\mathit{exports}(\mathtt{ref}(x_u)) & = & \emptyset & \\
\mathit{exports}(\mathtt{decl}(x_u)) & = & \{x_u\} & \\
\mathit{exports}(\mathtt{node}_a(n, \overrightarrow{t_i})) & = & [\![\beta]\!]\ (\overrightarrow{\mathit{exports}(t_i)}) & \text{when } \mathit{sign}(n) = \overrightarrow{(\beta_i)} \uparrow \beta
\end{array}$$

Figure 4. Resolution and Unresolution

$$\begin{array}{lcl}
\boxed{t \,/\, C \mapsto \Sigma} & & \\
t \,/\, \alpha_i & = & \alpha_i \to t \\
\mathtt{val}(v) \,/\, \mathtt{val}(v) & = & \epsilon \\
\mathtt{decl}(x_u) \,/\, \mathtt{decl}(x_u) & = & \epsilon \\
\mathtt{ref}(x_u) \,/\, \mathtt{ref}(x_u) & = & \epsilon \\
\mathtt{node}_a(n, \overrightarrow{t_i}) \,/\, \mathtt{node}_a(n, \overrightarrow{C_i}) & = & t_1 \,/\, C_1 + t_2 \,/\, C_2 + ...
\end{array}$$

Figure 6. Matching

inverses of one another: substitution is an inverse of matching and, given a reasonable precondition, matching is an inverse of substitution. Recall that we call the "pattern variables" in a pattern *holes*, and let $\mathit{holes}(C)$ be the set of all holes in the pattern. Then:

Lemma 3. *For all patterns C and substitutions Σ, if $\mathit{domain}(\Sigma) = \mathit{holes}(C)$, then $(\Sigma \bullet C) \,/\, C = \Sigma$*

Proof. Induct on C. In the inductive case,

$$\begin{array}{l}
(\Sigma \bullet \mathtt{node}_a(n, \overrightarrow{C_i})) \,/\, \mathtt{node}_a(n, \overrightarrow{C_i}) \\
= \mathtt{node}_a(n, \overrightarrow{\Sigma \bullet C_i}) \,/\, \mathtt{node}_a(n, \overrightarrow{C_i}) \\
= (\Sigma \bullet C_1) \,/\, C_1 + (\Sigma \bullet C_2) \,/\, C_2 + ... \\
= \Sigma_1 + \Sigma_2 + ... \quad \text{(by I.H.)} \\
= \Sigma
\end{array}$$

where Σ_i is Σ restricted to the holes of C_i. The last step relies on holes occurring at most once in C. ☐

Lemma 4. *For all terms t and patterns C, if $t \,/\, C$ exists then $(t \,/\, C) \bullet C = t$*

Proof. Induct on C. In the inductive case,

$$
\begin{aligned}
&(\mathtt{node}_a(n, \overrightarrow{t_i}) \,/\, \mathtt{node}_a(n, \overrightarrow{C_i})) \bullet \mathtt{node}_a(n, \overrightarrow{C_i}) \\
&= (t_1 \,/\, C_1 + ...) \bullet \mathtt{node}_a(n, \overrightarrow{C_i}) \\
&= \mathtt{node}_a(n, \overrightarrow{(t_1 \,/\, C_1 + ...) \bullet C_i}) \\
&= \mathtt{node}_a(n, \overrightarrow{(t_i \,/\, C_i \bullet C_i})) \\
&= \mathtt{node}_a(n, \overrightarrow{t_i}) \quad \text{(by I.H.)}
\end{aligned}
$$

(The second to last step is valid because for $\mathtt{node}_a(n, \overrightarrow{t_i}) \,/\, \mathtt{node}_a(n, \overrightarrow{C_i})$ to exist, $(t_1 \,/\, C_1 + ...)$ must all be disjoint, and $t_i \,/\, C_i$ binds all holes in C_i.) ☐

4.2 Desugaring and Resugaring

Now we can define *desugaring* and *resugaring* operations that translate ASDs in the surface language to ASDs in the core language and back.

Desugaring uses a helper function called *expand* that expands a single piece of syntactic sugar in a term. *Expand* looks up a desugaring function to apply based on the term's topmost node and applies it. This function can be Turing-complete, and is written in the host language. In order for resugaring to work, however, desugaring must be *compositional*, i.e., it must be parametric over its subterms. Hence, instead of expanding the entire term t at once, *expand* will first split it into a pattern and subterms, and then only expand the *pattern* C to a new pattern C'. *Expand* then returns the pair (C, C') of the old and new pattern.

Desugaring of a term t thus proceeds by calling *expand*(t) to obtain the pair of patterns (C, C'), using matching and substitution to rewrite C to C', and recursively substituting the desugared subterms of t. The newly desugared term will be wrapped in a tag noting the original and new patterns. Later, resugaring will make use of these tags to undo each of the desugaring functions.

