Inferring Scope through Syntactic Sugar

JUSTIN POMBRIO, Brown University
SHRIRAM KRISHNAMURTHI, Brown University
MITCHELL WAND, Northeastern University

Many languages use syntactic sugar to define parts of their surface language in terms of a smaller core. Thus some properties of the surface language, like its *scoping rules*, are not immediately evident. Nevertheless, IDEs, refactorers, and other tools that traffic in source code depend on these rules to present information to users and to soundly perform their operations. In this paper, we show how to lift scoping rules defined on a core language to rules on the surface, a process of *scope inference*. In the process we introduce a new representation of binding structure—scope as a preorder—and present a theoretical advance: proving that a desugaring system preserves α-equivalence even though scoping rules have been provided *only* for the core language. We have also implemented the system presented in this paper.

CCS Concepts: • **Software and its engineering → Extensible languages**; *Macro languages*;

Additional Key Words and Phrases: Scope, binding, syntactic sugar

ACM Reference format:
Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring Scope through Syntactic Sugar. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 28 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Syntactic sugar is pervasive in language technology. Language designers use it to shrink the size of a surface language to a small core, to make it tractable for processing. In addition, programmers use it to define domain-specific languages, and—if their language provides macros or other meta-programming capabilities—even to extend their language. Thus, syntactic sugar is a valuable weapon in the programming arsenal.

Unfortunately, syntactic sugar also obscures the relationship between the user's source program and the program that is actually type-checked, analyzed, or evaluated. In particular, traditionally, scoping rules are defined on the core language, not on the surface. However, many tools depend on source representations. For instance, editors need to know the surface language's scoping in order to perform auto-complete, distinguish free from bound variables, or draw arrows to show bound and binding instances. Likewise, refactorers need to know binding structure to perform correct transformations. These tools become harder to construct if scoping is only known for the core language.

Many tools that exploit binding information for the source do so by desugaring the program and obtaining its binding in the core language (this, for instance, is the approach used by DrRacket [Findler et al. 2002] for overlaying binding arrows on the source). However, this approach is far from ideal. It requires tools to be able to desugar programs and to resolve binding in the core language. This is an intimate level of knowledge of a language, though: syntactic sugar is supposed to be an abstraction, so external tools should ideally be unaware that a language even *has* syntactic sugar. Additionally, this approach fails completely if the source program cannot be desugared because it is incomplete or syntactically invalid (as programs are most of the time while

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

editing). It is therefore better to disentangle the editor from the language, providing the editor precisely what it needs: scoping rules for the surface language.

We therefore present a static inference process that, given a specification of syntactic sugar[1] and scoping rules on a core language, *automatically constructs* scoping rules for the surface language. The inferred rules are guaranteed to give the same binding structure to a surface program as that program would have in the core language after desugaring (theorem 5.4). Essentially, scope inference "pushes scope back through the sugar". We can think of this as statically lifting a "lightweight semantics" of the language. Thus it is a precursor to lifting other notions of semantics (whether type-checking rules or the evaluation rules themselves), though of course the mechanics of doing so will depend heavily on the semantics itself.

The intended application of this work is as follows:

(1) Begin with a core language with known scoping rules, and a set of pattern-based desugaring rules. (We give a formal description of scope in section 3, and a language for specifying scope in section 4.)
(2) Infer surface language scoping rules from the core scoping rules. (We give a scope inference algorithm in section 5, and show how to make it hygienic in section 7.)
(3) Add these inferred scoping rules to various tools that can exploit them (Sublime, Atom, CodeMirror, etc.).

An alternative approach would be to specify scoping rules for the surface language, and verify that they are consistent with the core language. This approach has been advocated for scoping [Herman and Wand 2008; Stansifer and Wand 2014], type systems [Lorenzen and Erdweg 2013], and formal semantics [Fisher and Shivers 2006]. However, this assumes that language developers are always programming language experts who are knowledgeable about binding, able to verify consistency, and willing to do this extra work. These are particularly unsafe assumptions for domain-specific languages, which we believe are a strong use case for our technique.

Contributions and Outline

Modelling Scope In section 3, we give a formal description of scope as a *preorder* (which we motivate through examples in section 2). This preorder then defines the name binding structure of a program, such as where variable references are bound, and which variable declarations shadow others.

Binding Specification Language In section 4, we present a *binding specification language*, i.e., a language for specifying the name binding structure of a programming language. This specification makes it possible to compute the scope structure (a preorder) of concrete programs in that language.

Scope Inference In section 5, we show how to *infer* these scoping rules through syntactic sugar. This is our main contribution. We describe our implementation and provide case studies in section 6, and prove that—given reasonable assumptions—desugaring after scope inference will be hygienic in section 7.

Due to space limitations, we omit some proofs and lemmas. They can be found in the supplement (http://cs.brown.edu/research/plt/dl/icfp2017/), which contains an extended version of the paper. To reduce the burden on readers, we have aligned lemma and definition numbers between the two versions: thus some lemma numbers are skipped here.

[1] We make no assumption that the core language is a subset of the surface language, so our results can be applied to, e.g., EDSLs.

2 TWO WORKED EXAMPLES

We will begin by building up to our scope inference technique via two worked examples. **They are slightly simplified for expository purposes**. Section 4 describes the generalization, and sections 6.1-6.3 provide examples. (While the generalization is sometimes important, it has no effect on the examples of this section.)

2.1 Example: Single-arm Let

For the first example, consider a simple Let construct that allows only a single binding:

$$t \quad ::= \quad (\texttt{Let}\ x^{\mathbf{D}}\ t_1\ t_2) \quad \text{“Let } x^{\mathbf{D}} \text{ equal } t_1 \text{ in } t_2\text{”}$$
$$\mid \quad \ldots$$

(Here the superscript D indicates that this occurrence of the variable x is a declaration of x.) In general, we will distinguish *declarations*, i.e., binding sites, from *references*, i.e., use sites.)

This Let may be desugared to Apply and Lambda by the following desugaring rule, which we will write using s-expressions:

$$(\texttt{Let}\ \bullet_1\ \bullet_2\ \bullet_3) \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ \bullet_1\ \bullet_3)\ \bullet_2)$$

Now suppose that we know the scoping rules of Apply and Lambda, and wish to derive what the scoping rules for Let must be, given the desugaring rule and assuming the language is statically and lexically scoped. More precisely, we wish to find a scoping rule for Let such that the desugaring rules *preserve binding structure* (and thus neither cause variable capture nor cause variables to become unbound).

The first step will be to write down what we know about the scope on the RHS (right hand side) of the rule. Pictorially, we might draw:

where the dotted lines show the tree structure of the AST, and where the teal/solid arrow means that the Lambda's parameter ($\bullet_1$) can be used in its body ($\bullet_3$). Similarly, there are no arrows among the children of Apply because function application does not introduce any binding.

We also know from lexical scope that any declarations in scope at a node in an AST should also be in scope at its children. This can be denoted with upward arrows:

In general, the meaning of the arrows is that a variable declaration is in scope at every part of the program which has a (directed) path to it. (In the case of variable shadowing, the outer declaration is in scope at the inner declaration, which in turn is in scope at some region; references in this region will be bound to the dominating inner declaration.)

Now we can begin to infer what the scope must look like on the LHS (left hand side) of the desugaring rule. We want the rule to *preserve* binding, therefore there should be a path from one hole to another in the LHS iff there is a similar path in the RHS. If there was a path from $\bullet_1$ to $\bullet_2$ in

the LHS but not in the RHS, that would mean that a variable (in $\bullet_1$) that used to be bound (by $\bullet_2$) could become unbound. Likewise, if there was a path between two holes in the RHS but not in the LHS, that could result in unwanted variable capture.

Thus, since there is a path from $\bullet_3$ to $\bullet_1$ in the rule's RHS, there must also be a path from $\bullet_3$ to $\bullet_1$ in the LHS. This gives:

(LHS)

In English, this arrow says that the variable declared at $\bullet_1$ is in scope at the Let's body $\bullet_3$, as expected.

There are still some missing arrows, however: there should be down arrows to indicate that any declaration in scope at the Let should also be in scope at its children. These can be inferred in a similar way: whenever there is a path from the root to a hole on the RHS, there should be a similar path on the LHS. Since on the RHS there are paths to each hole from the root, the same should hold true on the LHS:

(LHS)

This gives a complete scoping rule for this Let construct.

2.2 Example: Multi-arm Let*

Next, take a more involved example: a multi-armed Let* construct (in the style of Lisp/Scheme/Racket). It will have the following grammar:

t	$::=$	`(Let*` b t`)`	"Let-bind b in t"
	$\mid$	...	
b	$::=$	`(Bind` $x^{\mathbf{D}}$ t b`)`	"Bind $x^{\mathbf{D}}$ to t, and bind b"
	$\mid$	`EndBinds`	"No more bindings"

This grammar separates out the Let's bindings into nested subterms.[2] It is necessary to do this if more complex binding patterns are allowed, such as arbitrarily deep pattern-matching.

Let* can then be implemented with two desugaring rules:

$$\texttt{(Let* (Bind } \bullet_1 \; \bullet_2 \; \bullet_3\texttt{)} \; \bullet_4\texttt{)}$$
$$\Rightarrow \texttt{(Apply (Lambda } \bullet_1 \; \texttt{(Let* } \bullet_3 \; \bullet_4\texttt{))} \; \bullet_2\texttt{)}$$

$$\texttt{(Let* EndBinds } \bullet_1\texttt{)} \Rightarrow \texttt{(Begin } \bullet_1\texttt{)}$$

These rules would, for example, make the following transformation:

$$\texttt{(Let* (Bind } x^{\mathbf{D}} \texttt{ 1 (Bind } y^{\mathbf{D}} \texttt{ 2 EndBinds)) (Plus } x^{\mathbf{R}} \; y^{\mathbf{R}}\texttt{))}$$
$$\Rightarrow \texttt{(Apply (Lambda } x^{\mathbf{D}}$$
$$\texttt{(Apply (Lambda } y^{\mathbf{D}} \texttt{ (Plus } x^{\mathbf{R}} \; y^{\mathbf{R}}\texttt{)) 2)) 1)}$$

Given that we know the scoping rules of Apply, Lambda, and Begin, we can use them to derive the scoping rules for Let* and Bind. The scoping for the second rule is trivial, so we will concentrate just on the first rule.

As before, the first step is to write down what we know about the scope on the RHS:

[2] We call the bindings just "Bind", even though they are specific to Let. If a language has other forms of binding as well, "Bind" may need a more specific name such as "LetBind".

Unlike in the previous example, this diagram is not (necessarily) complete, since we don't yet know the scoping rule for Let*.[3] We have drawn two upward arrows on Let*, despite the fact that we don't yet know its scoping rule: technically, these arrows should (and can) be inferred, but we start with them to simplify this example.

Now we can begin to infer what the scope must look like on the LHS. As before, we want the rule to preserve binding. Thus, since the RHS has a path from $\bullet_3$ to $\bullet_1$ and from $\bullet_4$ to $\bullet_1$, the same must be true in the LHS (labeling the arrows for reference):

Notice that we drew the path from $\bullet_4$ to $\bullet_1$ with *two* arrows. This is because we will assume that scoping rules are local, relating only terms and their immediate children.

We have now learned something about the scoping rules for Let* and Bind! When read in English, these three arrows say that:

a. Declarations from a Let*'s binding list are visible in its body.
b. A Bind's variable declaration is provided by the Bind (so that it can be used by the Let*).
c. A Bind's variable declaration is visible to later Binds in the binding list.

This information can now be applied to fill in the previously incomplete RHS picture. Arrow (a) represents a fact about the scoping of *every* Let*, so it must also apply in the RHS (highlighting it orange/dashed for exposition):

Adding this arrow introduces a path from $\bullet_4$ to $\bullet_3$, however, that needs to be reflected back at the LHS!

[3]This will happen when desugaring rules use recursion.

