Syntactic sugar is pervasive in language technology. Programmers use it to shrink the size of a core language; to define domain-specific languages; and even to extend their language. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user’s source program and the transformed program. First, it obscures the evaluation steps the program takes when it runs, since these evaluation steps happen in the core (desugared) language rather than the surface (pre-desugaring) language the program was written in. Second, it obscures the scoping rules for the surface language, making it difficult for IDEs and other tools to obtain binding information. And finally, it obscures the types of surface programs, which can result in type errors that reference terms the programmer did not write. I address these problems by showing how evaluation steps, scoping rules, and type rules can all be lifted—or resugared—from core to surface languages, thus restoring the abstraction provided by syntactic sugar.
Resugaring:
Lifting Languages through Syntactic Sugar

by
Justin Pombrio
B. S., Worcester Polytechnic Institute, 2011

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2018
© Copyright 2018 by Justin Pombrio
This dissertation by Justin Pombrio is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________    ___________________________________________

Shriram Krishnamurthi, Director

Recommended to the Graduate Council

Date ______________    ___________________________________________

Mitchell Wand, Reader
Northeastern Computer Science

Date ______________    ___________________________________________

Eelco Visser, Reader
TU Delft Computer Science

Approved by the Graduate Council

Date ______________    ___________________________________________

Dean of the Graduate School
ACKNOWLEDGEMENTS

To Dad, for being my foundation.

To Nick and Alec and Andy, for building worlds with me.

To Joey and Eric and Chris, for making high-school fun.

To the board-gamers, tea-goers, movie-goers, and puzz-lers. I’d name you all, but you’re too many and I’d leave someone out. You know who you are.

To Evan, for all the hugs.

To Roberta Thompson, Dan Dougherty, and Joshua Guttman, for teaching me how to reason.

To Shriram, for teaching me how to write, how to present, how to critique, how to be a researcher.

To Mitchell Wand and Eelco Visser, for overseeing this large and technical body of work.

To Sorawee “Oak” Porncharoenwase, for diving deep into Pyret with me.

To the NSF, which partially funded all of this work.

To the many anonymous, unpaid reviewers who gave me detailed feedback on my work.
CONTENTS

List of Figures xi

1 SYNTACTIC SUGAR

1.1 What is it? 3
1.2 What is it good for? 4
1.3 When should you use it? 5
1.4 What are its downsides? 6
1.5 How can these (particular) downsides be fixed? 7
1.6 Roadmap 7

II DESUGARING

2 DESUGARING IN THE WILD! 11

2.1 A Sugar Taxonomy 11

2.1.1 Representation 12
2.1.2 Authorship: Language-defined or User-defined 13
2.1.3 Metalanguage 13
2.1.4 Desugaring Order 13
2.1.5 Time of Expansion 14
2.1.6 Expressiveness 14
2.1.7 Safety 15

2.2 Applying the Taxonomy to Desugaring Systems 17

2.2.1 Early Lisp Macros 17
2.2.2 Scheme Macros 17
2.2.3 Racket Macros 19
2.2.4 McMicMac Micros 19
2.2.5 C Preprocessor 20
2.2.6 C++ Templates 21
2.2.7 Template Haskell 22
2.2.8 Haskell Sugars 22
2.2.9 Rust Macros 23
2.2.10 Julia Macros 23
2.2.11 MetaOCaml 23
2.2.12 Resugaring Systems 24

3 NOTATION 25

3.1 ASTs 25
3.2 Desugaring  26
   3.2.1 Restrictions on Desugaring  27
   3.2.2 Relationship to Term Rewriting Systems  28

III RESUGARING

4 RESUGARING EVALUATION SEQUENCES  31
   4.1 Our Approach  31
   4.2 Informal Solution Overview  32
      4.2.1 Finding Surface Representations Preserving Emulation  32
      4.2.2 Maintaining Abstraction  33
      4.2.3 Striving for Coverage  34
      4.2.4 Trading Abstraction for Coverage  35
      4.2.5 Maintaining Hygiene  36
   4.3 Confection at Work  36
   4.4 The Transformation System  37
      4.4.1 Performing a Single Transformation  37
      4.4.2 Performing Transformations Recursively  43
      4.4.3 Lifting Evaluation  45
   4.5 Formal Justification  46
      4.5.1 Transformations as Lenses  46
      4.5.2 Desugar and Resugar are Inverses  48
      4.5.3 Ensuring Emulation and Abstraction  49
      4.5.4 Machine-Checking Proofs  51
   4.6 Obtaining Core-Language Steppers  51
   4.7 Evaluation  53
      4.7.1 Building on the Lambda Calculus  53
      4.7.2 Return  53
      4.7.3 Pyret: A Case Study  55
   4.8 Obtaining Hygiene  58
   4.9 Related Work  59

5 RESUGARING SCOPE RULES  63
   5.1 Introduction  63
   5.2 Two Worked Examples  65
      5.2.1 Example: Single-arm Let  65
      5.2.2 Example: Multi-arm Let*  67
      5.2.3 Scope as a Preorder  69
   5.3 Describing Scope as a Preorder  70
      5.3.1 Basic Assumptions  70
      5.3.2 Basic Definitions  71
      5.3.3 Validating the Definitions  72
### 5.4 A Binding Specification Language

- 5.4.1 Scoping Rules: Simplified
- 5.4.2 A Problem
- 5.4.3 The Solution
- 5.4.4 Scoping Rules: Un simplified
- 5.4.5 Well-Boundedness

### 5.5 Inferring Scope

- 5.5.1 Assumptions about Desugaring
- 5.5.2 Constraint Generation
- 5.5.3 Constraint Solving
- 5.5.4 Ensuring Hygiene
- 5.5.5 Discussion of the Hygiene Property
- 5.5.6 Correctness and Runtime

### 5.6 Implementation and Evaluation

- 5.6.1 Case Study: Pyret for Expressions
- 5.6.2 Case Study: Haskell List Comprehensions
- 5.6.3 Case Study: Scheme’s Named-Let
- 5.6.4 Case Study: Scheme’s do
- 5.6.5 Catalog of Scoping Rules (Extended)