Desugaring makes use of two operations over nodes. *sugars*$(n)(C)$ looks up the desugaring function associated with node type n and applies it to pattern C, and *head*(t) splits the term t to obtain the pattern C to be desugared. The pattern returned by *head*(t) may need to be more than just the topmost node of t. Take, for instance, a multi-armed let construct like `let x = 4, y = x in x + y`. One way of representing this term in our system is:

```
node(Let,
  node(Bind, decl(x), node(Num, val(4)),
    node(Bind, decl(y), ref(x)),
      node(EndBinds))),
  node(Plus, ref(x), ref(y)))
```

It would be important for `Let`'s desugaring function to be given all of its bindings, so the pattern returned by *head* in this case should be:

```
node(Let,
  node(Bind, α1, α2,
    node(Bind, α3, α4,
      node(EndBinds))),
  α5)
```

While *head* could in principle be a complicated function, we believe in practice it is sufficient to partition nodes into *primary* nodes like `Let` that can stand on their own, and secondary nodes like `Bind` and `EndBinds` that are merely part of the node above them; thus we define *head* in terms of a *is-primary* predicate. *is-primary* will return `true` for values, declarations, and references; it is language specific for `nodes`.

Desugaring is formally defined in Fig. 7, and resugaring in Fig. 8.[3] Desugaring and resugaring are overloaded to act on substitutions in the obvious way, e.g., $\Downarrow(\alpha \to t) = \alpha \to (\Downarrow t)$. Desugaring and resugaring are inverses of one another, up to a permutation of variables.

[3] Notice that resugaring begins with a *resolve* step: this is only really necessary in case evaluation copies a term, thus breaking the invariant that variable declarations in resolved terms all have unique subscripts.

$\boxed{\textit{desugar}(t) \mapsto t}$

$$
\begin{aligned}
\textit{desugar}(t) \quad &= \quad \Downarrow(\mathcal{R}(t)) \\
\Downarrow \mathtt{node}_a(n, \overrightarrow{t_i}) \quad &= \quad \mathtt{tag}_{C \Rightarrow C'}\ (\Downarrow(t \,/\, C) \bullet C') \\
&\qquad \text{when } \textit{expand}(t) = (C, C') \\
&\qquad \text{where } t = \mathtt{node}_a(n, \overrightarrow{t_i}) \\
\Downarrow t \quad &= \quad t \quad \text{otherwise}
\end{aligned}
$$

$\boxed{\textit{expand}(t) \mapsto (C, C)}$

$$
\begin{aligned}
\textit{expand}(\mathtt{node}_a(n, \overrightarrow{t_i})) \quad &= \quad (C, C') \\
&\qquad \text{when } \textit{head}(\mathtt{node}_a(n, \overrightarrow{t_i})) = C \\
&\qquad \text{and } \mathcal{R}(\textit{sugars}(n)(C)) = C'
\end{aligned}
$$

$\boxed{\textit{head}(t) \mapsto C}$

$$
\begin{aligned}
\textit{head}(\mathtt{node}_a(n, \overrightarrow{t_i})) \quad &= \quad \mathtt{node}_a(n, \overrightarrow{\textit{head}_{rec}(t_i)}) \\
&\qquad \text{when } \textit{is-primary}(n) \\
\textit{head}_{rec}(\mathtt{node}_a(n, \overrightarrow{t_i})) \quad &= \quad \mathtt{node}_a(n, \overrightarrow{\textit{head}_{rec}(t_i)}) \\
&\qquad \text{when } \textit{not}(\textit{is-primary}(n)) \\
\textit{head}_{rec}(t) \quad &= \quad \textit{new}\, i.\, \alpha_i \quad \text{otherwise}
\end{aligned}
$$

Figure 7. Desugaring

$\boxed{\textit{resugar}(t) \mapsto t \text{ or FAIL}}$

$$
\begin{aligned}
\textit{resugar}(t) \quad &= \quad \mathcal{U}(\Uparrow(\mathcal{R}(t))) \\
\Uparrow(\mathtt{tag}_{C \Rightarrow C'}\ t) \quad &= \quad (\Uparrow(t \,/\, C')) \bullet C \\
&\qquad \text{(or FAIL if } t \,/\, C' \text{ does not match)} \\
\Uparrow \mathtt{node}_a(n, \overrightarrow{t_i}) \quad &= \quad \text{FAIL} \\
\Uparrow t \quad &= \quad t \quad \text{otherwise}
\end{aligned}
$$

Figure 8. Resugaring

To show this, we will rely on terms having *honest tags*:

Definition 3. A term has *honest tags* when for each subterm of the form $\mathtt{tag}_{C \Rightarrow C'}\ t$, $t = (t \,/\, C') \bullet C'$.