In general, the algorithm is to monotonically add arrows until reaching the least fixpoint. In this particular case, arrow *d* is the last fact to be inferred:

d. A Bind also provides any declarations provided by later Binds in the binding list.

This concludes the interesting facts to be inferred about the scoping rules for Let* and Bind. We have ignored the upward arrows that reflect lexical scope from parent to child for simplicity, but these can be inferred by the same process.

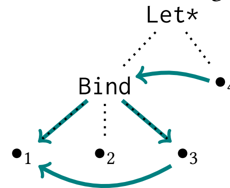
2.3 Scope as a Preorder

In the two preceding examples, we have expressed the scope of a program diagrammatically with arrows. When reasoning about scope, it will be helpful to be able to *transcribe* these diagrams into a textual form.

To do so, recall the (approximate) meaning of the arrows:[4] a declaration is in scope at every part of the program which has a (directed) path to it, and is shadowed by declarations of the same name that have a path to it. Thus the arrows are only meaningful insofar as they produce paths. Furthermore, paths have two important properties:

(1) They are closed under reflexivity: there is always an (empty) path from a to a.
(2) They are closed under transitivity: if there is a path from a to b and a path from b to c, then there is a path from a to c.

These are also precisely the properties that define a *preorder*. Thus, we will transcribe scope diagrams as preorders, writing $a \leqslant b$ when there is a path from a to b. For example, in the (incomplete) diagram we inferred for the LHS of the multi-arm Let* sugar:



The corresponding preorder is: $\bullet_4 \leqslant \texttt{Bind} \leqslant \bullet_3 \leqslant \bullet_1$

3 DESCRIBING SCOPE AS A PREORDER

We have informally described the notion of scope as a preorder, primarily using diagrams. In this section, we will describe it formally. First, however, we need to lay down some starting assumptions and basic definitions.

3.1 Basic Assumptions

We will make a number of assumptions to make reasoning about scope more tractable:

- We only deal with scoping that is *static* and *lexical*.

[4] We make their meaning precise in section 3.2.

- Scoping rules will be as local as possible, only relating a term to its *immediate* children. Longer relationships will be achieved by transitivity.[5]
- We work on an AST, instead of directly on the surface syntax. Variable references (use sites) and declarations (binding sites) are syntactically distinguished in this AST.
- Each kind of term has a fixed arity. (Indefinite arity is possible using a list encoding, as in Let* above.)

The last two of these assumptions guide our definition of (AST) terms. Terms will be parameterized over a set of *term constructors* $\mathcal{T}$, each with an *arity* : $\mathcal{T} \to \mathbb{N}$, and also over a set of *syntactic constants* C. We will write the AST in s-expression form:

$$
\begin{array}{lll}
t ::= & x^{\mathbf{D}}_{\mathbf{i}} & \text{(declaration at position } i\text{)} \\
\mid & x^{\mathbf{R}}_{\mathbf{i}} & \text{(reference at position } i\text{)} \\
\mid & (P\ t_1 \ldots t_n) & \text{(AST node)} \\
 & \text{where } P \in \mathcal{T} & \\
 & \text{and } \mathit{arity}(P) = n & \\
\mid & \mathit{const} & \text{(syntactic constant)} \\
 & \text{where } \mathit{const} \in C &
\end{array}
$$

References and declarations have both a name x (as written in the source), and an AST position i (that uniquely distinguishes them). Occasionally it will be useful to refer to a variable which could be either a declaration or a reference: in this case we omit the superscript, e.g. $x_{\mathbf{i}}$. Likewise, we will omit the position subscript $\mathbf{i}$ when there is no ambiguity. We will also sometimes write P in place of $(P\ t_1 \ldots t_n)$ when there is no danger of ambiguity.

3.2 Basic Definitions

We define scope in terms of a perorder. A *preorder* $(\leqslant)$ is a reflexive and transitive relation. It need not be anti-symmetric, however, so it is possible that $a \leqslant b$ and $b \leqslant a$ for distinct a and b. We capture scope as a preorder on a term t as follows:

Definition 3.1 (Scope). A *scope preorder* on a term t is a preorder $(\leqslant)$ on the references and declarations in t such that references are least in this preorder (i.e., nothing is ever smaller than a reference).

Definition 3.2. A reference $x^{\mathbf{R}}_{\mathbf{i}}$ is *in scope of* a declaration $x^{\mathbf{D}}_{\mathbf{j}}$ when $x^{\mathbf{R}}_{\mathbf{i}} \leqslant x^{\mathbf{D}}_{\mathbf{j}}$.

Definition 3.3. A declaration $x^{\mathbf{D}}_{\mathbf{i}}$ is *more specific than* another $x^{\mathbf{D}}_{\mathbf{j}}$ when $x^{\mathbf{D}}_{\mathbf{i}} \leqslant x^{\mathbf{D}}_{\mathbf{j}}$.

Note that these definitions rely on the *existence* of a preorder $(\leqslant)$, but don't say how to determine it for a given term. We will present *scoping rules* to do so in section 4. These definitions therefore provide very little on their own, but they can be built upon to define some common concepts:

Definition 3.4 (Bound). A reference is *bound* to the most specific declaration(s) that it has the same name as and is in scope of. More formally, we write:

$$x^{\mathbf{R}} \mapsto x^{\mathbf{D}} \triangleq x^{\mathbf{D}} \in \min\{x^{\mathbf{D}}_{\mathbf{i}} \mid x^{\mathbf{R}} \leqslant x^{\mathbf{D}}_{\mathbf{i}}\}$$

where $\min S$ finds the (zero or more) least elements of S:

$$\min S \triangleq \{a \in S \mid \nexists b \in S.\ b \leqslant a \text{ and } a \nleqslant b\}$$

Definition 3.5 (Unbound). A reference is *unbound* (or *free*) when it is not bound by any declaration.

[5] If we allowed non-local arrows, then in the previous example, inference would produce a single arrow from $\bullet_4$ to $\bullet_1$ instead of arrows "a" and "b". Then the orange/dashed arrow could *not* be inferred, since it relied on the existence of arrow "a", and the inference process would fail at its task.

Definition 3.6 (Ambiguously Bound). A variable reference is *ambiguously bound* when it is bound by more than one declaration.

Ambiguous binding may occur, for instance, if two declarations have the same name and are both parameters to the same function. In this case, a reference in the body of the function would be ambiguously bound to both of them. We also capture the idea of *shadowing*, where a more specific declaration hides a less specific declaration:

Definition 3.7 (Shadowing). A declaration *shadows* the most specific declarations that it has the same name as and is more specific than. Formally, $x_\mathrm{i}^\mathbf{D}$ shadows $x_\mathrm{j}^\mathbf{D}$ when:

$$x_\mathrm{i}^\mathbf{D} \mapsto x_\mathrm{j}^\mathbf{D} \triangleq x_\mathrm{j}^\mathbf{D} \in \min\{x_\mathrm{k}^\mathbf{D} \mid i \neq k \text{ and } x_\mathrm{i}^\mathbf{D} \leqslant x_\mathrm{k}^\mathbf{D}\}$$

(We use the same notation $\square \mapsto \square$ for both binding and shadowing because the definitions are analogous.)

3.3 Validating the Definitions

Since this description of scope is new, readers might wonder whether our definitions of concepts match their vernacular meaning. We provide evidence that they do in two forms.

First, we prove a simple lemma below showing that shadowing behaves as one would expect. Second, we show (section 3.4) that the notion of scope used in "Binding as Sets of Scopes" [Flatt 2016] obeys our scope-as-a-preorder definitions, for an appropriate choice of preorder ($\leqslant$). Additionally, in the supplement, we introduce a second, very intuitive definition of scope called scope-as-sets, and show that it is equivalent to scope-as-a-preorder up to a normalization.

Lemma 3.8 (Shadowing). *If one declaration shadows another, then a reference in scope of the shadowing declaration cannot be bound by the shadowed declaration.*

Proof. Suppose that $x_\mathrm{j}^\mathbf{D}$ shadows $x_\mathrm{i}^\mathbf{D}$ ($x_\mathrm{j}^\mathbf{D}$ is the shadowing declaration and $x_\mathrm{i}^\mathbf{D}$ is the shadowed declaration), and $x_\mathrm{k}^\mathbf{R}$ is in scope of $x_\mathrm{j}^\mathbf{D}$. What might $x_\mathrm{k}^\mathbf{R}$ be bound by? By definition, it will be bound by $\min\{x_\mathrm{l}^\mathbf{D} \mid x_\mathrm{k}^\mathbf{R} \leqslant x_\mathrm{l}^\mathbf{D}\}$. But since $x_\mathrm{k}^\mathbf{R} \leqslant x_\mathrm{j}^\mathbf{D} \leqslant x_\mathrm{i}^\mathbf{D}$, $x_\mathrm{i}^\mathbf{D}$ cannot be in this set, and $x_\mathrm{k}^\mathbf{R}$ cannot be bound by $x_\mathrm{i}^\mathbf{D}$. □

3.4 Relationship to "Binding as Sets of Scopes"

Scope-as-a-preorder aligns with the notion of scope expressed by Flatt [2016]. In his formulation, each subterm in the program is labeled with a *set of scopes*, called its *scope set*. A reference's binding (i.e., declaration) is then found "as one whose set of scopes is a subset of the reference's own scopes (in addition to having the same symbolic name)". If there is more than one such declaration, a reference is bound by the one with the largest (superset-wise) scope set. If there is no unique such element, then the reference is "ambiguous" [Flatt 2016, pp. 3].

This can be expressed in terms of scope-as-a-preorder. Take the preorder ($\leqslant$) to be (the least relation such that):

$$\begin{aligned} &x_\mathrm{i}^\mathbf{R} \leqslant x_\mathrm{i}^\mathbf{R} \\ &x^\mathbf{R} \leqslant y^\mathbf{D} \text{ iff } \textit{scope-set}\,(x^\mathbf{R}) \supseteq \textit{scope-set}\,(y^\mathbf{D}) \\ &x_\mathrm{i}^\mathbf{D} \leqslant y_\mathrm{j}^\mathbf{D} \text{ iff } \textit{scope-set}\left(x_\mathrm{i}^\mathbf{D}\right) \supseteq \textit{scope-set}\left(y_\mathrm{j}^\mathbf{D}\right) \end{aligned}$$

Then our definition of a reference's binding agrees with Flatt's, and our definition of an "ambiguously bound" reference agrees with his definition of an "ambiguous" reference.

4 A BINDING SPECIFICATION LANGUAGE

The previous section presented definitions for *representing* the scoping of a term. It did not, however, say how to *determine* the scoping of a term, i.e., what the specific preorder should be. In this section, we give a language for specifying *scoping rules* that, given a term, determine a preorder over its variables.

4.1 Scoping Rules: Simplified

The basic idea behind our binding language is that the binding structure of a term should be determined piecewise by its subterms. Thus every term constructor (e.g., λ or $+$) should specify a *scoping rule* that gives a preorder amongst itself and its children. A term's scope-as-a-preorder can then be found by taking the transitive closure of these local preorders across the whole term.