### 5.7 Proof of Hygiene

### 5.8 Related Work

- 5.8.1 Hygienic Expansion
- 5.8.2 Scope

### 5.9 Discussion and Future Work

### 6 Resugaring Types

- 6.1 Introduction
- 6.2 Type Resugaring
- 6.3 Theory
  - 6.3.1 Requirements on Desugaring
  - 6.3.2 Requirements on the Type System
  - 6.3.3 Requirements on Resugaring
  - 6.3.4 Main Theorem
- 6.4 Desugaring Features
  - 6.4.1 Calculating Types
  - 6.4.2 Recursive Sugars
  - 6.4.3 Fresh Variables
  - 6.4.4 Globals
  - 6.4.5 Variable Arities
6.5 Implementation 128
6.6 Evaluation 129
6.6.1 Type Systems 129
6.6.2 Case Studies 130
6.7 Related Work 133
6.8 Discussion and Conclusion 135
6.9 Demos 136
Bibliography 143
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bindings</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>Matching and substitution</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>Automaton macro execution example</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>Syntactic sugar in normal-mode Pyret</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>Alternate desugaring of addition</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>Scope Checking</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>Scope Inference Algorithm</td>
<td>85</td>
</tr>
<tr>
<td>8</td>
<td>Catalog of Scoping Rules (Arrows that follow from transitivity omitted)</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Proof of theorem 6</td>
<td>107</td>
</tr>
<tr>
<td>10</td>
<td>Proof of theorem 6 (continued)</td>
<td>108</td>
</tr>
<tr>
<td>11</td>
<td>Proof of theorem 6 (continued)</td>
<td>109</td>
</tr>
<tr>
<td>12</td>
<td>Proof of theorem 6 (continued)</td>
<td>110</td>
</tr>
<tr>
<td>13</td>
<td>Derivation of or. Top: an incomplete derivation. Bottom: a complete derivation, using t-premise.</td>
<td>115</td>
</tr>
<tr>
<td>14</td>
<td>Notation explanation</td>
<td>118</td>
</tr>
<tr>
<td>15</td>
<td>Proof of theorem 9</td>
<td>123</td>
</tr>
<tr>
<td>16</td>
<td>Type derivation of let</td>
<td>125</td>
</tr>
<tr>
<td>17</td>
<td>foreach type rule</td>
<td>131</td>
</tr>
<tr>
<td>18</td>
<td>Derivation examples</td>
<td>137</td>
</tr>
<tr>
<td>19</td>
<td>Derivation examples (cont.)</td>
<td>138</td>
</tr>
<tr>
<td>20</td>
<td>Sample SweetT usage</td>
<td>139</td>
</tr>
<tr>
<td>21</td>
<td>Sample sugars, pg.1</td>
<td>140</td>
</tr>
<tr>
<td>22</td>
<td>Sample sugars, pg.2</td>
<td>141</td>
</tr>
<tr>
<td>23</td>
<td>Sample sugars, pg.3</td>
<td>142</td>
</tr>
</tbody>
</table>
Part I

SYNTACTIC SUGAR
This chapter will introduce syntactic sugar, and work up to the thesis statement:

Many aspects of programming languages—in particular evaluation steps, scope rules, and type rules—can be non-trivially resugared from core to surface language, restoring the abstraction provided by syntactic sugar.

1.1 WHAT IS IT?

The term syntactic sugar was introduced by Peter Landin in 1964 [51]. It refers to surface syntactic forms that are provided for convenience, but could instead be written using the syntax of the rest of the language. This captures the spirit and purpose of syntactic sugar.

The name suggests that syntactic sugar is inessential: it “sweetens” the language to make it more palatable, but does not otherwise change its substance. The name also naturally leads to related terminology: desugaring is the removal of syntactic sugar by expanding it; and resugaring is a term that we will introduce in this thesis for recovering various pieces of information that were lost during desugaring.

As an example of a sugar, consider Java’s “enhanced for statement” (a.k.a., for-each loop), which can be used to print out the best numbers:

```java
for (int n : best_numbers) {
    System.out.println(n);
}
```

This enhanced for is quite convenient, but it isn’t really necessary, because you can always do the same thing with a regular for loop and an iterator:

```java
for (Iterator i = best_numbers.iterator(); i.hasNext(); ) {
    int n = (int) i.next();
    System.out.println(n);
}
```
And in fact the Java spec formally recognizes this equivalence, and writes, “The meaning of the enhanced for statement is given by translation into a basic for statement.” Ignoring some irrelevant details, it states that this sugar:

```java
for (<type> <var> : <expr>) { <statement> }
```
desugars into:

```java
for (Iterator i = <expr>.iterator(); i.hasNext(); ) {
<type> <var> = (<type>) i.next();
<statement>
}
```

Here, the things we have written in angle brackets are parameters to the sugar: they are pieces of code that it takes as arguments.

**BUT WHAT IS IT?** An astute reader may have noticed that we still haven’t actually defined syntactic sugar. Here is a reasonable definition:

A syntactic construct in an implementation of a programming language is **syntactic sugar** if it is translated at compile-time into the syntax of the rest of the language.

Thus syntactic sugar splits a language into two parts: a (small) core language and a rich set of usable syntax atop that core. In this thesis, we use the term **surface** to refer to the language the programmer sees, and **core** for the target of desugaring. This makes desugaring a part of compilation: it is compilation from a language to a subset of that language. It also makes clear that **macros** are a special case of syntactic sugar: they are a way of allowing users of a language to define syntactic sugar within the language itself.

### 1.2 WHAT IS IT GOOD FOR?

Syntactic sugar is an essential component of programming languages and systems, and it is now actively used in many practical settings:

- In the definition of language constructs in many languages ranging from Java to Haskell.
- To allow users to extend the language, in languages ranging from the Lisp family to C++ to Julia to Rust.
- To shrink the semantics of large scripting languages with many special-case behaviors, such as JavaScript and Python, to small core languages that tools can more easily process [32 65 66].
Desugaring allows a language to expose a rich surface syntax, while compiling down to a small core. Having a smaller core reduces the cognitive burden of learning the essence of the language. It also reduces the effort needed to write tools for the language or do proofs decomposed by program structure (such as type soundness proofs). Thus, heaping sugar atop a core is a smart engineering trade-off that ought to satisfy both creators and users of a language. Observe that this trade-off does not depend in any way on desugaring being offered as a surface linguistic feature (such as macros).

1.3 When Should You Use It?

Syntactic sugar is used to define abstractions. But languages have other ways to define abstractions already: functions, classes, data definitions, etc. If an abstraction can be implemented using these features, it’s almost always better to do so, because developers are already deeply familiar with them. Thus:

Syntactic sugar should only be used to implement an abstraction if it cannot be implemented in the core language directly.

Therefore, to find places where it is a good idea to use sugar, we should look for things that most programming languages cannot abstract over:

1. In most languages, variable names are first order and cannot be manipulated at run-time (e.g., a variable cannot be passed as an argument to a function: if you attempt to do so, the value the variable is bound to will be passed instead). Therefore, creating new binding constructs is a good use for syntactic sugar in most languages. However, in R, variable names can be abstracted over (e.g., “assign("x", 3); x” prints 3), and so sugar isn’t necessary for this purpose.

2. Most languages use eager evaluation, which forces the arguments to a function to be evaluated when it is called, rather than allowing the function to choose whether to evaluate them or not. Thus if an abstraction requires delaying evaluation, it is a good candidate to be a sugar. However, Haskell has lazy evaluation, and thus does not need sugars for this purpose.

3. Most languages cannot manipulate data definitions at run-time: e.g., field names cannot be dynamically constructed. Thus creating new data definition constructs (e.g., a way to define state machines) is a good use case for sugars. However, in Python, field names are first class (e.g., they can be
added or assigned using `setattr`, so it does not need sugars to abstract over fields in data definitions[1].

Overall, syntactic sugar is a way to extend a language. In a limited sense, this is what functions are for as well. Functions, however, are limited: they cannot take a variable as an argument, delay evaluation, introduce new syntax, etc. This is where sugar is sweetest.