Lemma 5. *For all terms t with honest tags, if $\Uparrow t \neq$ FAIL then $\Downarrow \Uparrow t \simeq t$.*

Proof Sketch. Proceed by induction on t. The interesting case is when the term t is tagged:

$$
\begin{aligned}
\Downarrow \Uparrow \mathtt{tag}_{C \Rightarrow C'}\ t \quad &= \quad \Downarrow((\Uparrow(t \,/\, C')) \bullet C) \\
&\qquad \text{with } (C, C') = \textit{expand}(t') \text{ for some } t' \\
&= \quad \mathtt{tag}_{C \Rightarrow C''} \Downarrow(((\Uparrow(t \,/\, C')) \bullet C) \,/\, C) \bullet C'' \\
&\qquad \text{where } \textit{expand}(\Uparrow(t \,/\, C')) = (C, C'') \\
&\qquad \text{and } C' \simeq C'' \\
&= \quad \mathtt{tag}_{C \Rightarrow C''} \Downarrow \Uparrow (t \,/\, C') \bullet C'' \\
&\qquad \text{by Lemma 3} \\
&\simeq \quad \mathtt{tag}_{C \Rightarrow C''}\ (t \,/\, C') \bullet C'' \\
&\qquad \text{by I.H.} \\
&\simeq \quad \mathtt{tag}_{C \Rightarrow C'}\ t \\
&\qquad \text{by Lemma 4}
\end{aligned}
$$

The first step (which introduces t') relies on the tags having been produced by a call to *expand*. ☐

Lemma 6. *For all terms t, $\Uparrow \Downarrow t = t$.*

Proof. Proceed by induction on t. The interesting case is where the term t is not atomic:

$$\begin{array}{rcll} \Uparrow\Downarrow t & = & \Uparrow \mathtt{tag}_{C\Rightarrow C'}\ (\Downarrow (t\,/\,C) \bullet C') & \\ & & \text{with } \mathit{expand}(t) = (C, C') & \\ & = & \Uparrow ((\Downarrow (t\,/\,C) \bullet C')\,/\,C') \bullet C & \\ & = & \Uparrow\Downarrow (t\,/\,C) \bullet C & \text{by Lemma 3} \\ & = & (t\,/\,C) \bullet C & \text{by I.H.} \\ & = & t & \text{by Lemma 4} \end{array}$$

(The side condition for Lemma 3 uses the fact that $\mathit{domain}(t\,/\,C) = \mathit{holes}(C) = \mathit{holes}(C')$.) □

Lemma 7. *For all terms t,* $\mathcal{R}(\Uparrow\mathcal{R}(t)) \simeq \Uparrow\mathcal{R}(t)$

Proof. The witness permutation is the mapping the second $\mathcal{R}$ enacts on variable declarations. This mapping exists since resugaring can neither drop nor duplicate variables. Now we must show that variable references are acted upon by the second $\mathcal{R}$ the same way as their corresponding declarations. This amounts to asking weather each variable reference $\mathtt{ref}(x_u)$ is in scope of exactly its declaration $\mathtt{decl}(x_u)$. It is: it cannot be in scope of any *other* declaration, because the *first* call to $\mathcal{R}$ gave them all distinct subscripts, and it cannot be *out* of scope of its $\mathtt{decl}(x_u)$ because that would mean that resugaring caused an identifier to become unbound, which could only happen if the initial program contained an unbound identifier. □

The previous paper on resugaring gave three properties that help define its correctness. We mirror them here.

The first property, Emulation, says that the resugared sequence is faithful to the core sequence it is supposed to represent.

Theorem 1 (Emulation)**.** *Every surface term desugars to (a term isomorphic to) the core term it purports to represent.*

Proof. We want to show that if a surface term $t' = \mathit{resugar}(t)$ is shown, then $\mathit{desugar}(t') \simeq \mathcal{R}(t)$.

$$\begin{array}{rcll} \mathit{desugar}(t') & = & \mathit{desugar}(\mathit{resugar}(t)) & \\ & = & \Downarrow(\mathcal{R}(\mathcal{U}(\Uparrow(\mathcal{R}(t))))) & \\ & \simeq & \Downarrow(\mathcal{R}(\mathcal{U}(\mathcal{R}(\Uparrow(\mathcal{R}(t)))))) & \text{by Lemma 7} \\ & \simeq & \Downarrow(\mathcal{R}(\Uparrow(\mathcal{R}(t)))) & \text{by Lemma 1} \\ & \simeq & \Downarrow(\Uparrow(\mathcal{R}(t))) & \text{by Lemma 7} \\ & \simeq & \mathcal{R}(t) & \text{by Lemma 5} \end{array}$$

□

The second property, Abstraction, says that surface terms are not "made up", but rather originate from the initial program. We give a stronger statement about Abstraction here than was given in the previous work; this is possible because nodes have identity.

Theorem 2 (Abstraction)**.** *If a term is shown in the reconstructed surface evaluation sequence, then each non-atomic part of it originated from the original program and has honest tags. (Assuming that evaluation does not modify tags.)*

Proof. Let $\mathcal{R}(t)$ be the original program, let $t_0 = \Downarrow\mathcal{R}(t)$, and suppose the program took i steps $t_0 \to \ldots \to t_i$ before being shown as $t'_i = \Uparrow t_i$. For resugaring to have succeeded, t_i must be composed from patterns of the form $\mathtt{tag}_{C\Rightarrow C'}\ C'$ (implying that the tags of t are honest). After resugaring, the atomic terms are left as they are, and each pattern $\mathtt{tag}_{C\Rightarrow C'}\ C'$ becomes C. Likewise, each pattern $\mathtt{tag}_{C\Rightarrow C'}\ C'$ can be traced back through evaluation to the desugaring of the original program, so $\mathtt{tag}_{C\Rightarrow C'}\ C'$ appears in t_0 and C appears in $\mathcal{R}(t)$. □

The third property, Coverage, says that "as many surface evaluation steps are shown as possible". It was dealt with purely informally in the previous paper, but now we formally give in Section 5 a sufficient condition for surface steps to be shown.