As an example, take the term $(\texttt{Lambda}\ x_1^{\mathbf{D}}\ (\texttt{Plus}\ x_2^{\mathbf{R}}\ 3))$. To find the bindings of this term, we must know the scoping rules for Lambda and Plus. A sensible rule for Plus is that a term $(\texttt{Plus}\ \bullet_1\ \bullet_2)$ has preorder $\bullet_1 \leqslant (\texttt{Plus}\ \bullet_1\ \bullet_2)$ and $\bullet_2 \leqslant (\texttt{Plus}\ \bullet_1\ \bullet_2)$, meaning that whatever a Plus term is in scope of, its children are too. For brevity, we will typically write $\bullet_1 \leqslant \texttt{Plus}$ and $\bullet_2 \leqslant \texttt{Plus}$ instead. Likewise, a sensible rule for Lambda is that a term $(\texttt{Lambda}\ \bullet_1\ \bullet_2)$ has preorder $(\bullet_2 \leqslant \bullet_1 \leqslant \texttt{Lambda})$, meaning that whatever a Lambda term is in scope of, its children are too, and that $\bullet_2$ (its body) is in scope of $\bullet_1$ (its declaration). Put together, and applied to the example term, these rules give that:

$$(\texttt{Lambda}\ x_1^{\mathbf{D}}\ (\texttt{Plus}\ x_2^{\mathbf{R}}\ 3))$$

has preorder:

$$x_2^{\mathbf{R}}, 3 \leqslant \texttt{Plus} \leqslant x_1^{\mathbf{D}} \leqslant \texttt{Lambda}$$

Thus $x_2^{\mathbf{R}} \mapsto x_1^{\mathbf{D}}$ by definition 3.4, as it should be.

4.2 A Problem

This isn't quite the whole picture, though. Consider the term

$$(\texttt{Lambda}\ x_1^{\mathbf{D}}\ (\texttt{Let*}\ (\texttt{Bind}\ x_2^{\mathbf{D}}\ x_3^{\mathbf{R}}\ \texttt{EndBinds})\ x_4^{\mathbf{R}}))$$

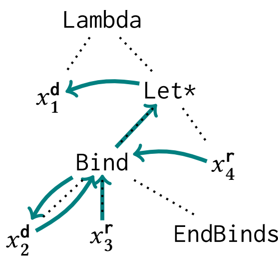
What will these scoping rules look like? Whatever they are, they should cause $x_2^{\mathbf{D}}$ to shadow $x_1^{\mathbf{D}}$, $x_3^{\mathbf{R}}$ to be bound by $x_1^{\mathbf{D}}$, and $x_4^{\mathbf{R}}$ to be bound by $x_2^{\mathbf{D}}$. Formally, we should have:

$$x_2^{\mathbf{D}} \mapsto x_1^{\mathbf{D}} \text{ and } x_3^{\mathbf{R}} \mapsto x_1^{\mathbf{D}} \text{ and } x_4^{\mathbf{R}} \mapsto x_2^{\mathbf{D}}$$

which implies that, at a minimum:

$$x_2^{\mathbf{D}} \leqslant x_1^{\mathbf{D}} \text{ and } x_3^{\mathbf{R}} \leqslant x_1^{\mathbf{D}} \text{ and } x_4^{\mathbf{R}} \leqslant x_2^{\mathbf{D}}$$

This places a set of requirements on the scoping rules for Lambda, Let*, and Bind. For instance, $x_3^{\mathbf{R}} \leqslant x_1^{\mathbf{D}}$ can only be achieved if $x_3^{\mathbf{R}} \leqslant \texttt{Bind} \leqslant \texttt{Let*} \leqslant x_1^{\mathbf{D}}$. Continuing this way gives the requirements (shown both pictorially and textually):



$$\begin{array}{lcl} \texttt{Let*} & \leqslant & x_1^{\mathbf{D}} \\ \texttt{Bind} & \leqslant & \texttt{Let*} \\ x_4^{\mathbf{R}} & \leqslant & \texttt{Bind} \\ x_2^{\mathbf{D}} & \leqslant & \texttt{Bind} \\ x_3^{\mathbf{R}} & \leqslant & \texttt{Bind} \\ \texttt{Bind} & \leqslant & x_2^{\mathbf{D}} \end{array}$$

However, this puts $x_3^{\mathbf{R}}$ in scope of $x_2^{\mathbf{D}}$, and as a result, $x_3^{\mathbf{R}}$ will be bound by $x_2^{\mathbf{D}}$! The problem is that Bind is trying to provide $x_2^{\mathbf{D}}$, to make it available in the body of Let*, but in doing so it incidentally makes it available in the Bind's definition (to $x_3^{\mathbf{R}}$). This is not how scoping dependencies should flow, and in the next two subsections we present the full, un-simplified version of our scoping rules that avoid this problem.

4.3 The Solution

The solution is to separate the bindings a term *imports* (i.e., requires) from the bindings it *exports* (i.e., provides). In the running example, for instance, the Bind *imports* $x_1^{\mathbf{D}}$, and *exports* $x_1^{\mathbf{D}}$ and $x_2^{\mathbf{D}}$ (with $x_2^{\mathbf{D}}$ shadowing $x_1^{\mathbf{D}}$). We will call imports and exports *ports*.

The scoping rules can now be re-interpreted with this in mind. Given a term t, they will determine a preorder not over the subterms of t (like we have presented it so far), but instead over the *ports* of the subterms of t. With this in mind, we offer four kinds of bindings:[6]

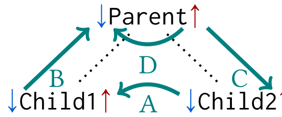
A. **`bind i in j`:** A term may make its i'th child's bindings available in its j'th child. If so, any declarations *exported* by child i will be *imported* by child j.

B. **`import i`:** A term's i'th child may import its parent's declarations. If so, it *imports* the declarations *imported* by its parent. (This is almost universal, as it represents lexical scope: declarations in scope at a node in an AST should also be in scope at its children. However, we do allow a term to hide all bindings from its child, if it so desires.)

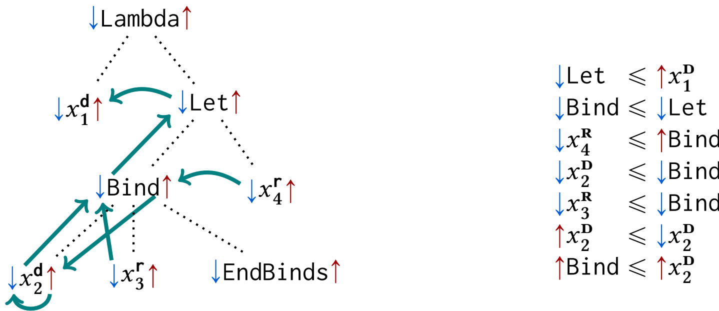
C. **`export i`:** A term's i'th child may export its declarations to its parent. If so, the term *exports* child j's *exports*.

D. **`re-export`:** A term may take the declarations it *imported*, and *export* them. (This is not terribly useful in practice, but we offer it for completion.)

These four kinds of paths may be represented graphically, showing imports as ↓ and exports as ↑:



With these new bindings in mind, the requirements for the example from the previous subsection become:



$$
\begin{aligned}
\downarrow\text{Let} &\leqslant \uparrow x_1^{\mathbf{D}}\\
\downarrow\text{Bind} &\leqslant \downarrow\text{Let}\\
\downarrow x_4^{\mathbf{R}} &\leqslant \uparrow\text{Bind}\\
\downarrow x_2^{\mathbf{D}} &\leqslant \downarrow\text{Bind}\\
\downarrow x_3^{\mathbf{R}} &\leqslant \downarrow\text{Bind}\\
\uparrow x_2^{\mathbf{D}} &\leqslant \downarrow x_2^{\mathbf{D}}\\
\uparrow\text{Bind} &\leqslant \uparrow x_2^{\mathbf{D}}
\end{aligned}
$$

[6] There is a close analogy between ports and attributes in attribute grammars [Knuth 1968]: namely, imports are analogous to inherited attributes and exports are analogous to synthesized attributes. The paths between imports and exports that are allowed by our binding language (e.g., child export to parent export, but not child export to parent import) are precisely the relationships between inherited and synthesized attributes that are allowed in attribute grammars. Most algorithms for evaluating attribute grammars disallow cycles, however, while our preorders allow them.

Under this new preorder, $x_3^{\mathbf{R}} \mapsto x_1^{\mathbf{D}}$ and $x_4^{\mathbf{R}} \mapsto x_2^{\mathbf{D}}$ as desired.

4.4 Scoping Rules: Unsimplified

We have given an intuition behind our scoping rules; now we present them formally.

Each port will have one of two *signs* (import or export):

$$\begin{array}{rcll} d & ::= & \downarrow & \text{(import)} \\ & | & \uparrow & \text{(export)} \end{array}$$

A *port*, then, pairs a term t with a sign:

$$a, b, c \quad ::= \quad \downarrow t \mid \uparrow t \quad \text{(port)}$$

A set of *scope rules* Σ gives a relation for each term constructor P that describes the scoping relationships between a term constructed with P and its subterms:

Definition 4.1. A set of *scope rules* Σ is a partial map from term constructors P of arity n to binary relations over $\{1, \dots, n, \text{R}^{\uparrow}, \text{R}^{\downarrow}\}$, such that:

- The relation is transitive.
- $\text{R}^{\uparrow}$ is a least element ($\nexists a.\ (a, \text{R}^{\uparrow}) \in \Sigma[P]$)
- $\text{R}^{\downarrow}$ is a greatest element ($\nexists a.\ (\text{R}^{\downarrow}, a) \in \Sigma[P]$)

Here i represents the ith child term, $\text{R}^{\uparrow}$ represents the parent term's exports, and $\text{R}^{\downarrow}$ represents the parent term's imports. We will call pairs in the relation (e.g., $(1, \text{R}^{\downarrow})$) *facts*, and will equate them with their description in our binding language (so that $(1, \text{R}^{\downarrow}) =$ `import 1`). The sign on the port on i can be determined knowing that the fact it is part of must be one of the four kinds of bindings described in section 4.3. We will write $s \sqsubseteq t$ to mean that s is a subterm of t, and write $a \sqsubseteq t$ to mean $\exists s.\ (a = \downarrow s \text{ or } a = \uparrow s)$ and $s \sqsubseteq t$.

As an example of scope rules, the rules for Lambda are:

$$\Sigma[\text{Lambda}] = \{(1, \text{R}^{\downarrow}), (2, \text{R}^{\downarrow}), (2, 1)\} = \{\texttt{import 1}, \texttt{import 2}, \texttt{bind 1 in 2}\}$$

These scope rules determine the scoping for individual (sub)terms. The scoping of a *full* term is found by applying the scoping rules locally at each subterm, then taking the reflexive transitive closure of this global relation:

Definition 4.2. The *scoping* of a full term t under scoping rules Σ is the set of judgements of the form $\Sigma, t \vdash a \leqslant b$ defined by the "Declarative Rules" and "Shared Rules" of fig. 1.

The judgments in the figure have the form $\Sigma, t \vdash a \leqslant b$, which means that "$a \leqslant b$ in term t using scoping rules Σ". A judgment is *well formed* when $a, b \sqsubseteq t$. (Later, we will also use judgments of the form $\Sigma, C \vdash a \leqslant b$; these are governed by identical rules, allowing each term t to instead be a context C.)

Rules SD-Import, SD-Export, SD-Bind, and S-ReExport capture the direct meaning of the scoping rules. S-Refl, S-Refl2, and SD-Trans give the transitive reflexive closure. SD-Decl allows declarations to extend the current scope. S-Lift says that facts learned about a subterm remain true in the whole term.