1.4 WHAT ARE ITS DOWNSIDES?

There are two sets of downsides to syntactic sugar. The first set of downsides applies to languages that allow user-defined syntactic sugar (i.e., macros), and arises from the powerful nature of syntactic sugar. Sugars can manipulate many things such as control flow and variable binding that are otherwise fixed in a language. As a result, badly written sugars can lead to code that is convoluted in ways not otherwise possible. Furthermore, syntactic sugar (by design) hides details: programmers only ever see the sugar, and not the desugared code (except perhaps in error messages), and thus may not be fully aware of what sort of code they are implicitly writing. In short, it can be dangerous to allow users to extend a language, and this is exactly what macros allow. However, this is a language design issue and thus outside the scope of this thesis.

Instead, we address situations in which the abstraction provided by sugar leaks [77]. The code generated by desugaring can be large and complicated, creating an onerous comprehension burden; it may even use features of the core language that the user does not know. Therefore, programmers using sugar must not be forced to confront the details of the sugar; they should only confront the core language when they use it directly. This is a concern irrespective of whether sugars are defined as part of the language or whether users can define their own sugars.

We call out three particular ways that syntactic sugar breaks abstractions:

**Evaluation steps** Syntactic sugar obscures the evaluation steps the program takes when it runs, since these evaluation steps happen in the core language rather than the surface language the program was written in.

**Scope rules** In a similar manner, syntactic sugar obscures the (implicitly defined) scope rules for a surface language.

---

[1] It may sound like we are suggesting that everything should be manipulatable at run-time. We are not. The more things which are fixed at compile time (variable names, field names, etc.), the more (i) programmers can reason about their programs; (ii) tools can reason about programs; (iii) compilers can optimize programs (without herculean effort). It is **good** that sugar is sometimes necessary.
**1.5 How can these (particular) downsides be fixed?**

We address these three problems with a general approach called resugaring. Thus my thesis statement is that:

Many aspects of programming languages—in particular evaluation steps, scope rules, and type rules—can be non-trivially resugared from core to surface language, restoring the abstraction provided by syntactic sugar.

We hope that this work will give desugaring its rightful place in the programming language space, freeing future implementers and researchers to take full advantage of sugar in their semantics and systems work. Only when using syntactic sugar no longer inadvertently breaks abstractions can the adage “oh, that’s just syntactic sugar” finally become true.

**1.6 Roadmap**

The rest of this thesis is organized as follows:

**Part II: Desugaring**

Chapter 2 discusses and taxonomizes existing desugaring systems (of which there are a wide variety).

Chapter 3 provides notation that will serve as groundwork for the rest of the chapters.

**Part III: Resugaring**

Chapter 4 shows how to resugar evaluation sequences.

Chapter 5 shows how to resugar scope rules.

Chapter 6 shows how to resugar type rules.

Related work is discussed in each resugaring chapter, wherever it is most relevant.
Part II

DESUGARING
DESGARIN IN THE WILD

There are a bewildering variety of desugaring systems. In this chapter, we categorize a representative set of them into a taxonomy.

We stretch this taxonomy to include some systems that aren’t quite desugaring systems, but that are closely associated with them. Notably, we discuss staged metaprogramming systems, which (i) operate on code at run-time, rather than compile-time, and (ii) have to explicitly, rather than implicitly, desugar code.

2.1 A SUGAR TAXONOMY

There are many dimensions by which desugaring mechanisms vary:

REPRESENTATION Desugaring is a syntax-to-syntax transformation, but how is that syntax represented? There is a big difference between transformations on the text of the program vs. its concrete surface syntax vs. its abstract surface syntax.

AUTHORSHIP Are sugars defined by developers of the language (and thus relatively fixed), or by users of the language (and thus flexible)?

METALANGUAGE What is the metalanguage? That is, in what language are sugars written? Is it the same language the programs are written in, thus allowing sugars and code to be interspersed, or a different language?

DESGARING ORDER In what order are constructs desugared? Most importantly, are nested sugars desugared from the innermost to outermost (io), or from outermost to innermost (oi)?

TIME OF EXPANSION When does desugaring occur? Is it at compilation time, or at run-time (as in staged metaprogramming systems)?

We will consider four dimensions related to the expressiveness of desugaring systems:

PARAMETERS Can a sugar take an arbitrary expression as a parameter, or only a primitive value?

As previously noted, if the “phase” at which desugaring happens is run-time, then the system isn't really a desugaring system, though it may be similar enough to warrant our classification of it.
result Do sugars exclusively produce expressions, or can they produce any kind of syntax?

deconstruction If it can take an expression, can it deconstruct it, or only use it parametrically?

sugar-defining sugars Can sugars define other sugars?

We will also consider four dimensions related to safety:

syntactic safety Can desugaring produce syntactically invalid code?

hygiene Can a sugar accidentally capture a variable?

scope safety Can a sugar produce an unbound variable?

type safety Can a sugar produce code that contains a type error?

We next discuss each of these dimensions in detail. After that, table 1 assesses a number of desugaring systems along these dimensions, and section 2.2 discusses these desugaring systems.

2.1.1 Representation

There are many ways to represent a program. The most prevalent are as text and as a tree. Programs are most commonly saved as and edited as text (notable exceptions include languages or editors that are visual, block based, or projectional), and they are most commonly internally represented as trees (notable exceptions include assembly, whose code is linear, and Forth, which does not have a parsing phase).

Desugaring systems may be based on either representation. However, text is a terrible representation for desugaring rules. The semantics of a language is almost always defined in terms of its (abstract syntax) tree representation. Thus, insofar as a programmer is forced to think of their program as text rather than as a tree, they are being distracted from its semantics. There are well-known examples of bugs that arise in text-based desugaring rules unless they are written in a very defensive style: we discuss these in 2.2.5.

There are variations among tree representations as well: desugaring rules may work over the concrete syntax of the language, or over its abstract syntax. The concrete syntax of a language is its syntax as seen by the programmer, e.g. \( f(1) \) for a function application. The abstract syntax is the internal representation of that syntax, which for this example might (e.g.) be \( \text{apply} (\text{id } f) \ 1 \). Concrete syntax might seem similar to a text-based representation, but there is an important distinction: \( f(1 \text{ in a textual representation is part of a term, but there is no such thing as } f(1 \text{ in the concrete syntax: it is ill-formed.} \right) \)
Overall, this is a very loose categorization of representations: we are placing a wide variety of possible representations into a few large buckets.

### 2.1.2 Authorship: Language-defined or User-defined

Sugars may either be specified and implemented as part of the language, or they may be defined by users. For example, Haskell list comprehensions are defined by the Haskell spec \[61\] and implemented in the compiler(s); thus they are language-defined. Template Haskell sugars \[75\], on the other hand, can be defined (and used) in any Haskell program; thus they are user-defined.

Language-defined sugars are a convenient method of simplifying language design, and they are largely invisible: ideally, users of the language should not be able to tell which syntactic constructs were implemented as sugar and which were built in. In contrast, user-defined sugars are much more visible. They give users the power to extend the language. As a consequence, when this extended language is shared, other users must contend with it, which may be a gift or a burden, depending on its quality.

### 2.1.3 Metalanguage

The metalanguage is the language that the sugars are written in. There are a number of common possibilities:

- The metalanguage may be the host language itself.
- The metalanguage may be the implementation language of the compiler (which is frequently the same language).
- The metalanguage may be a set of pattern-based rules in the general style of Scheme’s `syntax-rules`. In table \[1\] we use “rules” as a shorthand for “a set of pattern-based desugaring rules”.