4.3 Hygiene

Finally, we can show that desugaring and resugaring are hygienic in the sense put forward by Herman and Wand [7]. They proposed the strong statement that if two terms in the surface language are α-equivalent, then their desugarings are α-equivalent; we will prove this for our system.

Recall that we define $s =_\alpha t$ to mean that $\mathcal{R}(s) \simeq \mathcal{R}(t)$; thus the question of whether a function respects α-equivalence can sometimes be reduced to one of whether it is *equivariant*: whether it respects terms that only differ up to a permutation of their variables. (Equivariance is a concept from Nominal Logic [5].) $\Downarrow$ and $\Uparrow$ are equivariant.

Lemma 8 ($\Downarrow$ is equivariant)**.** *If* $s \simeq t$*, then* $\Downarrow s \simeq \Downarrow t$*.*

Proof sketch. head is equivariant, and $\mathit{sugars}(n)$ is trivially equivariant when applied to the patterns obtained from *head* since they do not contain variables. Thus *expand* is equivariant, in the sense that if $s \simeq t$ and $\mathit{expand}(s) = (C_s, C'_s)$ and $\mathit{expand}(t) = (C_t, C'_t)$, then $\exists\sigma^*,\ C_s = \sigma^* \bullet C_t$ and $C'_s = \sigma^* \bullet C'_t$. To show that $\Downarrow$ is equivariant, we have to show that for all σ and t, $\Downarrow(\sigma \bullet t) = \sigma' \bullet \Downarrow t$ for some σ'.

If t is not a $\mathtt{node}$, $\Downarrow(\sigma \bullet t) = \sigma \bullet t = \sigma \bullet \Downarrow t$. Otherwise, let $\mathit{expand}(t) = (C, C')$ and $\mathit{expand}(\sigma \bullet t) = (\sigma^* \bullet C, \sigma^* \bullet C')$. Then:

$$\begin{array}{rcl} \Downarrow(\sigma \bullet t) & = & \mathtt{tag}_{\sigma^*\bullet C\Rightarrow\sigma^*\bullet C'}\ (((\sigma\bullet t)\,/\,(\sigma^*\bullet C)) \bullet (\sigma^*\bullet C')) \\ & = & \mathtt{tag}_{\sigma^*\bullet C\Rightarrow\sigma^*\bullet C'}\ \sigma^* \bullet ((t\,/\,C) \bullet C') \\ & = & \sigma^* \bullet \mathtt{tag}_{C\Rightarrow C'}\ ((t\,/\,C)\bullet C') \\ & = & \sigma^* \bullet \Downarrow t \end{array}$$

(The second step uses the fact that σ and σ^* must be identical when restricted to the variables of C.) □

Lemma 9 ($\Uparrow$ is equivariant)**.** *If* $s \simeq t$*, then* $\Uparrow s \simeq \Uparrow t$*.*

Proof. It suffices to show that $\Uparrow(\sigma \bullet s) = \sigma \bullet \Uparrow s$ for all s and σ. Induct on s; in the inductive case $s = \mathtt{tag}_{C\Rightarrow C'}\ t$:

$$\begin{array}{rcll} \Uparrow(\sigma \bullet s) & = & \Uparrow \mathtt{tag}_{\sigma\bullet C\Rightarrow\sigma\bullet C'}\ \sigma \bullet t & \\ & = & (\Uparrow(\sigma \bullet t\,/\,\sigma \bullet C')) \bullet (\sigma \bullet C) & \\ & = & (\sigma \bullet \Uparrow(t\,/\,C')) \bullet (\sigma \bullet C) & \text{(by I.H.)} \\ & = & \sigma \bullet ((\Uparrow(t\,/\,C')) \bullet C) & \\ & = & \sigma \bullet \Uparrow \mathtt{tag}_{C\Rightarrow C'}\ t & \\ & = & \sigma \bullet \Uparrow s & \end{array}$$

□

While $\mathcal{U}$ is not equivariant (for example, it transforms $(\lambda x_1.\, x_1)(\lambda x_2.\, x_2)$ into $(\lambda x.\, x)(\lambda x.\, x)$), it does respect α-equivalence.