These rules are not, however, syntax-directed. We give a syntax-directed version of the rules in the figure, under "Algorithmic Rules" and "Shared Rules". These two rule sets are equivalent:

THEOREM 4.3 (ALGORITHMIC SCOPE CHECKING). *The declarative and algorithmic scope checking rules (fig. 1) [with shared rules common to both] are equivalent.*

PROOF. Given in the supplement. ◻

$\boxed{\Sigma, t \vdash a \leqslant b}$

Declarative Rules

$$\text{SD-Trans}\ \frac{\Sigma, t \vdash a \leqslant b \quad \Sigma, t \vdash b \leqslant c}{\Sigma, t \vdash a \leqslant c}$$

$$\text{SD-Import}\ \frac{\texttt{import}\ i \in \Sigma[P]}{\Sigma, (P\ t_1 \dots t_n) \vdash \downarrow t_i \leqslant \downarrow (P\ t_1 \dots t_n)}$$

$$\text{SD-Export}\ \frac{\texttt{export}\ i \in \Sigma[P]}{\Sigma, (P\ t_1 \dots t_n) \vdash \uparrow (P\ t_1 \dots t_n) \leqslant \uparrow t_i}$$

$$\text{SD-Bind}\ \frac{\texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P]}{\Sigma, (P\ t_1 \dots t_n) \vdash \downarrow t_i \leqslant \uparrow t_j}$$

Shared Rules (Declarative & Algorithmic)

$$\text{S-Refl1}\ \frac{}{\Sigma, t \vdash \downarrow t \leqslant \downarrow t} \qquad \text{S-Refl2}\ \frac{}{\Sigma, t \vdash \uparrow t \leqslant \uparrow t}$$

$$\text{S-Lift}\ \frac{\Sigma, t_i \vdash a \leqslant b}{\Sigma, (P\ t_1 \dots t_n) \vdash a \leqslant b}$$

$$\text{S-Decl}\ \frac{}{\Sigma, x^{\mathbf{D}} \vdash \uparrow x^{\mathbf{D}} \leqslant \downarrow x^{\mathbf{D}}}$$

$$\text{S-ReExport}\ \frac{\texttt{re-export} \in \Sigma[P]}{\Sigma, (P\ t_1 \dots t_n) \vdash \uparrow (P\ t_1 \dots t_n) \leqslant \downarrow (P\ t_1 \dots t_n)}$$

Algorithmic Rules

$$\text{SA-Import}\ \frac{\Sigma, t_i \vdash a \leqslant \downarrow t_i \quad \texttt{import}\ i \in \Sigma[P]}{\Sigma, (P\ t_1 \dots t_n) \vdash a \leqslant \downarrow (P\ t_1 \dots t_n)}$$

$$\text{SA-Export}\ \frac{\texttt{export}\ i \in \Sigma[P] \quad \Sigma, t_i \vdash \uparrow t_i \leqslant a}{\Sigma, (P\ t_1 \dots t_n) \vdash \uparrow (P\ t_1 \dots t_n) \leqslant a}$$

$$\text{SA-Bind}\ \frac{\Sigma, t_i \vdash a \leqslant \downarrow t_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad \Sigma, t_j \vdash \uparrow t_j \leqslant b}{\Sigma, (P\ t_1 \dots t_n) \vdash a \leqslant b}$$

Fig. 1. Scope Checking

These scope checking rules say how to find a preorder over all of the *ports* in a term. However, section 3 is based on preorders over the *variables* in a term.[7] This is obtained as the restriction of the entire preorder to variables, as captured by the following rule:

$$\boxed{\Sigma, t \vdash x \leqslant y} \qquad \text{S-Var}\ \frac{\Sigma, t \vdash \downarrow x_{\mathrm{i}} \leqslant \downarrow y_{\mathrm{j}}}{\Sigma, t \vdash x_{\mathrm{i}} \leqslant y_{\mathrm{j}}}$$

[7] In fact, scope-as-a-preorder could be used with a *different* binding language, so long as it can be used to extract a preorder.

The definitions for binding and shadowing (definitions 3.4 and 3.7) can then be expressed as inference rules:

$$\boxed{\Sigma, t \vdash x \mapsto y} \qquad \text{S-Bound}\ \frac{x^{\mathbf{D}} \in \min\{x_{\mathrm{i}}^{\mathbf{D}} \mid \Sigma, t \vdash x^{\mathbf{R}} \leqslant x_{\mathrm{i}}^{\mathbf{D}}\}}{\Sigma, t \vdash x^{\mathbf{R}} \mapsto x^{\mathbf{D}}}$$

$$\text{S-Shadow}\ \frac{x_{\mathrm{j}}^{\mathbf{D}} \in \min\{x_{\mathrm{k}}^{\mathbf{D}} \mid i \neq k \text{ and } \Sigma, t \vdash x_{\mathrm{i}}^{\mathbf{D}} \leqslant x_{\mathrm{k}}^{\mathbf{D}}\}}{\Sigma, t \vdash x_{\mathrm{i}}^{\mathbf{D}} \mapsto x_{\mathrm{j}}^{\mathbf{D}}}$$

These definitions form a scope preorder:

Lemma 4.4. *For any set of scoping rules* Σ *and term* t*, the relation* $\{(x_{\mathrm{i}}, x_{\mathrm{j}}) \mid \Sigma, t \vdash x_{\mathrm{i}} \leqslant x_{\mathrm{j}}\}$ *is a scope preorder satisfying the requirements of definition 3.1.*

Proof sketch. The relation is a preorder by the derivation rules S-Refl1, S-Refl2, and SD-Trans. We must also show that references are least. Suppose instead that $x_{\mathrm{i}} \leqslant x_{\mathrm{j}}^{\mathbf{R}}$ for some $i \neq j$. Then $\downarrow x_{\mathrm{i}} \leqslant \downarrow x_{\mathrm{j}}^{\mathbf{R}}$, which is syntactically impossible to achieve by the declarative judgements. □

4.5 Well-Boundedness

Definition 3.4 (on being bound) can be used to define α-equivalence. Two terms are α-equivalent if (i) each term is "well-bound"; (ii) they have the same "shape" (i.e., they are identical ignoring their variable names); and (iii) for every binding $x^{\mathbf{R}} \mapsto x^{\mathbf{D}}$ in one term, an analogous binding exists in the same location in the other term. To formalize what "same location" means, we will use a *join* operator ($s \bowtie t$) that checks that s and t have the same shape and finds a bijection between their variable occurrences as a witness to this fact:

$$\begin{array}{llll}
x_{\mathrm{i}}^{\mathbf{D}} & \bowtie\ y_{\mathrm{j}}^{\mathbf{D}} & = \{x_{\mathrm{i}}^{\mathbf{D}} \leftrightarrow y_{\mathrm{j}}^{\mathbf{D}}\} & \\
x_{\mathrm{i}}^{\mathbf{R}} & \bowtie\ y_{\mathrm{j}}^{\mathbf{R}} & = \{x_{\mathrm{i}}^{\mathbf{R}} \leftrightarrow y_{\mathrm{j}}^{\mathbf{R}}\} & \\
\mathit{const} & \bowtie\ \mathit{const} & = \varnothing & \\
(P\ s_1 \ldots s_n) & \bowtie\ (P\ t_1 \ldots t_n) & = \bigcup_{i \in 1..n} s_i \bowtie t_i & \\
s & \bowtie\ t & = \text{UNDEFINED} & \text{otherwise}
\end{array}$$

Likewise, to formalize "well-bound", we will use the rules to determine when two declarations *conflict*; for instance if they have the same name and are both parameters to the same function. We will consider terms with conflicting declarations to be ill-bound.

Definition 4.5 (Conflicting Declarations). Two variable declarations $x_{\mathrm{i}}^{\mathbf{D}}$ and $x_{\mathrm{j}}^{\mathbf{D}}$ *conflict* in a term t when:

$$\text{S-Conflict}\ \frac{\Sigma, t \vdash a \leqslant x_{\mathrm{i}}^{\mathbf{D}} \quad \Sigma, t \vdash a \leqslant x_{\mathrm{j}}^{\mathbf{D}} \qquad \min_{\Sigma, t}\{x_{\mathrm{i}}^{\mathbf{D}}, x_{\mathrm{j}}^{\mathbf{D}}\} = \{x_{\mathrm{i}}^{\mathbf{D}}, x_{\mathrm{j}}^{\mathbf{D}}\}}{\Sigma, t \vdash x_{\mathrm{i}}^{\mathbf{D}}\ \mathit{conflicts}\ x_{\mathrm{j}}^{\mathbf{D}}}$$

(If a variable reference is *ambiguously bound* (definition 3.6), then its bindings declarations must be in conflict.)

A term t is *well-bound* with respect to scoping rules Σ when every reference is bound by exactly one declaration, and there are no conflicting declarations:

$$\text{S-WB}\ \frac{\forall x^{\mathbf{R}} \in t.\ \exists! x^{\mathbf{D}} \in t.\ \Sigma, t \vdash x^{\mathbf{R}} \mapsto x^{\mathbf{D}} \qquad \nexists x_{\mathrm{i}}^{\mathbf{D}}, x_{\mathrm{j}}^{\mathbf{D}} \in t.\ \Sigma, t \vdash x_{\mathrm{i}}^{\mathbf{D}}\ \mathit{conflicts}\ x_{\mathrm{j}}^{\mathbf{D}}}{\Sigma \vdash \text{WB}\ t}$$

The definition of α-equivalence with respect to the scoping rules Σ is then:

Definition 4.6 (α-equivalence).

$$\text{S-}\alpha\text{-Eqv}\ \frac{\Sigma \vdash \textsc{wb}\ s \quad \Sigma \vdash \textsc{wb}\ t \quad s \bowtie t = \psi \qquad \forall x^{\mathbf{R}}, x^{\mathbf{D}}.\ \Sigma, s \vdash x^{\mathbf{R}} \mapsto x^{\mathbf{D}} \text{ iff } \Sigma, t \vdash \psi(x^{\mathbf{R}}) \mapsto \psi(x^{\mathbf{D}})}{\Sigma \vdash s =_\alpha t}$$

(We will also talk about α-equivalence and well-boundedness of patterns. The definitions are identical.)

In section 6.5 of the supplement, we show a catalog of scoping rules that can be expressed in our binding language.

5 INFERRING SCOPE

In this section we show how to infer scope by lifting scoping rules from a core language to the surface language. The input to this inference process is twofold: first, the core language must have associated scoping rules, and second, the syntactic sugar must be given as a set of pattern-based rewrite rules. The output of scope inference is a set of scoping rules for the surface language.

The process is loosely analogous to type inference: type inference finds the most general type annotations such that a program type-checks; scope inference will find the smallest set of surface scoping rules under which desugaring preserves α-equivalence. More precisely, given a core language with scoping rules Σ_{core}, and a desugaring f, our algorithm finds scoping rules Σ_{surf} that preserves α-equivalence (theorem 7.1), so that:

$$\Sigma_{surf} \vdash s =_\alpha t \quad \text{implies} \quad \Sigma_{core} \vdash f(s) =_\alpha f(t)$$

Furthermore, Σ_{surf} will be least, so that if Σ'_{surf} also has this property, then $\forall P.\ \Sigma_{surf}[P] \subseteq \Sigma'_{surf}[P]$.

The general algorithm for scope inference is given in fig. 2. The next three subsections explain our assumptions about desugaring, and then the algorithm.

5.1 Assumptions about Desugaring

We will assume that desugaring is given (externally to the language) by a set of rewrite rules of the form $C \Rightarrow C'$, where C and C' are contexts (terms with *holes* $\bullet_i$ in them, *not* evaluation contexts).[8] [9]

$$\begin{array}{rcll} C & ::= & \ldots & \text{(as in term } t\text{)} \\ & \mid & \bullet_i & \text{(hole)} \end{array}$$

Furthermore, we assume that for every rule $C \Rightarrow C'$:

(1) Every hole in C' also appears in C.
(2) No hole in C or C' appears more than once.
(3) C contains no references or declarations. (Rather, these should be contained in its holes during expansion.)
(4) References and declarations in C' are given fresh names during expansion to ensure hygiene. (Our implementation also supports *global* references—e.g., calling `print`—but we leave this out of the paper for simplicity.)

When desugaring, there may be more than one rewrite rule that applies to a given term. We make no assumption about which will be chosen; even a non-deterministic desugaring is allowed. A more typical choice is to apply rules in outside-in order, as is done by Scheme-style `syntax-rules` macros [Kelsey et al. 1998].

[8] Formally, this is a term rewriting system (TRS) [Klop 1992]. We are calling the TRS's variables *holes* to avoid confusing them with references and declarations, which are constants from the perspective of the TRS.
[9] For now, we will admit *unhygienic* sugars; hygiene will be addressed in section 7.