### 2.1.4 Desugaring Order

There are two major desugaring strategies used in desugaring systems. They loosely correspond to the call-by-value and call-by-name run-time evaluation strategies, but differ in some important ways, so we will instead refer to them by their original names: Outside-in (oi) and Inside-out (io) desugaring: \[21\]

**Outside-in**: oi desugaring is superficially similar to call-by-name evaluation, in that desugaring proceeds from the outside in. However, there is one
crucial difference: in call-by-name evaluation, if a function uses (e.g., does case analysis on) one of its arguments, it causes that argument to be evaluated. In contrast, if an or sugar does case analysis on one of its parameters (which are pieces of syntax), it does not force that parameter to be desugared! Rather, the case analysis happens on the uninterpreted syntax of that parameter.

There is an advantage to this: it allows sugars to truly extend the syntax of the language because they can treat the syntax of their parameters however they want. In the most extreme case, it may be impossible to even tell where the innermost sugars are, so or is the only possible desugaring order.

**inside-out:** or desugaring is similar to call-by-value evaluation: the innermost sugars desugar first. The above paragraph suggested that or order may not be possible, but there are a number of common settings in which it is: (i) the syntax that sugars can introduce is limited enough that you can always tell where nested sugars are; (ii) sugars act on the AST rather than on concrete syntax; or (iii) sugars cannot deconstruct their parameters, so the desugaring order is largely irrelevant anyways. In these settings, or order has the advantage of being more analogous to evaluation (which developers are quite familiar with). It is also sometimes useful for sugars to use the desugared version of their parameters: for example, C++ templates can be used to derive specialized implementations for particular types, and it is convenient to allow that type to be a template expression.

### 2.1.5 Time of Expansion

We distinguish between three different “times” of expansion:

**run** means that “sugars” are actually expanded at run-time. This only occurs for metaprogramming systems, which are not true desugaring systems.

**compile** means that sugars are always expanded in a separate phase before the program runs. (To be more precise, the phases that occur in order are (i) desugar, (ii) compile, and (iii) run. We describe this as “compile” for simplicity and because “compile-time” is a well-established phrase.)

**variety** Racket macros can run at an arbitrary number of different phases, including at compile-time and at run-time.

### 2.1.6 Expressiveness

We will examine four measures of expressiveness of each desugaring system:
PARAMETERS What kind of parameters can be passed to a sugar? In the most general desugaring systems, any syntactic category—be it expression, statement, type, field name, class definition, etc.—can be passed to a sugar. Some systems are more limited, however: for example, C++ templates can only take types or primitive values (e.g., numbers) as parameters.

RESULT What kind of syntax can a sugar desugar to? Can it be any syntactic category, or is it more limited, e.g., to expressions only? For example, C++ templates can only desugar into definitions, such as function definitions or struct definitions. They cannot desugar into (e.g.) expressions.

deconstruction Sugars are given syntax as parameters. Can they deconstruct this syntax (e.g., by case analysis) or must they treat it parametrically (i.e., only compose it)?

There is a strong interaction between the ability to deconstruct parameters and desugaring order (OI vs. IO). With OI order, the parameters that are deconstructed are in the surface language, since they haven’t been desugared yet. Under IO order, however, the parameters are in the core language, because they have been desugared.

sugar-defining sugars Can sugars define sugars? That is, can a sugar expand into the definition of another sugar?

Of course, four measures isn’t enough to fully capture how expressive a desugaring system is! However, it should give a rough idea of the situations in which it is appropriate to use.

2.1.7 Safety

Last—but certainly not least—we will consider four different safety properties of desugaring systems:

syntactic safety Can a sugar expand to syntactically invalid code? For example, in the C Preprocessor (section 2.2.5), this (textual) macro:

```
#define discriminant(a,b,c) ((b) * (b) - (4 * (a) * (c))
```

is a completely valid macro that the preprocessor will happily expand for you, leading to syntax errors at its use site because its parentheses aren’t balanced.

Thus we will call the C Preprocessor syntactically unsafe. In contrast, a syntactically safe desugaring system would raise an error on such a sugar definition, even if that sugar was never used. This is roughly analogous to

Safety and expressiveness are counterpoints. Safety comes from the ability of the compiler to tell whether you’re doing something wrong, and the less expressive the language, the easier it is to tell.
type checking: a type checker will warn that a function definition contains a type error even if that function is never used.

Lisps (i.e., languages with parenthetical syntax) have an interesting relationship to syntactic safety because they generally have two “layers” of syntax. The lower layer is simply s-expressions, and ensures that parentheses are well-balanced (and that there are no tokenization errors). The higher layer checks that the s-expression makes syntactic sense. For example, in Scheme, `(define x 1)` is invalid at both layers, `(define ((x)) 1)` is valid at the first layer but not at the second, and `(define x 1)` is valid at both layers. This can be called bicameral syntax [49, page 13], by analogy to legislative houses that are split into a lower and upper level.

As a result, it is possible for a Lisp’s macro system to respect the lower layer of syntax, but not the higher layer. And in fact all of the Lisps we assessed fall into this category. We write this in the table as “partly” syntax safe.

**Hygiene** Unlike the other safety properties discussed here, hygiene [45] is not about the error behavior of the desugaring system. Instead, it is about how desugaring treats variables. As an example, consider this simple or sugar (using Racket syntax):

```
(define-syntax-rule
  (or a b)
  (let ((temp a)) (if temp temp b))
```

If a desugaring system did nothing special with variables, then this code:

```
(let ((temp "70 degrees")
  (or false temp))
```

would desugar into this code:

```
(let ((temp "70 degrees")
  (let ((temp false)) (if temp temp temp))
```

which would then evaluate to `false`, which is wrong. The issues is that the user-written variable called `temp` is captured by the sugar-introduced variable called `temp`. Thus this naive desugaring is *unhygienic*.

At its core, hygiene is lexical scoping for sugars. You should be able to tell where a user-written variable is bound by looking at its code (in the example, see that the `temp` in `or false temp` should be bound by the surrounding `let`). Likewise, you should be able to tell where a sugar-introduced variable is bound by looking at the sugar definition (and there are analogous examples where a sugar-introduced variable gets captured by a user-written variable).
While this account of hygiene is sufficient for this chapter’s purposes, there is plenty more to say about hygiene. For an overview of the various kinds of possible hygiene violations, see Adams [2]. For an alternative take on the hygiene property, see section 5.5.4 of this thesis.

**Scope Safety** Can a sugar introduce scope errors? We will say that a desugaring system is *scope safe* if sugars cannot introduce an unbound identifier or cause a user-defined variable to become unbound.

**Type Safety** Can a sugar introduce type errors? We will say that a desugaring system is *type safe* if sugars cannot introduce type errors. For example, this MetaOCaml program (the `< ... >` syntax quotes an expression):

```ml
let sugar() = `<3 + "four">.`;;
```

does not compile because of the type error present in the sugar—even though the sugar is never used.

## 2.2 Applying the Taxonomy to Desugaring Systems

In this section, we apply the taxonomy to a number of desugaring systems. Table 1 shows the results, and the rest of this section briefly describes each desugaring system in turn. The entries of this table come primarily from test cases, which are available at [https://github.com/justinpombrio/thesis](https://github.com/justinpombrio/thesis) in the sugars folder.