Lemma 10 ($\mathcal{U}$ respects α-equivalence of resolved terms)**.** *If* $s =_\alpha t$ *and* $s = \mathcal{R}(s')$ *and* $t = \mathcal{R}(t')$*, then* $\mathcal{U}(s) =_\alpha \mathcal{U}(t)$*.*

Proof. By the definition of $(=_\alpha)$, we want to show that $\mathcal{R}(\mathcal{U}(s)) \simeq \mathcal{R}(\mathcal{U}(t))$, knowing just that $\mathcal{R}(s) \simeq \mathcal{R}(t)$. First, use Lemma 1 to see that:

$$\mathcal{R}(\mathcal{U}(s)) = \mathcal{R}(\mathcal{U}(\mathcal{R}(s'))) \simeq \mathcal{R}(s') = s$$
$$\mathcal{R}(\mathcal{U}(t)) = \mathcal{R}(\mathcal{U}(\mathcal{R}(t'))) \simeq \mathcal{R}(t') = t$$

Thus, $\mathcal{R}(\mathcal{U}(s)) \simeq \mathcal{R}(\mathcal{R}(\mathcal{U}(s))) \simeq \mathcal{R}(s) \simeq \mathcal{R}(t) \simeq \mathcal{R}(\mathcal{R}(\mathcal{U}(t))) \simeq \mathcal{R}(\mathcal{U}(t))$ (using the fact that $\mathcal{R}(\mathcal{R}(t)) = t$ for all terms t). □

Theorem 3 (Hygiene). *If $s =_\alpha t$ then desugar$(s) =_\alpha$ desugar(t). Likewise, if $s =_\alpha t$ then resugar$(s) =_\alpha$ resugar(t).*

Proof.

$$\textit{desugar}(s) = \Downarrow(\mathcal{R}(s)) \simeq \Downarrow(\mathcal{R}(t)) = \textit{desugar}(t)$$

The middle step is valid because $\mathcal{R}(s) \simeq \mathcal{R}(t)$ by the assumption that $s =_\alpha t$ and because $\Downarrow$ is equivariant by Lemma 8.

$$\begin{array}{rcll} \textit{resugar}(s) & = & \mathcal{U}(\Uparrow(\mathcal{R}(s))) & \\ & \simeq & \mathcal{U}(\mathcal{R}(\Uparrow(\mathcal{R}(s)))) & \text{by Lemma 7} \\ & \simeq & \Uparrow(\mathcal{R}(s)) & \text{by Lemma 1} \\ & \simeq & \Uparrow(\mathcal{R}(t)) & \\ & \simeq & \mathcal{U}(\mathcal{R}(\Uparrow(\mathcal{R}(t)))) & \text{by Lemma 1} \\ & \simeq & \mathcal{U}(\Uparrow(\mathcal{R}(t))) & \text{by Lemma 7} \\ & = & \textit{resugar}(t) & \end{array}$$

□

5. From Individual Terms to Evaluation Sequences

We have proved three properties about resugaring: Emulation, Abstraction, and hygiene. All three of these properties, however, only talk about *individual terms*, not entire evaluation sequences. In particular, not every core step will be resugared to a surface evaluation step; sometime a core term cannot be resugared so the corresponding surface step will be skipped. Recall the final example (column 3) in Fig. 1. The third core evaluation step (where the outer `if` is evaluated away) is skipped. We can now better justify it being skipped: showing a surface term for it would violate Abstraction, since this term does not have honest tags — the tag claims that the `if` node has one identity (which originated from sugar), while it actually has another (which originated from user code).

Here is the full core evaluation sequence. We omit a couple of evaluation steps where a constant simplifies to a value, such as `VERBOSE` → `true`. We also omit node subscripts, since they won't be relevant to the discussion:

```
let port = 80 in
  [C1 ⇒ C2]
  let port = if true then
               (if VERBOSE
                then STDERR
                else DEVNULL)
               else DEVNULL in
    write("Port: " + to_str(•), •)
          ↓
[C1 ⇒ C2]
let port = if true then
             (if VERBOSE
              then STDERR
              else DEVNULL)
             else DEVNULL in
  write("Port: " + to_str(80), •)
          ↓

[C1 ⇒ C2]
let port = if VERBOSE
             then STDERR
             else DEVNULL in
  write("Port: " + to_str(80), •)
          ↓
[C1 ⇒ C2]
let port = STDERR in
  write("Port: " + to_str(80), •)
          ↓
write("Port: " + to_str(80), STDERR)
          ↓
write("Port: " + "80", STDERR)
          ↓
write("Port: 80", STDERR)
          ↓
void
```

The surface evaluation sequence, however, is much more sparse:

```
let port = 80 in
  log "Port: " + to_str(•)
  to (if VERBOSE then STDERR else DEVNULL)
  when true
          ↓
log "Port: " + to_str(80)
to (if VERBOSE then STDERR else DEVNULL)
when true
          ↓
void
```

We have argued that it is good that the third core evaluation step was not resugared. But it should be worrisome that all the other steps were skipped as well. It would be nice, for instance, to show evaluation steps for the string being logged:

```
log "Port: " + to_str(80)
to (if VERBOSE then STDERR else DEVNULL)
when true
          ↓
log "Port: " + "80"
to (if VERBOSE then STDERR else DEVNULL)
when true
          ↓
log "Port: 80"
to (if VERBOSE then STDERR else DEVNULL)
when true
```

These steps were not shown, however, since it would break the Emulation property. Since the sugar's `let` has been substituted away by the time these string operations are performed, these hypothetical surface steps would not desugar into the actual core evaluation steps.