$inferScope\,(\Sigma,\ \{C_i \Rightarrow C_i'\}_{i\in 1..n})$ ≜ Let $\Sigma_{surf} = solve\left(\Sigma,\ \bigcup_{i\in 1..n} genConstrs\,(C_i \Rightarrow C_i')\right)$
$checkScope\,(\Sigma_{surf},\ \{C_i \Rightarrow C_i'\}_{i\in 1..n})$
Return Σ_{surf}

$genConstrs\,(C \Rightarrow C')$ ≜ $\{genConstr\,(a \leqslant b,\ C \Rightarrow C')\}_{a,b\in H}$
where $H = holes\,(C) \cup holes\,(C') \cup \{\text{R}\}$

$genConstr\,(a \leqslant b,\ C \Rightarrow C')$ ≜ $(genConj\,(a \leqslant b,\ C), genConj\,(a \leqslant b,\ C'))$
where (p, q) means a constraint "p iff q"

$genConj\,(a \leqslant b,\ C)$ ≜ Smallest Σ_{surf} such that $\Sigma, C \vdash a \leqslant b$.
(To compute this, take the premises of the (unique) derivation of $\Sigma, C \vdash a \leqslant b$ using the Algorithmic Scope Checking rules (fig. 1).)

$solve\,(\Sigma_{core},\ constraints)$ ≜ Initialize $\Sigma_{surf} = \Sigma_{core}$
Until fixpoint:

- If a fact f in a constraint is in Σ_{surf}:
  Delete f from the constraint
- If one side of a constraint is empty:
  Delete the constraint
  Add the other side to Σ_{surf}
  (maintaining transitive closure)
- If any fact in Σ_{surf} is in the complement of Σ_{core}:
  ERROR: Reject this sugar

Return Σ_{surf}

$checkScope\,(\Sigma,\ \{C_i \Rightarrow C_i'\}_{i\in 1..n})$ ≜ For each rule $C \Rightarrow C'$:
Assert that if $\Sigma, C \vdash a \leqslant b$ and $a \in C'$ then $b \in C'$
(otherwise ERROR)
Assert that each reference $x^{\mathbf{R}} \in C'$ is bound by a unique declaration $x^{\mathbf{D}} \in C'$

Fig. 2. Scope Inference Algorithm

In general, a rewrite will look like:

$$E[C[t_1, ..., t_n]] \Rightarrow E[C'[t_1, ..., t_n]]$$

where E and C are contexts, and $C[t_1, ..., t_n]$ denotes replacing the n holes of context C with terms $t_1, ..., t_n$. (In section 2, C was called the LHS, and C' the RHS.) The outer context E is important because when a piece of sugar expands, while its *expansion* doesn't typically depend on its surrounding context, its binding structure might. For example, E might be (`Lambda` $x^{\mathbf{D}}$ $\bullet$), and $x^{\mathbf{R}}$ inside the hole may be unbound without it.

5.2 Constraint Generation

The first step to scope inference is generating a set of constraints for each desugaring rule that, if satisfied, ensure that it will preserve binding structure. Specifically, fix a rewrite rule $C \Rightarrow C'$. It is important that this rewrite does not change the binding of any variable *outside* of C. To achieve this, it will suffice that the preorder on the *boundary* of C is the same as the preorder among the boundary of C'. The boundary, here, is the set of holes in C, together with the root (i.e., the whole term). For example, in $E[C[t_1, ..., t_n]]$, $\bullet_i$ bounds t_i, and C (the root) bounds E. In general, we will call this property *scope-equivalence*:

Definition 5.1 (Scope-equivalence of contexts). $\Sigma \vdash C \cong C'$ means that $\forall a, b \in \{\bullet_1, ..., \bullet_n, \textsc{r}\}$.

$$\Sigma, C \vdash a \leqslant b \text{ iff } \Sigma, C' \vdash a \leqslant b$$

where $\textsc{r}$ ("root") stands in for C or C', as appropriate, and omitted port signs are determined by what our binding language allows:

$$\begin{array}{lcl} \Sigma, C \vdash \bullet_i \leqslant \bullet_j & \triangleq & \Sigma, C \vdash {\downarrow}\bullet_i \leqslant {\uparrow}\bullet_j \\ \Sigma, C \vdash \bullet_i \leqslant \textsc{r} & \triangleq & \Sigma, C \vdash {\downarrow}\bullet_i \leqslant {\downarrow}C \\ \Sigma, C \vdash \textsc{r} \leqslant \bullet_j & \triangleq & \Sigma, C \vdash {\uparrow}C \leqslant {\uparrow}\bullet_j \\ \Sigma, C \vdash \textsc{r} \leqslant \textsc{r} & \triangleq & \Sigma, C \vdash {\uparrow}C \leqslant {\downarrow}C \end{array}$$

When two contexts are scope-equivalent, rewriting one to the other within a term does not change the scope of the rest of the term:

Definition 5.2 (Scope-preservation). A rewrite

$$E[C[t_1, ..., t_n]] \Rightarrow E[C'[t_1, ..., t_n]]$$

preserves scope relative to a set of scoping rules Σ if $\forall\, a, b \sqsubseteq E, t_1, ..., t_n$ (i.e., each of a and b lies in one of $E, t_1, ..., t_n$):

$$\Sigma, E[C[t_1, ..., t_n]] \vdash a \leqslant b \text{ iff } \Sigma, E[C'[t_1, ..., t_n]] \vdash a \leqslant b$$

Lemma 5.3 (Scope-equivalent contexts preserve scope). *If $\Sigma \vdash C \cong C'$, then any rewrite $E[C[t_1, ..., t_n]] \Rightarrow E[C'[t_1, ..., t_n]]$ preserves scope.*

Proof. We will prove the forward implication of the *iff* in scope-preservation; the reverse is symmetric. View the $(\leqslant)$ preorder as a directed graph. Our given is that there is a path from a to b in $E[C[t_1, ..., t_n]]$, where neither a nor b lie in C. Some subpaths of this path may traverse C; these subpaths are bounded by ${\downarrow}t_1, {\uparrow}t_1, ..., {\downarrow}t_n, {\uparrow}t_n, {\downarrow}C, {\uparrow}C$. The fact that $\Sigma \vdash C \cong C'$ means that these subpaths can be converted to subpaths in C', bounded instead by ${\downarrow}t_1, {\uparrow}t_1, ..., {\downarrow}t_n, {\uparrow}t_n, {\downarrow}C', {\uparrow}C'$. Replace these subpaths. Now the whole path goes from a to b in $E[C'[t_1, ..., t_n]]$. □

We can use scope-equivalence to turn a rewrite rule $C \Rightarrow C'$ into a set of constraints that hold iff the rewrite rule preserves scope. There will be one constraint for every pair (a, b) from the boundary. Each constraint will have the form:

$$F_1 \wedge F_2 \wedge ... F_n \;\; \texttt{iff} \;\; F'_1 \wedge F'_2 ... \wedge F'_m$$

where each F_i is a fact (e.g. `bind 2 in 1` $\in \Sigma[Let]$). This constraint is found by stating that the premises of the derivation of $\Sigma, C \vdash a \leqslant b$ hold "iff" the premises of the derivation $\Sigma, C' \vdash a \leqslant b$ hold. These derivations are guaranteed to be unique, and can found efficiently, because the algorithmic scope-checking rules (fig. 1) are syntax-directed.

As an example of this constraint generation, take the desugaring rule for Let:

$$(\texttt{Let}\ \bullet_1\ \bullet_2\ \bullet_3) \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ \bullet_1\ \bullet_3)\ \bullet_2)$$

One of the necessary constraints says that:

$$\begin{array}{rl} & \Sigma, (\texttt{Let}\ \bullet_1\ \ \bullet_2\ \bullet_3) \vdash \bullet_1 \leqslant \bullet_2 \\ \text{iff} & \Sigma, (\texttt{Apply}\ (\texttt{Lambda}\ \bullet_1\ \ \bullet_3)\ \bullet_2) \vdash \bullet_1 \leqslant \bullet_2 \end{array}$$

Each side of this "iff" has a unique derivation using the algorithmic scope-checking rules (fig. 1). Replacing each side with the premises of its derivation yields the constraint:

$$\texttt{bind 2 in 1} \in \Sigma[\texttt{Let}] \text{ iff } \texttt{bind 2 in 1} \in \Sigma[\texttt{App}] \wedge \texttt{import 1} \in \Sigma[\texttt{Lam}]$$

Since the boundary has size four ($\bullet_1$, $\bullet_2$, $\bullet_3$, and R), continuing this way leads to a total of $4^2 = 16$ constraints:

$$\begin{array}{rlrr}
\texttt{bind 1 in 1} \in \Sigma[\texttt{Let}] & \text{iff} & & \texttt{bind 1 in 1} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 2 in 1} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{bind 2 in 1} \in \Sigma[\texttt{App}] \wedge & \texttt{import 1} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 3 in 1} \in \Sigma[\texttt{Let}] & \text{iff} & & \texttt{bind 2 in 1} \in \Sigma[\texttt{Lam}] \\
\texttt{import 1} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{import 1} \in \Sigma[\texttt{App}] \wedge & \texttt{import 1} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 1 in 2} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{bind 1 in 2} \in \Sigma[\texttt{App}] \wedge & \texttt{export 1} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 2 in 2} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{bind 2 in 2} \in \Sigma[\texttt{App}] & \\
\texttt{bind 3 in 2} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{bind 1 in 2} \in \Sigma[\texttt{App}] \wedge & \texttt{export 2} \in \Sigma[\texttt{Lam}] \\
\texttt{import 2} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{import 2} \in \Sigma[\texttt{App}] & \\
\texttt{bind 1 in 3} \in \Sigma[\texttt{Let}] & \text{iff} & & \texttt{bind 1 in 2} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 2 in 3} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{bind 2 in 1} \in \Sigma[\texttt{App}] \wedge & \texttt{import 2} \in \Sigma[\texttt{Lam}] \\
\texttt{bind 3 in 3} \in \Sigma[\texttt{Let}] & \text{iff} & & \texttt{bind 2 in 2} \in \Sigma[\texttt{Lam}] \\
\texttt{import 3} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{import 1} \in \Sigma[\texttt{App}] \wedge & \texttt{import 2} \in \Sigma[\texttt{Lam}] \\
\texttt{export 1} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{export 1} \in \Sigma[\texttt{App}] \wedge & \texttt{export 1} \in \Sigma[\texttt{Lam}] \\
\texttt{export 2} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{export 2} \in \Sigma[\texttt{App}] & \\
\texttt{export 3} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{export 1} \in \Sigma[\texttt{App}] \wedge & \texttt{export 2} \in \Sigma[\texttt{Lam}] \\
\texttt{re-export} \in \Sigma[\texttt{Let}] & \text{iff} & \texttt{re-export} \in \Sigma[\texttt{App}] &
\end{array}$$

We have just described how to generate constraints—covering the *gen* functions in fig. 2—and the previous lemma shows that the constraints generated this way capture our aim in scope inference. We now turn to solving these constraints.

5.3 Constraint Solving

These constraints can be solved by searching for their least fixpoint, starting with the initial knowledge of the scoping rules for the core language. Finding the *least* fixpoint is sensible, because by default, declarations should not be in scope. Since all of the constraints have the form of an "iff" between conjunctions, the least fixpoint exists and can be found by monotonically growing the set of known facts.

Solving for the least fixpoint gives a set of scoping rules for the surface and core languages such that the desugaring rules preserve this scope. Since the least fixpoint was seeded with the known scoping rules for the core language, its output will contain at least those facts. However, they may have inferred *additional*, incorrect facts about the scope of the core language. For instance, consider the following "Lambda flip flop" rule (where `Flip` and `Flop` are constants, i.e., nodes of arity 0):

$$(\texttt{LambdaFF Flip}\ \bullet_1\ \ \bullet_2) \Rightarrow (\texttt{Lambda}\ \bullet_1\ \ \bullet_2)$$

$$(\texttt{LambdaFF Flop}\ \bullet_1\ \ \bullet_2) \Rightarrow (\texttt{Lambda}\ \bullet_2\ \ \bullet_1)$$

In traditional hygienic macro expansion systems this desugaring is considered to be OK: the scope of a term is *defined* by the scope of its desugaring, which may vary on things such as the choice between Flip and Flop constants. However, we will take the opposite view: this desugaring should

be rejected because the scope it produces for LambdaFF cannot be captured by (reasonable) static scoping rules.