### 2.2.1 Early Lisp Macros

Early versions of Lisp [62, 49, 56, 63] introduced over time a variety of macro-like features, including: the `eval` operator (which takes an s-expression as an argument at runtime, and interprets it as code); fexprs (functions whose arguments are passed to them without first being evaluated); quasiquotations [4]; and macros. These macros were arbitrary Lisp functions from s-expression to s-expression that run at compile-time. They were unhygienic; indeed, even the notion of hygiene had yet to be worked out.

### 2.2.2 Scheme Macros

**Version: R5RS, as implemented in Racket v6.10.1**

The Scheme language [42] was the first to introduce pattern-based desugaring rules [46], and hygienic macro expansion [45]. It set the stage for desugaring systems to come, including this thesis. The fact that we have so little to say
Table 1: Taxonomization of Desugaring Systems. (Categories with asterisks are described further in that desugaring system’s subsection.)

<table>
<thead>
<tr>
<th>System</th>
<th>Metalanguage</th>
<th>Authorship</th>
<th>Deconstruction</th>
<th>Representation</th>
<th>Sugar-defining Sugars</th>
<th>Syntax Safe</th>
<th>Hygienic</th>
<th>Type Safe</th>
<th>Scope Safe</th>
<th>Parameters</th>
<th>Time of Expansion</th>
<th>Desugaring Order</th>
<th>Metalanguage</th>
<th>Authorship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Early Lisp Macros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheme</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Racket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>McMicMac Micros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Lisps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C Preprocessor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ Templates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Template Haskell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haskell Sugars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rust Macros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Julia Macros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MetaOCaml</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resugaring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eval Resugaring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scope Resugaring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Resugaring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

about Scheme’s macros is testament to how standard its view of macros has become.

2.2.3 Racket Macros

VERSION: RACKET v6.10.1

Racket has an extremely powerful and heavily used macro system. It is part and parcel of the language: Racket is meant to be a language for defining languages, and does so with its macro system. Dozens of languages have been defined this way, including:

- The Racket language itself, most of which is defined by macros (including most of its mechanisms for defining macros).
- The Typed Racket language, whose type system is defined entirely by macros.
- A number of non-programming languages, such as the Scribble document templating system.

Given the way Racket’s macros are used, they are best viewed as compiler extensions more than simply syntactic sugar.

For an overview of the aspirations of Racket and its macro system, see the Racket Manifesto.

2.2.4 McMicMac Micros

Krishnamurthi et al. describe a desugaring system called McMicMac that is based on a combination of macros and micros, and built on top of Scheme. While macros perform an AST-to-AST transformation, micros transform the AST to an IR (an “intermediate representation” used by the compiler). Relevantly, the IR is treated as data and cannot contain macros or micros, and is thus not recursively desugared.

Micros extend macros in a couple of ways:

- Micros can take as arguments and produce as results values that aren’t just syntax. For example, a set of micros can keep track of the lexical environment, and use it to compute the set of free variables of a term.
- Micros can be grouped into vocabularies, with each vocabulary representing a different kind of transformation, and these vocabularies can be composed.

Overall, McMicMac promotes micros as a useful generalization of macros.
2.2.5 \textit{C Preprocessor}

\textbf{Version: gcc v6.3.0}

The C Preprocessor (hereafter \texttt{cpp}) \cite{cpp} is a \textit{text preprocessor}: a source-to-source transformation that operates at the level of text. (More precisely, it operates on a token stream, in which the tokens are approximately those of the C language). It is usually run before compilation for C or C++ programs, but it is not very language specific, and can be used for other purposes as well. Unlike the rest of the desugaring systems described in this chapter, \texttt{cpp} is not Turing complete. It avoids Turing completeness by a simple mechanism: if a macro invokes itself (directly or indirectly), the recursive invocation will not be expanded.

A number of issues arise from the fact that \texttt{cpp} operates on tokens, and is thus unaware of the higher-level syntax of C \cite{cpp}. As an example, consider this innocent looking \texttt{cpp} desugaring rule that defines an alias for subtraction:

\begin{verbatim}
#define SUB(a, b) a - b
\end{verbatim}

This rule is completely broken. Suppose it is used as follows:

\begin{verbatim}
SUB(0, 2 - 1))
\end{verbatim}

This will expand to $0 - 2 - 1$ and evaluate to $-3$. We can revise the rule to fix this:

\begin{verbatim}
#define SUB(a, b) (a) - (b)
\end{verbatim}

This will fix the last example, but it is still broken. Consider:

\begin{verbatim}
SUB(5, 3) * 2
\end{verbatim}

This will expand to $5 - 3 * 2$ and evaluate to $-1$. The rule can be fully fixed by another set of parentheses:

\begin{verbatim}
#define SUB(a, b) ((a) - (b))
\end{verbatim}

In general, both the inside boundary of a rule (the parameters \texttt{a} and \texttt{b}), and the outside boundary (the whole \texttt{rhs}) need to be protected to ensure that the expansion is parsed correctly. If the sugar is used in expression position, as in the \texttt{SUB} example, this can be done with parentheses. In other positions, different tricks must be used: e.g., a rule meant to be used in statement position can be wrapped in \texttt{do {...} while(0)}. This is a poor state of affairs: software developers should not need to learn and remember to use tricks like this simply to write working macros.

There are other issues that arise with text-based transformations as well, such as variable capture. Furthermore, all of these issues are inherent to text-based transformations, and cannot be fixed from within the paradigm. \textit{Overall, code transformations should never operate at the level of text (or token streams)}. 
2.2 Applying the Taxonomy to Desugaring Systems

2.2.6 C++ Templates

Version: g++ v4.8.4

C++ templates [39] are not general-purpose sugars, because they cannot take code as a parameter. Thus you cannot, for example, express a sugar that takes an expression e and expands it to e + 1.

Instead, C++ templates are used primarily to instantiate polymorphic code by replacing type parameters with concrete types. For example, take the following template declarations from [39, page 344]:

```cpp
template<class T>
constexpr T pi = T(3.1415926535897932385L);

template<class T>
T circular_area(T r) {
    return pi<T> * r * r;
}
```

The first template declares a constant pi, which can be instantiated with different (presumably numeric) types T. The second template declares a function circular_area, which can also be parameterized over different types T, and uses T both for its radius argument and for its pi constant. We’ll use this function template as a running example.

Besides function definitions, several other kinds of declarations can be templated, including methods, classes, structs, and type aliases. The behavior of each is similar. A template may be invoked by passing parameters in angle brackets. An invoked template acts like the kind of thing the template declared, and can be used in the same positions. Thus, e.g., a struct template should be invoked in type position; and our running function template example should be invoked in expression position to make a function, which can then be called:

```cpp
float area = circular_area<float>(1);
```

When a template is invoked like this, a copy of the template definition is made, with the template parameters replaced with the concrete parameters. In our example, this produces the code:

```cpp
float circular_area(float r) {
    return pi<float> * r * r;
}
```

So far we have only described type parameters, but templates can also take other kinds of parameters, including primitive values (such as numbers) and

*If a template is invoked multiple times with the same parameters, only one copy of the code will be made, however.*
other templates. The ability to manipulate numbers and invoke other templates at compile time make C++ templates powerful and, unsurprisingly, Turing complete. However, templates cannot be parameterized over code, and thus are not general-purpose sugars. For example, most of the examples in this thesis cannot be written as C++ templates.