Fortunately, the `log` sugar can be refactored to show these steps, simply by let-binding the message to be printed:

```
log α to β when γ
          ⇓
let msg = α in
  let port = if γ then β else DEVNULL in
    write("Port: ", msg, port)
```

After this change, the reductions for the message argument to `log` are shown. We will not show the entire evaluation sequence, but one of the core steps is:

```
[C1 ⇒ C2]
let msg = "Port: " + "80" in
  let port =
      if VERBOSE then STDERR else DEVNULL in
    write(msg, •)
```

which gets resugared to the surface term:

```
log "Port: " + "80"
to (if VERBOSE then STDERR else DEVNULL)
when true
```

While this particular instance of calling the `log` sugar shows nice surface steps, the fact that the sugar had to be rewritten begs the question of whether it must in all cases. We will use the phrase *coverage* to talk about the number of steps a sugar shows: a sugar with good coverage shows many steps in the reconstructed surface evaluation sequence. In this section, we introduce theory to help show when this is the case. For this particular sugar, we will be able to apply the general theory to show that, whenever $\alpha \to \alpha'$,

$$\texttt{log } \alpha \texttt{ to } \beta \texttt{ when } \gamma \to \texttt{log } \alpha' \texttt{ to } \beta \texttt{ when } \gamma$$

Towards this end, we will first talk about evaluation contexts (a traditional concept) and non-evaluation contexts (a new concept), then state a general *Coverage* theorem, and then show how that theorem can be applied in this case.

Terminology Switch To better match typical terminology, we will now switch to calling patterns C as *contexts*, and write the substitution of $\alpha_1 \to t_1, ..., \alpha_k \to t_k$ into the context C as $C[t_1, ..., t_k]$.

5.1 Evaluation Contexts and Non-evaluation Contexts

Evaluation contexts [4] are contexts of a single hole obeying certain syntactic criteria. In our setting, it is possible that the terms plugged into the evaluation context's hole depend on the evaluation context; hence we will instead work with *enclosing evaluation contexts* $E, t_1, ..., t_k$, where E is an evaluation context and $t_1, ..., t_k$ are terms (that may depend on E and each other). Evaluation contexts typically enjoy the following properties, which we will make use of:

Step If $E[t]$ takes a step, then $E[t] \to E[t']$ for some t'.

Composition If E_1 and E_2 are evaluation contexts, then so is $E_1[E_2]$.

Independence If $E[\alpha, t_1, ..., t_k]$ is an evaluation context over α and $E[t, t_1, ..., t_k] \to E[t', t_1, ..., t_k]$, and $E[\alpha, t'_1, ..., t'_k]$ is also an evaluation context over α, then $E[t, t'_1, ..., t'_k] \to E[t', t'_1, ..., t'_k]$. (In other words, the reduction of a redex does not depend on things outside of it, except insofar as they may cause the redex to be located elsewhere.)

In our running example, for any terms t_1 and t_2, the context $E[\alpha]$ defined by:

```
let msg = α in
  let port = if t₁ then t₂ else DEVNULL in
    write("Port: ", msg, port)
```

is an evaluation context.

To state the Coverage theorem, we will need a related but new concept, called a *non-evaluation context*. A non-evaluation context is the opposite of an evaluation context: its redex (the next subterm within it to be reduced) is *outside* of its holes. Using the same example, for any term t that can take a step, the context $C[\beta, \gamma]$ defined by:

```
let msg = t in
  let port = if β then γ else DEVNULL in
    write("Port: ", msg, port)
```

is a non-evaluation context. In general, a non-evaluation context is a context $C[\alpha_1, ..., \alpha_n]$ that can be written as $C'[t, \alpha_1, ..., \alpha_n]$ where for all $t_1, ..., t_k$, $C'[\alpha, t_1, ..., t_n]$ is an evaluation context over α.

5.2 Evaluation Steps for Non-evaluation Contexts

The Coverage theorem we will use to prove that a sugar will show certain steps will be built up in two parts. First, we will lift the notion of evaluation to apply to non-evaluation contexts, so that it makes sense not only to talk about a term taking a step $t \to t'$, but also of a non-evaluation context C taking a step $C \to C'$. The Coverage theorem will then lift this notion to surface terms as well.