Let us work through scope inference for this example. From the first rule, we can learn (from the Lambda on the RHS) that `bind 2 in 3` $\in \Sigma$[LambdaFF], and from the second rule, we can learn that `bind 3 in 2` $\in \Sigma$[LambdaFF]. Applying either of these facts to the *other* rule gives that `bind 2 in 1` $\in \Sigma$[Lambda]: the *body* of the Lambda is in scope at its *parameter*! This contradicts the known signature for Lambda (we know that `bind 2 in 1` $\notin \Sigma$[Lambda]), so these rules would be rejected. In general, scope inference fails when the least fixpoint contains facts about the scope of a core language construct that are not part of that construct's signature.

5.4 Ensuring Hygiene

We have described how to infer scope by generating and then solving constraints. There are two checks we should perform, however, to ensure that desugaring cannot produce unbound identifiers. These checks are performed by *checkScope* in fig. 2:

- Any references introduced on the RHS of a sugar must be bound. For instance, a sugar could not simply expand to $x^{\mathbf{R}}$, because that would be unbound.
- A sugar cannot delete a hole that might contain a bound declaration. For instance, it could not rewrite (`lambda` $\bullet_1$ $\bullet_2$) to $\bullet_2$, because $\bullet_2$ might contain a reference bound by a declaration in $\bullet_1$. In general, if a sugar deletes any hole, then it must also delete all smaller holes (those that are less in the preorder).

These two checks ensure that sugars cannot cause unbound identifier exceptions. Besides obviously being a problem, we would like to prevent this because it violates our notion of *hygiene*. However, these problematic sugars would not be considered unhygienic in the traditional sense.

Traditionally, research on hygiene has focused on preventing sugars from accidentally capturing user-defined references and vice versa. For instance, if a user binds $x^{\mathbf{D}}_{\mathrm{i}}$ and then uses $x^{\mathbf{R}}$ inside a sugar, and the sugar locally binds $x^{\mathbf{D}}_{\mathrm{j}}$, then $x^{\mathbf{R}}$ should not be bound by $x^{\mathbf{D}}_{\mathrm{j}}$. These hygiene violations are called "introduced-binder" and "introduced-reference" violations, respectively. There are also more subtle violations in which desugaring makes observations about declaration equality [Adams 2015].

However, there is a simpler goal we can aim for that gets at the heart of the problem, and subsumes all of these specific properties. The goal is that if two programs are α-equivalent, then they will still be α-equivalent after a desugaring f:

$$\Sigma_{surf} \vdash s =_\alpha t \text{ implies } \Sigma_{core} \vdash f(s) =_\alpha f(t)$$

(Recall from definition 4.6 that α-equivalence is parameterized by Σ. Therefore, in the above antecedent and consequent, α-equivalence is respectively defined by Σ_{surf} and Σ_{core}.)

This prevents accidental variable capture because α-renaming the captured variable would cause it to not be captured, changing the α-equivalence-class of the program. It also prevents the introduction of unbound identifiers, because a program with an unbound identifier is not α-equivalent to any other program (it is outside the domain of α-equivalence).

Most hygiene papers don't mention this criterion for a simple reason: $=_\alpha$ is not defined on their surface language, so they cannot even state the requirement. Recent exceptions to this rule [Herman and Wand 2008; Stansifer and Wand 2014] get around it by requiring sugar-writers to supply scoping rules for the surface language. These scoping rules then define α-equivalence for the surface language. In contrast, we *infer* scoping rules for the surface language, and can then ask whether these inferred rules preserve α-equivalence. In section 7 we will show that they do, so long as inference was successful and *scopeCheck* passed.

This covers the *solve* algorithm in fig. 2, and completes our description of scope inference: (i) find constraints for every desugaring rule; (ii) find their least fixpoint, starting with the known scoping rules for the core language; and (iii) check that none of the sugars can produce unbound identifiers.

5.5 Correctness and Runtime

The *inferScope* algorithm correctly solves the constraints:

Theorem 5.4 (Rewrites preserve scope).
Let $\Sigma_{surf} = inferScope\left(\Sigma_{core}, \{C_i \Rightarrow C_i'\}_{i\in 1..n}\right)$. *Then any rewrite of the form* $E[C_i[t_1, ..., t_n]] \Rightarrow E[C_i'[t_1, ..., t_n]]$ *will preserve scope. Furthermore,* Σ_{surf} *is least (it is contained in every other set of scoping rules that would be preserved).*

Proof. Given in the supplement. □

Corollary 5.5 (Desugaring preserves scope).
Let $\Sigma_{surf} = inferScope\left(\Sigma_{core}, \{C_i \Rightarrow C_i'\}_{i\in 1..n}\right)$. *Then desugaring with the rules* $\{C_i \Rightarrow C_i'\}_{i\in 1..n}$ *will preserve scope.*

Furthermore, scope inference runs in time $O(\Sigma_{P\in surf}\, arity\,(P)^3)$:

Lemma 5.6. $inferScope\,(\Sigma, \mathbb{C})$ *runs in time* $O(size\,(\mathbb{C}) + \Sigma_{P\in surf}\, arity\,(P)^3)$.

Proof. The running time of *inferScope* is dominated by *solve*, which in turn is dominated by two operations: iterating over the facts in $\mathbb{C}$, and adding facts to Σ_{surf}. Iterating over the facts in $\mathbb{C}$ takes time $size\,(\mathbb{C})$, where $size\,(\mathbb{C})$ is the total number of facts in $\mathbb{C}$. Each fact added to Σ_{surf} requires maintaining the transitive closure of Σ_{surf}, for the node type P of the fact. This can be done with an amortized cost of $O(arity\,(P))$ per P-fact added. (To add a fact $a \leqslant b \in \Sigma[P]$ that does not appear in Σ_{surf}, insert it and then recursively add $a \leqslant c \in \Sigma[P]$ for every fact $b \leqslant c \in \Sigma[P]$, and add $c \leqslant b \in \Sigma[P]$ for every fact $c \leqslant a \in \Sigma[P]$.) Since there are $O(arity\,(P)^2)$ possible P-facts to add, this adds an additional $O(\Sigma_{P\in surf}\, arity\,(P)^3)$ running time. □

The cubic parameter is concerning, but not a problem in practice for a number of reasons. First, $arity\,(P)$ tends to be small. Second, this algorithm is run off-line, and once per language. Finally, as we discuss in section 6, in practice the running time is extremely small.

6 IMPLEMENTATION AND EVALUATION

We have implemented the scope inference algorithm. Beyond what is shown in the paper, the implementation also allows (i) marking variables as *global references* that should refer to globally available identifiers in the expanded program, such as `print`, and (ii) a select form of copying a hole, where the hole contains a declaration and the copy is meant to be a reference of the same name. We will submit this implementation for artifact evaluation.

Besides the examples shown earlier in the paper, we have tested this implementation on sugars from three languages:

- All of the sugars that bind values in the Pyret language (pyret.org): namely `for` expressions, `let` statement clustering (nested bindings are grouped into a single `let`), and `function` declarations.
- Haskell list comprehensions, which include guards, generators, and local bindings.
- All of the sugars that bind values in R5RS Scheme [Kelsey et al. 1998]: namely `let`, `let*`, `letrec`, and `do`.

Some of the desugarings use ellipses in their definition, and thus had to be translated to match our fixed-arity assumption. (To do so, we introduced auxiliary AST node types and used those to express the equivalent looping.) `letrec` required one further adjustment to successfully infer scope.[10] After that, our tool successfully inferred scope for all of the sugars except for Scheme's do. In the rest of this section, we will describe many of these sugars in more detail, ending with do.

In practice, the running times are very modest. In our implementation in Rust (rust-lang.org) all of the sugars we have tested run in about 130ms on a generic desktop, of which 60ms is parsing time. Therefore, the speed is even fast enough for scope inference to be used as part of a language developer's rapid prototyping workflow.

6.1 Case Study: Pyret `for` Expressions

Consider the "for expressions" of the Pyret language:

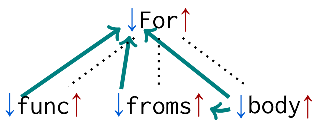
```
for fold(p from 1, n from range(1, 6)):
  p * n
end # Produces 5! = 120
```

This example desugars into:

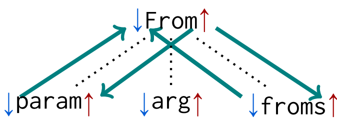
```
fold(lam(p, n): p * n end, 1, range(1, 6))
```

In general, the `for` syntax takes a function expression, any number of `from` clauses, and a body. It desugars into a call to the function, passing it as arguments (i) a lambda whose parameters are the LHSs of each `from` and whose body is the body of the `for`, and (ii) the RHS of each `from`.

Our system produces the following scoping rules for `for`, shown both textually and pictorially:[11]




```
import func          ∈ Σ[For]
import froms         ∈ Σ[For]
import body          ∈ Σ[For]
bind froms in body   ∈ Σ[For]
```




```
import param ∈ Σ[From]
import arg   ∈ Σ[From]
import froms ∈ Σ[From]
export param ∈ Σ[From]
export froms ∈ Σ[From]
```

6.2 Case Study: Haskell List Comprehensions

Haskell list comprehensions consist of sugar for *boolean guards* that filter the list, *generators* that specify the domain of the elements in the list, and *local bindings*. To quote the language standard [Simon Peyton Jones 2003, section 3.11]: "List comprehensions satisfy these identities, which may be used as a translation into the kernel:"

---

[10] The change was to have the desugaring distinguish between the letrec having zero bindings or one-or-more bindings. This prevented a fact of the form `bind i in i` from being applied to the binding list of the desugared `let`, which would make its bindings recursive. We have not found a principled account for why this was necessary.

[11] In the textual representation of the scoping rules, we give names to a node's children. Formally, these should be indices.

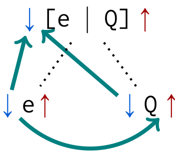
```
[ e | True ]           =  [e]                              (Base case)
[ e | q ]              =  [ e | q, True ]                  (Base case)
[ e | b, Q ]           =  if b then [ e | Q ] else []      Boolean guards
[ e | p <- l, Q ]      =  let ok p = [ e | Q ]
                                ok _ = []
                           in concatMap ok l               Generators
[ e | let decls, Q ]   =  let decls in [ e | Q ]           Local bindings
```

"where *e* ranges over expressions, *p* over patterns, *l* over list-valued expressions, *b* over boolean expressions, *decls* over declaration lists, *q* over qualifiers, and *Q* over sequences of qualifiers."