Template expansion uses IO desugaring order. This is important because it is possible to define both a generic template that applies most of the time, and a specialized template that applies if a parameter has a particular value. For example, this could be used to make a HashMap use a different implementation if its keys are ints. Thus it is important that a template see the concrete type (e.g. int) that is passed to it, even if this type is the result of another template expansion.

2.2.7 Template Haskell

**Version:** GHC v7.6.3

Haskell allows user-defined sugars by way of the Template Haskell system [75]. It allows terms to be written either in concrete surface syntax:

```
  [ 1 + 2 ]
```

or in abstract surface syntax:

```haskell
  return $ AppE (AppE (VarE '(+))
                  (LitE (IntegerL 1)))
            (LitE (IntegerL 2))
```

It maintains hygiene by requiring terms to be wrapped in a Monad (called Q) that ensures that introduced variables are given fresh names (unless explicitly marked otherwise). However, this can easily be broken via quotation brackets ([| ... |]), so we count Template Haskell as unhygienic.

2.2.8 Haskell Sugars

Besides its template system, Haskell also has some sugars built in to the language. For example, the Haskell specification states that list comprehensions [61 section 3.11] are given by the following transformation:

```
  [e | True] = [e]
  [e | q] = [e | q, True]
  [e | b, Q] = if b then [e | Q] else []
  [e | p <- 1, Q] = let ok p = [e | Q]
```
We will use this sugar as a case study for both resugaring scope rules (section 5.6.2) and resugaring type rules (section 6.6.2).

### 2.2.9 Rust Macros

**Version: 1.24.0-nightly**

Rust has a hygienic macro system with pattern-based rules similar to Scheme’s syntax-rules. Rust also has a second “procedural” macro system. This secondary macro system allows macros to be written as arbitrary Rust functions. However, it can only be used in very specific circumstances—at the time of this writing, only to add custom derive traits—so we do not discuss it further.

### 2.2.10 Julia Macros

**Version: 0.4.5**

Julia macros are written as Julia functions, and can construct and deconstruct arbitrary syntax. They are mostly unhygienic. (The user guide discusses hygiene, but in testing the macro system appears to be unhygienic outside of the specific situations described.) Beyond this, Julia also allows arbitrary run-time metaprogramming: constructing, deconstructing, and evaluating code at run-time. Table 1 describes only Julia’s macro system.

### 2.2.11 MetaOCaml

MetaOCaml is a multi-stage programming extension to OCaml. This means that it provides three new constructs to the language:

- **brakcets** (written `.< ... >.`) Quote an expression, producing a syntax value at run-time.
- **escape** (written `.~`) Splice values into a quotation.
- **run** (written `.!`, or `run`) Given a syntax value at run-time, evaluate it.

Thus this code:

```ocaml
open Runcode;;
open Printf;;
let x = .<printf("k")>. in !. .<(.~x; .~x)>.;;
```
prints `kk`. This method of programming is called \textit{multi-stage} because these quotations can be arbitrarily nested. An expression under \textit{n} quotations is said to run in stage \textit{n}.

The \textit{upside} to multi-stage programming is that it can provide very strong safety guarantees. MetaOCaml guarantees that staged functions (i.e., functions that produce quoted code) cannot produce ill-formed, ill-scoped, or ill-typed code. This is checked at compile-time, before the staged function is even invoked. This is vastly safer than the other desugaring systems we surveyed.

The \textit{downside} to multi-stage programming is that it is not a full-fledged desugaring system. It cannot introduce new syntax, it cannot deconstruct syntax, and its “desugaring” happens explicitly at run-time via calls to \texttt{run}. For all of these reasons, multi-staged programming is best viewed as a way of more efficiently organizing computation rather than as a means of extending a language.

2.2.12 Resugaring Systems

Table 1 lists our three resugaring systems. This is referring to the \textit{kinds of desugaring systems} that these resugaring approaches support. There are two interesting aspects worth discussing.

First, while the resugaring systems appear to do well on the expressiveness front, there is a limitation hidden in the metalanguage being “rules”. While many of the other desugaring systems provide an “escape hatch” out of their “rules” allowing arbitrary code to run during the desugaring process, our resugaring work strictly requires pattern-based rules.

Second, the safety guarantees are provided by a mix of the \textit{desugaring} and \textit{resugaring} systems. Each of the resugaring systems is marked as “syntax safe” and “hygienic”, because those are assumptions it makes about the desugaring system it is built on top of. Scope resugaring is additionally marked as “scope safe”, because scope resugaring would reject any sugar that was not scope safe. Likewise, type resugaring would reject any sugar that was not type safe (and as a result any sugar that was not scope safe).
NOTATION

In this chapter, we define notation and set assumptions that will be used throughout the rest of the thesis.

Most importantly, we assume that desugaring is given (externally to the language) as a set of pattern-based rewrite rules (à la Scheme’s `syntax-rules`). This is important because it will form an expressive limit on our techniques for resugaring evaluation sequences (chapter 4), scope rules (chapter 5), and type systems (chapter 6).

3.1 asts

For the purposes of resugaring scope (chapter 5), we require asts (abstract syntax trees) to explicitly distinguish between variable declarations \( x^D \) (i.e., binding sites), and variable references \( x^R \) (i.e., uses). We also require variables to have an ast position \( i \) that uniquely distinguishes them. These distinctions will be important in chapter 5, but can otherwise be ignored.

We will refer to asts (and parts of asts) as terms, and write them as \( e \). Terms can be inductively defined as:

\[
\begin{align*}
\text{constructor } C & \ ::= \ name \quad \text{syntactic construct name} \\
\text{term } e & \ ::= \ value \quad \text{primitive value} \\
& \quad | \ (C \ e_1 ... e_n) \quad \text{ast node} \\
& \quad | \ x^R_i \quad \text{variable reference at position } i \\
& \quad | \ x^D_i \quad \text{variable declaration at position } i
\end{align*}
\]

By “constructor”, we mean the name of a syntactic construct, such as `Plus` or `If`. Thus \( (\text{Plus} \ x^R_i \ 1) \) is a term.
3.2 Desugaring

We require that desugaring be based on patterns: it proceeds by matching a LHS (left-hand-side) pattern against a term, and then substituting into the RHS (right-hand-side) pattern. We will describe how this works, piece by piece.

First, patterns $p$ are terms that can contain pattern variables $\alpha$:

$$ p ::= \alpha \quad \text{pattern variable}$$

$$ | \text{value} \quad \text{primitive value}$$

$$ | (C \, p_1 \ldots p_n) \quad \text{AST node}$$

$$ | x^p_i \quad \text{variable reference at position } i$$

$$ | x^d_i \quad \text{variable declaration at position } i$$

A term can then be matched against a pattern—written $e/p$—to produce an environment $\gamma$. An environment is a mapping from pattern variables to terms:

$$ \gamma ::= \{ \alpha \mapsto e, \ldots \} $$

Once an environment is obtained, it can be substituted into a pattern—written $\gamma \circ p$—to produce a term.