Lemma 11. *Let C be a non-evaluation context.*

$$\begin{array}{lll} \textit{If} & \exists E, t_1, ..., t_k. & E[C[t_1, ..., t_k]] \to_{core} E[C'[t_1, ..., t_k]] \\ \textit{then} & \forall E, t_1, ..., t_k. & E[C[t_1, ..., t_k]] \to_{core} E[C'[t_1, ..., t_k]] \end{array}$$

Proof. Let $E, t_1, ..., t_k$ be the existentially quantified variables and $E', t'_1, ..., t'_k$ be the universally quantified ones. By the definition of non-evaluation contexts, $C[\alpha_1, ..., \alpha_k] = E^*[t^*, \alpha_1, ..., \alpha_k]$ (for some E^*, t^*, where E^* is an evaluation context over its first hole). By the composition property above, $E[E^*]$ and $E'[E^*]$ are evaluation contexts. By the step property, there exists a term t^{**} such that:

$$\begin{array}{rcl} E[C[t_1, ..., t_k]] & = & E[E^*[t^*, t_1, ..., t_k]] \\ & \to & E[E^*[t^{**}, t_1, ..., t_k]] \\ & = & E[C'[t_1, ..., t_k]] \end{array}$$

and by the independence property,

$$\begin{array}{rcl} E'[C[t'_1, ..., t'_k]] & = & E'[E^*[t^*, t'_1, ..., t'_k]] \\ & \to & E'[E^*[t^{**}, t'_1, ..., t'_k]] \\ & = & E'[C'[t_1, ..., t_k]] \end{array}$$

□

When this holds, we will say that $C \to C'$, and when $C_1 \to C_2 \to ... \to C_n$, we will say that $C_1 \to^* C_n$. Thus we can talk about evaluation steps for non-evaluation contexts in the core language.

Finally, we can state the Coverage theorem that generalizes the previous lemma to also work on surface terms that must be desugared before being evaluated ($\to^*_{core}$ refers to actual evaluation steps in the core language, and $\to^*_{surf}$ refers to reconstructed evaluation steps in the surface language):

Theorem 4 (Coverage)**.** *If desugar$(C) \to^*_{core}$ desugar(C'), then* $\forall E, t_1, ..., t_k$, $E[C[t_1, ..., t_k]] \to^*_{surf} E[C'[t_1, ..., t_k]]$

Proof. We just have to show that $\Downarrow (E[C[t_1, ..., t_k]]) \to^*_{core} \Downarrow (E[C'[t_1, ..., t_k]])$, given the hypothesis. Using the above lemma and the fact that desugaring is compositional:

$$\begin{array}{rcl} \Downarrow (E[C[t_1, ..., t_k]]) & = & \Downarrow E[\Downarrow C[\Downarrow t_1, ..., \Downarrow t_k]] \\ & \to^*_{core} & \Downarrow E[\Downarrow C'[\Downarrow t_1, ..., \Downarrow t_k]] \\ & = & \Downarrow (E[C'[t_1, ..., t_k]]) \end{array}$$

□

Similarly to the previous lemma, when this holds, we will say that $C \to_{surf} C'$, and when $C_1 \to_{surf} C_2 \to_{surf} ... \to_{surf} C_n$, we will say that $C_1 \to^*_{surf} C_n$. Thus we can talk about evaluation steps for non-evaluation contexts in the *surface* language.

Let us illustrate this theorem with our `log` example. Suppose that $t \to t'$ for some terms t and t'. Since the context $E[\alpha]$ given by

```
let msg = α in
  let port = if t₁ then t₂ else DEVNULL in
    write("Port: ", msg, port)
```

is an evaluation context, we know that $E[t] \to E[t']$.

Next, define $C_{core}[\beta, \gamma]$ to be the non-evaluation context given by:

```
let msg = t in
  let port = if β then γ else DEVNULL in
    write(msg, port)
```

and $C'_{core}[\beta, \gamma]$ to be the non-evaluation context:

```
let msg = t' in
  let port = if β then γ else DEVNULL in
    write(msg, port)
```

Likewise, define $C_{surf}[\beta, \gamma]$ to be a context in the surface language that desugars to C_{core}:

`log "Port: " +` t `to` β `when` γ

and C'_{surf} to be the surface context with t':

`log "Port: " +` t' `to` β `when` γ

By Lemma 11, $C_{core} \rightarrow C'_{core}$. And by the coverage theorem, using the fact that $\Downarrow C_{surf} = C_{core}$ and $\Downarrow C'_{surf} = C'_{core}$, we learn that for all β and γ,

`log` t `to` β `when` $\gamma \rightarrow$ `log` t' `to` β `when` γ

6. Implementation

We have implemented a prototype of this system and tested it on a simple language. Implementing this system for a real language in the wild requires the same effort as that discussed in previous work [13, section 7]. In particular, a core evaluation sequence needs to be obtained; this sequence is the starting point for resugaring (which attempts to resugar each core term). This can be obtained by instrumenting the evaluator, or by modifying the program before evaluating it. Any system that works by syntactic rewriting and exposes intermediate syntactic terms—such as some theorem provers and term-rewriting systems—would be even easier to adapt to work with our resugarer, so long as it is amenable to representing terms as ASDs.

7. Related Work

There is a long history of trying to relate compiled code back to its source. This problem is especially pronounced in debuggers for optimizing compilers [6]. The previous work on resugaring [13] describes these in more detail and explains why they address a strictly weaker problem (relating locations rather than reconstructing terms, and not providing semantic guarantees); the same relationship applies to our work. Compared to the previous resugaring work, we have discussed the use of ASDs and scope resolution in order to (i) achieve hygiene, and (ii) give stronger formal properties: see Coverage in Theorem 4 and Abstraction in Theorem 2.