For example, the perfect numbers (numbers equal to the sum of their divisors) can be calculated by:

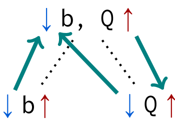
```
[n | n <- [1..], let d = divisors n, sum d == n]
```

Our system successfully infers the scope of these sugars. We will describe them one at a time. First, list comprehensions `[e | Q]` consist of an expression *e* and a list of *qualifiers* *Q*. Any declarations exported by *Q* (such as *n* above) should be in scope at *e*:



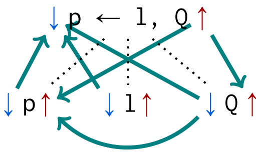

```
import e       ∈ Σ[ListComprehension]
import Q       ∈ Σ[ListComprehension]
bind Q in e    ∈ Σ[ListComprehension]
```

Boolean guards b, Q have a boolean expression *b* that is used to filter the list, and a sequence of more qualifiers *Q*. The scope of a boolean guard expression is simple: besides lexical scope, any declarations from *Q* are exported:



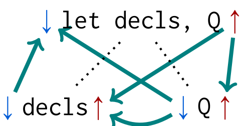

```
import b ∈ Σ[LC_Guard]
import Q ∈ Σ[LC_Guard]
export Q ∈ Σ[LC_Guard]
```

A generator expression `p ← l, Q` binds elements of list *l* to pattern *p*. *p* is bound in *Q*, and the declarations of both *p* and *Q* are exported:




```
import p       ∈ Σ[LC_Generator]
import l       ∈ Σ[LC_Generator]
import Q       ∈ Σ[LC_Generator]
bind p in Q    ∈ Σ[LC_Generator]
export p       ∈ Σ[LC_Generator]
export Q       ∈ Σ[LC_Generator]
```

Finally, local bindings *decls* are bound in the rest of the qualifiers *Q*, and also exported:




```
import decls       ∈ Σ[LC_Let]
import Q           ∈ Σ[LC_Let]
bind Q in decls    ∈ Σ[LC_Let]
export decls       ∈ Σ[LC_Let]
export Q           ∈ Σ[LC_Let]
```

6.3 Case Study: Scheme's Named-Let

The Scheme language standard defines two variants of the `let` sugar. The regular variant of `let` has the syntax `(let ((x val) ...) body)`, and binds each declaration `x` to the corresponding `val` in `body`. The scope of this variant can be inferred similarly to how we inferred the scope of `let*` in section 2.

The other variant is called "named" let. Its syntax is `(let f ((x val) ...) body)`, and it behaves like the regular `let` except that it additionally binds `f` to `(lambda (x ...) body)`. It can thus be used for recursive computations, such as reversing a list:

```
(define (reverse lst)
  (let rev ([unreversed lst]
            [reversed empty])
    (if (empty? unreversed)
        reversed
        (rev (cdr unreversed)
             (cons (car unreversed) reversed)))))
```

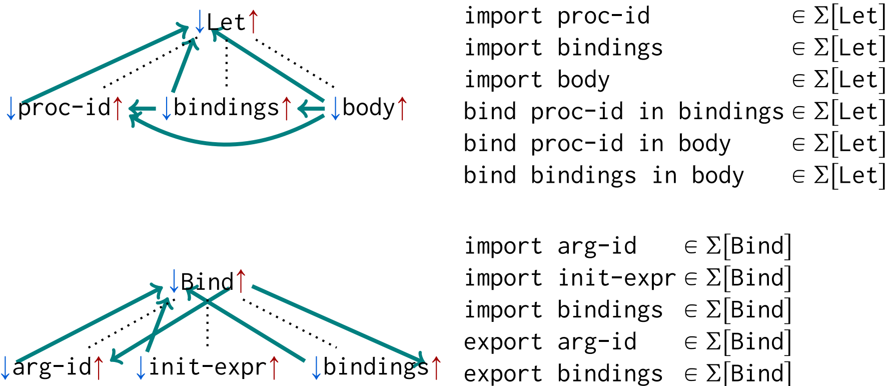
Named-let desugars by the rule:[12]

```
(define-syntax-rule
  ;;; The named let sugar:
  (let proc-id ([arg-id init-expr] ...) body)
  ;;; Desugars into:
  (letrec ([proc-id (lambda (arg-id ...) body)])
    (proc-id init-expr ...)))
```

We will represent the AST for named-let expressions with the grammar:

t	::=	(Let x^{D} b t)	"Named-let: bind initial values b and recursive function x^{D} in t"
	|	...	
b	::=	(Bind x^{D} t b)	"Bind x^{D} to t, and bind b"
	|	EndBinds	"No more bindings"

Translating the desugaring to use this grammar, our system correctly infers the binding structure:



[12] We describe Racket's desugaring because it is slightly more clear (using better variable names, and putting the application inside of the `letrec`). These differences have no effect on scope inference.

While this correctly reflects the scoping of named-let, observe that it permits the let-bindings to shadow the function name. This follows because `(arg-id ...)` can shadow `proc-id` in the macro definition. Of course, if a program actually did this, it would render the named part of the named-let useless! Nevertheless, we faithfully reflect the language, and indeed our inferred scope may be a useful diagnostic to the language designer.

6.4 Case Study: Scheme's do

Scheme's do expression can be used to perform what `do-while` and `for` loops might do in another language. For instance, this do expression reads three numbers off of `stdin`, before displaying their sum.

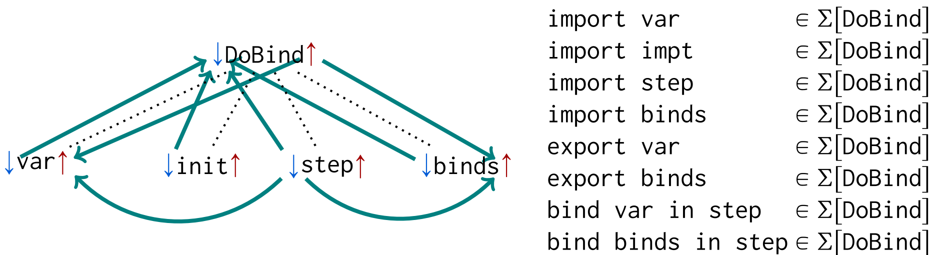
```
(do ((sum 0)
     (i 0 (+ i 1)))
  ((= i 3) (display "The sum is: ") (display sum) (newline))
    (set! sum (+ sum (string->number (read-line)))))
```

In general, do binds a list of variables [`sum` and `i`] to initial values [`0` and `0`], and then repeatedly evaluates the body of the loop [`(set! sum ...)`] and updates the variables according to optional step expressions [`(+ i 1)`] until a condition [`(= i 3)`] is met, at which point it evaluates a final sequence of expressions [`(display "The sum is: ") (display sum) (newline)`].

The desugaring of do is given by [Kelsey et al. 1998, derived forms]:

```
(define-syntax do (syntax-rules ()
  ((do ((var init step ...) ...)
       (test expr ...)
       command ...)
  (letrec ((loop (lambda (var ...)
                   (if test
                     (begin #f expr ...)
                     (begin command ...
                            (loop (do "step" var step ...) ...))))))
    (loop init ...)))
  ((do "step" x) x)
  ((do "step" x y) y)))
```

We will focus on the scope of the binding list, as scope inference fails on it. Its correct scope is:



While our binding language can express this scope, our algorithm is unable to handle inferring scope for it: it incorrectly infers that `var` is in scope at `init`. In more detail, whatever a desugaring does, it must at some point take apart the binding list. However, once one of the declarations `var` has been removed from the list, it must have a path to the rest of the list. Unfortunately that

path will put both `init` *and* `step` in scope of it. Therefore we cannot infer scope for this macro. In general, we cannot handle binding lists in which the bindings are visible in some expressions within the list (`step`) but not others (`init`).

This can naturally be fixed by putting `do` in the core language, but can also be addressed by altering the *syntax* slightly: separating the `init` list from the `step` list (which are semantically different entities) in the AST would avoid this unwanted conflation. More broadly, however, we believe that extending scope inference to work on desugaring rules with ellipses can solve this problem directly, as it is only the intermediate steps where the binding list is deconstructed that pose a problem. This raises questions that we leave for future work.[13]

6.5 Catalog of Scoping Rules (Extended in Supplement)

In the supplement, we show diagrams of many scoping rules that can be expressed in our binding language. These rules include functions, let-statements, letrec-statements, nested pattern-matching, and all of the sugars that bind values in a featureful language in use, Pyret.

7 PROOF OF HYGIENE

We will show that our scope inference algorithm (when successful) always produces surface scoping rules such that desugaring is hygienic. Again, we say that a desugaring f is hygienic when it preserves α-equivalence:

$$\Sigma_{surf} \vdash s =_\alpha t \quad \text{implies} \quad \Sigma_{core} \vdash f(s) =_\alpha f(t)$$

We will show this by way of a theorem that provides a necessary and sufficient condition for hygiene, assuming that desugaring obeys our assumptions. Recall that our definition of α-equivalence is strong, including that both terms are well-bound; thus we will need to show that the result of desugaring remains well-bound (so long as its input is).

To discuss the properties required for this theorem, we will divide variables into categories: variables in $f(t)$ are either New (fresh) or Copied from t, and variables in t are either Used (if they were copied) or Unused otherwise. Formally, let ϕ be the mapping from *copied* variables in $f(t)$ to their sources in t, and:

$$\begin{array}{lcl} \mathit{Used} & \triangleq & \mathit{range}\,(\phi) \\ \mathit{Unused} & \triangleq & \mathit{vars}\,(t) - \mathit{range}\,(\phi) \\ \mathit{Copied} & \triangleq & \mathit{domain}\,(\phi) \\ \mathit{New} & \triangleq & \mathit{vars}\,(f(t)) - \mathit{domain}\,(\phi) \end{array}$$

(where $\mathit{vars}\,(t)$ is the set of *all* variables in t).

We now turn to the properties required for hygiene. We will first list some clearly necessary properties, and then show that they are also sufficient.

First, f must avoid variable capture; thus $f(t)$ cannot contain bindings between new variables (introduced by f) and copied variables (taken from t):

Property 1.

$$\begin{array}{lll} \forall x^{\boldsymbol{R}} \in \mathit{Copied}. & \forall x^{\boldsymbol{D}} \in \mathit{New}. & \Sigma, f(t) \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}} \\ \forall x^{\boldsymbol{R}} \in \mathit{New}. & \forall x^{\boldsymbol{D}} \in \mathit{Copied}. & \Sigma, f(t) \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}} \end{array}$$

Second, f must preserve binding structure among the variables it copies:[14]

[13] What do scope specifications over arbitrary-length lists look like? Can the ith element be bound in the $(i+1)$st element (e.g., for `let*`)? How about $(i+1)$ in i, or i in j for all i and j? How does scope inference handle these cases, while still being correct, fast, and hygienic?

[14] This property does not appear in prior work on hygiene because such work typically assumes that the binding structure of a surface term is *defined by* the binding structure of its desugaring, causing this property to be true by definition.

Property 2.

$$\begin{array}{l}\forall x^{\boldsymbol{R}}, x^{\boldsymbol{D}} \in \mathit{Copied}.\\ \Sigma, f(t) \vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}} \text{ iff } \Sigma, t \vdash \phi(x^{\boldsymbol{R}}) \mapsto \phi(x^{\boldsymbol{D}})\end{array}$$

Finally, f must preserve well-boundedness. Thus, it must not cause a reference to become unbound or introduce a new unbound reference:

Property 3.

$$\begin{array}{lll}\forall x^{\boldsymbol{R}} \in \mathit{Used}. & \forall x^{\boldsymbol{D}} \in \mathit{Unused}. & \Sigma, t \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\\ \forall x^{\boldsymbol{R}} \in \mathit{New}. & \exists! x^{\boldsymbol{D}} \in \mathit{New}. & \Sigma, f(t) \vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\end{array}$$

While these three properties are clearly necessary, it is by no means clear that they are sufficient to guarantee α-equivalence preservation. However, the following theorem shows that they are both necessary and sufficient to ensure that f preserves α-equivalence:

Theorem 7.1 (Fundamental Hygiene Theorem). *Let f be a desugaring function over terms with respect to scoping rules Σ that obeys the assumptions of section 5.1. Then f respects α-equivalence iff for every (well-bound) input term t (numbering by property number):*

$$\begin{array}{llll}1. & \forall x^{\boldsymbol{R}} \in \mathit{Copied}. & \forall x^{\boldsymbol{D}} \in \mathit{New}. & \Sigma, f(t) \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\\ 1. & \forall x^{\boldsymbol{R}} \in \mathit{New}. & \forall x^{\boldsymbol{D}} \in \mathit{Copied}. & \Sigma, f(t) \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\\ 2. & \forall x^{\boldsymbol{R}} \in \mathit{Copied}. & \forall x^{\boldsymbol{D}} \in \mathit{Copied}. & \Sigma, f(t) \vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\\ & & & \text{iff } \Sigma, t \vdash \phi(x^{\boldsymbol{R}}) \mapsto \phi(x^{\boldsymbol{D}})\\ 3. & \forall x^{\boldsymbol{R}} \in \mathit{Used}. & \forall x^{\boldsymbol{D}} \in \mathit{Unused}. & \Sigma, t \not\vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\\ 3. & \forall x^{\boldsymbol{R}} \in \mathit{New}. & \exists! x^{\boldsymbol{D}} \in \mathit{New}. & \Sigma, f(t) \vdash x^{\boldsymbol{R}} \mapsto x^{\boldsymbol{D}}\end{array}$$

where ϕ is the mapping from copied *variables in $f(t)$ to their sources in t.*

Proof. Given in the supplement. □

We can now see that *inferScope* is hygienic. Property 1 is easily ensured by giving variables fresh names, which is one of our assumptions about desugaring. Property 2 follows from our inference process: since desugaring preserves scope by theorem 5.4, bindings between variables must not change. Finally, property 3 is exactly what *checkScope* checks.