We can now say how to expand a single desugaring rule. Each rule has the form:

$$ \text{desugaring rule } r ::= p \Rightarrow p' $$

where $p$ is the rule’s LHS, and $p'$ is the rule’s RHS. To expand a rule $p \Rightarrow p'$ in a term $e$, match against the LHS and substitute into the RHS: $(e/p) \circ p'$.

Finally, desugaring is the recursive expansion of all of the sugars in a term. We will typically assume that desugaring happens in order (see section 2.1.4), although chapter 5 will be agnostic to desugaring order.

An example Putting this all together, suppose we have the following sugar, which encodes Let using Lambda:

$$ (\text{Let } \alpha \beta \delta) \Rightarrow (\text{Apply } (\text{Lambda } \alpha \delta) \beta) \quad \text{“Let } \alpha \text{ equal } \beta \text{ in } \delta\” $$

and we want to expand the term $(\text{Let } x^p_1 x^k)$. To do so, we match against the LHS:

$$ \gamma = (\text{Let } x^p_1 x^k)/(\text{Let } \alpha \beta \delta) = \{ \alpha \mapsto x^p, \beta \mapsto 1, \delta \mapsto x^k \} $$

and substitute into the RHS:

$$ \{ \alpha \mapsto x^p, \beta \mapsto 1, \delta \mapsto x^k \} \circ (\text{Apply } (\text{Lambda } \alpha \delta) \beta) = (\text{Apply } (\text{Lambda } x^p x^k) \, 1) $$

Since there are no more sugars in the term, we are done desugaring.
3.2.1 Restrictions on Desugaring

Not every syntactically valid desugaring rule is semantically sensible, and not every semantically sensible desugaring rule is feasible to resugar. Here we give (i) a set of well-formedness criteria on desugaring rules, without which they don’t make semantic sense, and (ii) a set of restrictions on desugaring rules that we need to effectively resugar them. These restrictions are sufficient for all of the resugaring methods presented in this thesis.

Well-formedness criteria for desugaring rules

For every desugaring rule:

1. Each pattern variable in the rhs also appears in the lhs. Otherwise the pattern variable would be unbound during expansion.

2. The lhs pattern contains no references or declarations. Rather, these should be bound to its pattern variables during expansion. For example, the Let sugar above could not have been written as:

   \((\text{Let } x^0 \beta \delta) \Rightarrow (\text{Apply (Lambda } x^0 \delta) \beta)\) “Let \(x^0\) equal \(\beta\) in \(\delta\)”

3. Each desugaring rule’s lhs must have the form \((C \ p_1, ..., \ p_n)\). I.e., it cannot be just a variable or primitive. We will need this fact in chapter 4.

Restrictions on desugaring rules

For every desugaring rule:

1. Each pattern variable appears at most once in the lhs and at most once in the rhs. Allowing duplicate pattern variables complicates matching, unification, and proofs of correctness. It also copies code and, in the worst case, can exponentially blow up programs. We therefore disallow duplication, with some exceptions for pattern variables bound to atomic terms.

2. The rules’ lhs must be disjoint. If they are not, it presents issues both for resugaring evaluation sequences (section 4.5.1), and resugaring type systems (section 6.3.1).

   Some of the resugaring chapters relax these assumptions. For instance, the evaluation resugaring chapter (chapter 4) formally supports ellipses (which allow matching zero or more repetitions of a pattern), and the scope resugaring chapter is agnostic to the order of desugaring (which is assumed to be \(\text{oi}\) here). However, all chapters handle at least these kinds of sugars.
A set of pattern-based desugaring rules can be viewed as a term rewriting system \[43\]. Two of the main questions studied about term rewriting systems is how to tell whether they are **confluent**, and whether they **terminate**. Confluence asks: if rewrites are applied in an arbitrary order, are they guaranteed to eventually produce the same result? The main mechanism for studying this is **critical pairs** \[16\], which capture the situations in which rewrite rules overlap. In practice, however, desugaring systems tend to fix a desugaring order (e.g. oi), making the question of confluence moot.

Likewise, there is literature on determining whether a set of rewrite rules always terminates (which is undecidable in general) \[37\]. This could be used to check whether desugaring is guaranteed to halt (with potential false positives). Termination checking is not necessary for **resugaring**, however: term resugaring, scope rule resugaring, and type rule resugaring all halt, even if desugaring itself may not.
Part III

RESUGARING
RESUGARING EVALUATION SEQUENCES

In this chapter, we tackle the challenge of combining syntactic sugar with semantics. Given a set of desugaring rules written in the style of the last chapter, we show how to resugar program execution: to automatically convert an evaluation sequence in the core language into a representative evaluation sequence in the surface syntax. Each step in the surface language emulates one or more steps in the core language. The computed steps hide the desugaring, thus maintaining the abstraction provided by the surface language.

The chief challenge is to remain faithful to the original semantics—we can’t change the meaning of a program!—and to ensure that the internals of the code introduced by the syntactic sugar does not leak into the output. Our chief mechanisms for achieving this are to (a) perform static checks on the desugaring rules to ensure they fall into the subset we can handle, and (b) rewrite the reduction relation with instrumentation to track the origin of terms. We implement these ideas in a tool called Confection, and formally verify key properties of our approach, given simplifying assumptions, in the Coq proof assistant.

This chapter comes from work published in PLDI 2014 and ICFP 2015 under the titles of Resugaring: Lifting Evaluation Sequences through Syntactic Sugar and Hygienic Resugaring of Compositional Desugaring (both co-authored with Shriram Krishnamurthi) [67] [68].

4.1 OUR APPROACH

We aim to compute sensible evaluation sequences in a surface language, while remaining faithful to the core language’s semantics. One approach would be to attempt to construct a lifted (to the surface language) reduction-relation directly. It is unclear, however, how to do this without making deep assumptions about the core language evaluator (for instance, assuming that it is defined as a term-rewriting system that can be composed with desugaring).

Our approach instead makes minimal assumptions about the evaluator, treating it as a black-box (since it is often a complex program that we may not be able
to modify. We assume only that we have access to a stepper that provides a sequence of evaluation steps (augmented with some meta information) in the core language. In section 4.6, we show how to obtain such a stepper from a generic, black-box evaluator with a strategy that can be implemented by pre-processing the program before evaluation.

Our high-level approach is to follow the evaluation steps in the core language, find surface-level representations of some of the core terms, and emit them. Not every core-level term will have a surface-level representation; these steps will be skipped in the output. The evaluation sequence shown, then, is the sequence of surface-level representations of the core terms that were not skipped. Central to this approach are three properties:

**Emulation** Each term in the generated surface evaluation sequence desugars into the core term which it is meant to represent (up to term isomorphism).

**Abstraction** Code introduced by desugaring is never revealed in the surface evaluation sequence, and code originating from the original input program is never hidden by resugaring.

**Coverage** Resugaring is attempted on every core step, and as few core steps are skipped as possible.

4.2 Informal Solution Overview

We first present the techniques used by our solution, and some subtleties, informally. We choose a familiar desugaring example: the rewriting of $\text{or}$, as used in languages like Lisp and Scheme. We assume the surface language has $\text{or}$ as a construct, while the core does not. We present our examples using a traditional infix concrete syntax.