Van Deursen et al. [17] formalize the concept of tracking the origins of terms within term rewriting systems (which in their case represent the *evaluator*, not the *syntactic sugar* as in our case). They go on to show various applications, including visualizing program execution, implementing debugger breakpoints, and locating the sources of errors. Their work does not involve the use of syntactic sugar, however, while our work hinges on the interplay between syntactic sugar and evaluation. Nevertheless, we have adopted their notion of origin tracking for our transformations.

We now list several related works that served as inspiration for or are related to our work, or could be used in place of some of our components. None of these, however, actually offers resugaring, which is our principal focus.

Specifying Binding Structure There is a plethora of languages for specifying the binding structure for a programming language. We choose the binding algebra of Romeo [16] because it is powerful enough to specify, e.g. `let`, `let*`, and `letrec`, while still being strongly compositional in a way that allows our $\mathcal{R}$ and $\mathcal{U}$ operations to have a simple inductive definition. There are, however, many other binding specification languages of equal merit. Binding specification in the Ott semantic engineering tool [14] is very similar to Romeo's. Likewise, Weirich et al. give a set of binding combinators in Haskell of similar power [18].

Neron et al. [11] introduce *scope graphs* as a formal representation for binding structure. Scope graphs are more powerful than other binding structure representations in that they handle module scope. While scope graphs represent binding structure, however, they do not specify how to obtain it (a crucial requirement for our use): this is left for other systems such as the group's previous NaBL name binding language [9]. While NaBL itself lacks expressive power—it cannot describe the binding structure of, e.g., `let*`—we believe our work could be adapted to work with scope graphs on top of a different binding declaration language.

In contrast to these efforts, the typed HOAS [12] and PHOAS [2] efforts are excellent *representations* of abstract syntax, but do not say how to *construct* that syntax in a language-agnostic way. We therefore believe it would take much more effort to utilize them for scope resolution. Nevertheless, our work is largely agnostic to the differences between these systems so long as they can satisfy the core needs of scope resolution: taking a surface term and the scoping rules for the surface language and assigning fresh subscripts to all variable declarations.

Hygienic Transformations A detailed comparison of our approach to hygiene against traditional hygienic algorithms is given in Section 3.

Traditional approaches to hygiene suffered from an inability to formally state a general specification for hygiene. The difficulty is that the real goal for hygiene is for macros (or syntactic sugar) to preserve α-equivalence, but α-equivalence is typically only *defined* for the core language. Thus Herman and Wand advocate that macros specify the binding structure of the constructs they introduce, and build a system that does so [7]. Romeo follows in these footsteps with a more powerful system. We use Romeo's binding algebra to specify surface language α-equivalence, thus allowing the direct statement of hygiene in Theorem 3: desugaring (and resugaring) preserve α-equivalence.

An interesting alternative approach is put forward by Erdweg et al. with the *name-fix* algorithm [3]. *name-fix* also makes use of scope resolution, albeit in a different way than we do. Instead of using scope resolution to *avoid* capture in the first place, *name-fix* uses it to *detect* capture and rename variables as necessary to repair it after the fact. Both *name-fix* and our system assume that nodes have identity, but we make the additional assumption that variables have subscripts that can be set by the resolution algorithm. We also give a general algorithm for resolving scope given scoping rules for a language, whereas *name-fix* assumes the resolution function is provided to it.

A recent piece of work on hygienic transformations by Adams [1] advances the theory of hygiene by giving a relatively algorithm-independent notion of hygiene, and using it to derive an elegant hygienic transformer. We are able to show a more direct definition of hygiene (preserving α-equivalence), in exchange for requiring the scope of the surface language to be declared, which Adams avoids in keeping with the hygiene tradition.

Acknowledgments

We thank Eelco Visser, Sebastian Erdweg, Paul Stansifer, and our anonymous reviewers for their feedback. We especially thank Pierre Neron for his extremely detailed comments. This work was partially supported by the US National Science Foundation.

References

[1] M. D. Adams. Towards the essence of hygiene. In *Principles of Programming Languages*, 2015.

[2] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, 2008.

[3] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *European Conference on Object-Oriented Programming*, 2014.

[4] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2): 235–271, 1992.

[5] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2000.

[6] J. Hennessy. Symbolic debugging of optimized code. *Transactions on Programming Languages and Systems*, 4(3), 1982.

[7] D. Herman and M. Wand. A theory of hygienic macros. In *European Symposium on Programming Languages and Systems*, 2008.

[8] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming*, 1986.

[9] G. Konat, L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering*, 2012.

[10] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[11] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *European Symposium on Programming Languages and Systems*, 2015. To appear.

[12] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Programming Languages Design and Implementation*, 1988.

[13] J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Programming Languages Design and Implementation*, 2014.

[14] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. In *International Conference on Functional Programming*, 2007.

[15] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[16] P. Stansifer and M. Wand. Romeo: a system for more flexible binding-safe programming. In *International Conference on Functional Programming*, 2014.

[17] A. Van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5–6), 1993.

[18] S. Weirich, B. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, 2011.