8 RELATED WORK

Our work bears some similarity to that of Pombrio and Krishnamurthi [2014, 2015] on resugaring: take part of a core language and lift or *resugar* it to the surface language. However, they lift *evaluation steps* though sugar *dynamically*, while we lift *scoping rules* through sugar *statically*.

Below, we discuss work related to two aspects of our approach to scope inference. However, none of them infer scope through syntactic sugar; we therefore believe that the central contribution of this paper is novel.

Hygienic Expansion

The real goal of hygienic expansion is to preserve α-equivalence: α-renaming a program should not change its meaning. Typically, however, α-equivalence is only *defined* for the core language. Thus, traditional approaches to hygiene have had to focus on avoiding specific issues like variable capture [Kohlbecker et al. 1986]. Recent work by Adams [2015] advances the theory by giving an algorithm-independent set of issues to avoid. However, even this work lacks the ground truth of *alpha*-equivalence preservation to base its claims on.

In contrast, Herman and Wand [2008] advocate that sugar specify the binding structure of the constructs they introduce, and build a system that does so. Stansifer and Wand [2014] follow

with a more powerful system called Romeo based on the same approach. (We will discuss the binding languages used by these two tools in the next subsection.) Since we *infer* scope rules for the surface language, we can verify that desugaring preserves α-equivalence without requiring scope annotations on sugars.

Erdweg et al. [2014] put forward an interesting alternative approach to hygiene with the *name-fix* algorithm. *Name-fix* assumes that the scoping for the surface language is known. Instead of using this information to *avoid* unwanted variable capture in the first place, *name-fix* uses it to *detect* variable capture and rename variables as necessary to repair it after the fact. Erdweg et al. prove that *name-fix* preserves α-equivalence, but for a *weaker* definition of α-equivalence than ours that doesn't include well-boundedness (thus allowing desugaring to produce unbound variables).

Our work differs from the above work: we assume that scope is defined *only* for the core language, and not for the surface language (à la Erdweg) or for individual rewrite rules (à la Herman). This assumption is also made by traditional capture-avoiding work on hygiene. However, by inferring scoping rules from the core to the surface language, we gain two benefits: (i) we can prove that our approach is correct with respect to the *ground truth* of α-equivalence preservation (theorem 7.1), and (ii) we can produce a set of standalone scoping rules for the surface language. To our knowledge, this approach has not been taken before.

Scope

We will divide related work on scoping into two main categories. First, "Modeling Scope" discusses ways in which the scope of a term can be *represented*. Our description of scope as a preorder (section 3) falls in this category. Second, "Binding Specification Languages" discusses ways in which scope can be *determined* for a given term. Our binding language (section 4) falls in this category.

Modeling Scope. Our description of scope-as-a-preorder is similar to the view expressed by Flatt [2016] in "Binding as Sets of Scopes".[15] In fact, Flatt's notion of scope can be expressed as a preorder, as we show in section 3 of the supplement.

Neron et al. [2015] describe *scope graphs*, which are also based on a similar view, but have a more complicated set of definitions. Unlike scope-as-a-preorder, however, scope graphs include mechanisms for handling module scope, which gives it the ability to model both modules and also other constructs like objects and field lookup. Our scope-as-a-preorder binding language can actually be extended to handle modules, but doing so breaks our transitivity assumption, which we need to infer scope, so we have left it out of this paper and consider this a problem for future work.

Binding Specification Languages. Our preorder-based binding specification language is novel, but similar in expressiveness to many others. It is perhaps most similar to Stansifer and Wand [2014]'s Romeo. The primary difference between the two is that Romeo has slightly more expressive power: given two declarations $x_1^{\mathbf{D}}$ and $x_2^{\mathbf{D}}$, it is possible in Romeo for $x_1^{\mathbf{D}}$ to shadow $x_2^{\mathbf{D}}$ in one part of a term, but $x_2^{\mathbf{D}}$ to shadow $x_1^{\mathbf{D}}$ in a different part of a term.[16] It is not clear if this power has any practical applications, but we choose to avoid it both for aesthetic reasons (we do not believe two declarations should be allowed to shadow one another), and to simplify scope inference (which would otherwise have to manipulate formulas over Romeo's combinators, instead of merely preorders).

In a similar vein, Sewell et al. [2010] present a semantics engineering workbench called Ott, which includes a comparable binding specification language. Like Romeo, Ott would allow two declarations to each shadow one another in different places. Furthermore, it gives additional power,

[15] These were discovered independently: scope-as-a-preorder arose from some of the ideas from Romeo [Stansifer and Wand 2014].

[16] In Romeo, this would be expressed using the $\triangleright$ combinator, as $\beta_1 \triangleright \beta_2$ and $\beta_2 \triangleright \beta_1$.

by allowing terms to *name* what they provide. For instance, a term could export *two* binding lists, one named "value-bindings" and one named "type-bindings".

Weirich et al. [2011] present a binding specification language called Unbound, which can be expressed using scope-as-a-preorder (and hence is no more expressive than it). They implement Unbound in Haskell, and give language-agnostic implementations of operations such as constructing and deconstructing terms, determining α-equivalence, and performing substitution. In Unbound, binding is specified via a set of *binding combinators*. These binding combinators can be expressed as a preorder.[17]

There are many more binding specification languages [Aczel 1978; Konat et al. 2012; Pottier 2005]. We have chosen what we believe to be a representative sample for comparison. We have shown that our binding specification language compares favorably in expressiveness, while simultaneously being simple enough to enable scope inference.

Conclusion and Future Work

We have presented what we believe is the first algorithm for inferring scoping rules through syntactic sugar. It makes use of our description of scope as a preorder in section 3, and our binding language for specifying the scope of a programming language in section 4. The case studies in section 6 show that all of the aspects of this paper are able to deal with many interesting scoping constructs from real languages.

We see three clear directions in which to try to extend scope inference. First, support for ellipses in sugar definitions would make writing sugars easier. Second, allowing named imports and exports—à la Ott [Sewell et al. 2010]—would make sugars like `do` inferable. Third, modules—à la Scope Graphs [Neron et al. 2015]—are necessary for inferring scope for modules and for classes. These last two changes are relatively straightforward extensions to our binding language, but research questions when applied to scope inference.

ACKNOWLEDGMENTS

We thank Paul Stansifer, Sebastian Erdweg, and Matthew Flatt for their feedback, and the anonymous reviewers who provided helpful feedback on the paper. This work was partially supported by the NSF.

[17] The translation of Unbound to scope-as-a-preorder is as follows. `Name` constructs a declaration or reference, depending on whether it is a Term or Pattern. Patterns P have scoping rules that state `export` $i \in \Sigma[P]$ and `import` $i \in \Sigma[P]$ for every i. Terms T have scoping rules that state `import` $i \in \Sigma[T]$ for every i. Finally, each of the four binding combinators obey the scoping rule for patterns or for terms, as appropriate, in addition to the following facts:

Bind P T	{`bind 1 in 2` $\in \Sigma$[Bind]}	Embed T	$\varnothing$
Rebind P P	{`bind 1 in 2` $\in \Sigma$[Rebind]}	Rec P	{`bind 1 in 1` $\in \Sigma$[Rec]}

REFERENCES

Peter Aczel. 1978. *A General Church-Rosser Theorem.* Technical Report. University of Manchester.

Michael D. Adams. 2015. Towards the Essence of Hygiene. In *Principles of Programming Languages.* ACM, New York, NY, USA, 457–469. DOI: http://dx.doi.org/10.1145/2676726.2677013

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *European Conference on Object-Oriented Programming.* Springer-Verlag, Berlin, Heidelberg, 489–514. DOI: http://dx.doi.org/dx.doi.org/10.1007/978-3-662-44202-9_20

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Krishnamurthi Shriram, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12 (2002), 159–182. DOI: http://dx.doi.org/10.1017/S0956796801004208

David Fisher and Olin Shivers. 2006. Static analysis for syntax objects. In *International Conference on Functional Programming.* ACM, New York, NY, USA.

Matthew Flatt. 2016. Binding as Sets of Scopes. In *Principles of Programming Languages.* ACM. DOI: http://dx.doi.org/10.1145/2914770.2837620

David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *European Symposium on Programming Languages and Systems.* Springer-Verlag, Berlin, Heidelberg, 48–62. DOI: http://dx.doi.org/10.1007/978-3-540-78739-6_4

R. Kelsey, W. Clinger, and J. Rees (eds.). 1998. *Revised [5] Report on the Algorithmic Language Scheme.* Cambridge University Press.

J. W. Klop. 1992. Term Rewriting Systems. *Handbook of Logic in Computer Science* (1992). DOI: http://dx.doi.org/citation.cfm?id=162559

Donald E. Knuth. 1968. Semantics of Context-Free Languages. In *Mathematical Systems Theory.* Springer-Verlag. DOI: http://dx.doi.org/10.1007/BF01692511

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *ACM Conference on LISP and Functional Programming.* ACM, New York, NY, USA, 11. DOI: http://dx.doi.org/10.1145/319838.319859

Gabrie Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering.* Springer-Verlag, Berlin, Heidelberg, 311–331. DOI: http://dx.doi.org/10.1007/978-3-642-36089-3_18

Florian Lorenzen and Sebastian Erdweg. 2013. Modular and Automated Type-Soundness for Language Extensions. In *International Conference on Functional Programming.* ACM, New York, NY, USA, 12. DOI: http://dx.doi.org/10.1145/2544174.2500596

Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *European Symposium on Programming Languages and Systems.* Springer-Verlag, Berlin, Heidelberg, 205–231. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_9

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. In *Programming Languages Design and Implementation.* ACM, New York, NY, USA, 361–371. DOI: http://dx.doi.org/10.1145/2594291.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. In *International Conference on Functional Programming.* ACM, New York, NY, USA, 75–87. DOI: http://dx.doi.org/10.1145/2784731.2784755

François Pottier. 2005. An Overview of Cαml. In *Electronic Notes in Theoretical Computer Science.* DOI: http://dx.doi.org/10.1016/j.entcs.2005.11.039

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of Functional Programming* (2010). DOI: http://dx.doi.org/10.1017/S0956796809990293

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

Paul Stansifer and Mitchell Wand. 2014. Romeo: a System For More Flexible Binding-Safe Programming. In *International Conference on Functional Programming.* ACM, New York, NY, USA, 53–65. DOI: http://dx.doi.org/10.1145/2628136.2628162

Stephanie Weirich, Brent Yorgey, and Tim Sheard. 2011. Binders Unbound. In *International Conference on Functional Programming.* ACM, New York, NY, USA, 333–345. DOI: http://dx.doi.org/10.1145/2034574.2034818