4.2.1 Finding Surface Representations Preserving Emulation

We start by defining a simple, binary version of $\text{or}$ (that let-binds its first argument in case it has side-effects):

$$\text{or}(x, y) \rightarrow \text{Let}([\text{Binding}(\text{"t"}, x)],
\text{If}(\text{Id}(\text{"t"}), \text{Id}(\text{"t"}), y)));$$

In this section we focus on abstract syntax and ignore the mapping to it from concrete syntax.

Consider the surface term $\text{not(true) OR not(false)}$, and suppose it is left-associative. After desugaring, this would evaluate as follows in the core language (assuming a typical call-by-value evaluator):
(\text{let } t = \text{not}(\text{true}) \text{ in }
  \text{if } t \text{ then } t \text{ else } \text{not}(\text{false})
\rightarrow \text{let } t = \text{false} \text{ in }
  \text{if } t \text{ then } t \text{ else } \text{not}(\text{false})
\rightarrow \text{if } \text{false} \text{ then } \text{false} \text{ else } \text{not}(\text{false})
\rightarrow \text{not}(\text{false})
\rightarrow \text{true}
)

In the surface language, we would wish to see this as (using dashed arrows to denote reconstructed steps):

\( \text{not}(\text{true}) \text{ OR } \text{not}(\text{false}) \)
\[ \rightarrow \text{false} \text{ OR } \text{not}(\text{false}) \]
\[ \rightarrow \text{not}(\text{false}) \]
\[ \rightarrow \text{true} \]

The first two terms in the core evaluation sequence are precisely the expansions of the first two steps in the (hypothetical) surface evaluation sequence. This suggests we can \textit{unexpand} core terms into surface terms by running rules “in reverse”: matching against the \textit{rhs} and substituting into the corresponding \textit{lhs}. (To preserve Emulation, unexpansion must be an inverse of expansion: we prove that this is so in section \hyperref[sec:hygiene]{4.5.2}) We will now show how the last two steps may come about.

\subsection{Maintaining Abstraction}

When unexpanding, we should only unexpand code that originated from a sugar. If the surface program itself is

\( \text{let } t = \text{not}(\text{true}) \text{ in }
  \text{if } t \text{ then } t \text{ else } \text{not}(\text{false}) \)

it should not unexpand into \( \text{not}(\text{true}) \text{ OR } \text{not}(\text{false}) \): this would be confusing and break the second clause of the Abstraction property.

We therefore augment each subterm in a core term with a list of \textit{tags}, which indicate whether a term originated in the original source or from sugar. Unexpansion attempts to process terms marked as originating from a desugaring rule; this unexpansion will fail if the tagged term no longer has the form of the rule’s \textit{rhs}. When this happens, we conclude that there is no surface representation of the core term and skip the step.

\begin{flushright}
Whereas the tags used in hygienic macro expansion \cite{compcourses} specify time steps, the tags in resugaring specify which sugar the code originated from.
\end{flushright}
To illustrate this, we revisit the same core evaluation sequence as before.

“\{\texttt{Head\_Or}\: \}” is a tag on the desugared expression that indicates it originated from the \texttt{Or} sugar, and “\{\texttt{Body}\: \}” is a tag that indicates it originated from sugar (without saying which, although in this case it came from \texttt{Or}):

\[
\{\texttt{Head\_Or}\: \text{let } t = \texttt{not(true)} \text{ in} \\
\text{\{\texttt{Body}\: \text{if } t \text{ then } t \text{ else } \texttt{not(false)}\}}\}
\rightarrow \{\texttt{Head\_Or}\: \text{let } t = \texttt{false} \text{ in} \\
\text{\{\texttt{Body}\: \text{if } t \text{ then } t \text{ else } \texttt{not(false)}\}}\}
\rightarrow \text{\{\texttt{Body}\: \text{if } \texttt{false} \text{ then } \texttt{false} \text{ else } \texttt{not(false)}\}}
\rightarrow \text{\texttt{not(false)}}
\rightarrow \text{\texttt{true}}
\]

The tags on the first two steps suggest that \texttt{Or}’s desugaring rule be applied in reverse. They can be unexpanded because they match the \texttt{rhs} of \texttt{Or}. The third step fails to resugar because the \texttt{if} is never unexpanded, and thus is a leftover fragment of sugar. This step is therefore skipped, yielding no surface step. The last two steps are not tagged and are therefore included in the surface evaluation sequence as-is.

4.2.3 Striving for Coverage

Emulation and Abstraction guarantee an accurate surface evaluation sequence, but they do not guarantee a useful one. For instance, the following evaluation sequence is perfectly consistent with these two properties:

\[
\text{\texttt{not(true)}} \text{ OR } \text{\texttt{not(false)}}
\rightarrow \text{\texttt{true}}
\]

However, a stepper that only shows the final step is unhelpful. We therefore propose a third property, Coverage, which states that steps are not “unnecessarily” skipped. While Emulation and Abstraction are formally proved in section 4.5.3 we have not found a complete formalization of Coverage, so we can only strive to attain it in our systems and evaluate it in practice. Our examples (section 4.3 and section 4.7) show that we do indeed obtain detailed and useful surface evaluation sequences.
4.2.4  Trading Abstraction for Coverage

Suppose the surface term \( A \lor B \lor C \) parses to \( \text{Or}(A, B, C) \). We therefore want to extend \( \text{Or} \) to handle more than two sub-terms. We can do this by adding another rule:

\[
\text{Or}(\{x, y\}) \rightarrow \\
\text{Let}(\{\text{Binding}("t", x)\}, \\
\text{If}(\text{Id}("t"), \text{Id}("t"), y)));
\]

\[
\text{Or}(\{x, y, ys \ldots\} \rightarrow \\
\text{Let}(\{\text{Binding}("t", x)\}, \\
\text{If}(\text{Id}("t"), \text{Id}("t"), \text{Or}(\{y, ys \ldots\})))
\]

We assume a prioritized semantics in which rules are tried in order; the first rule whose LHS matches the invocation is used. The ellipses denote zero or more repetitions of the preceding pattern [46].

Consider the surface term \( (\text{false} \lor \text{false} \lor \text{true}) \). Given the revised definition of \( \text{Or} \), unexpansion would yield the following lifted evaluation steps:

\[
\text{false} \lor \text{false} \lor \text{true} \\
\rightarrow \text{false} \lor \text{true} \\
\rightarrow \text{true}
\]

In particular, it correctly suppresses any presentation of the recursive invocation of \( \text{Or} \) introduced by desugaring—precisely what Abstraction demands! However, there are settings (such as debugging or education) where the user might wish to see this invocation, i.e., to obtain the surface evaluation sequence:

\[
\text{false} \lor \text{false} \lor \text{true} \\
\rightarrow \text{false} \lor \text{true} \\
\rightarrow \text{true}
\]

Thus, we let sugar authors make part of a rule’s RHS transparent by prefixing it with \( ! \). Here, writing the second \( \text{Or} \) rule as:

\[
\text{Let}(\{\text{Binding}("t", x)\}, \\
\text{If}(\text{Id}("t"), \text{Id}("t"), \text{Or}(\{y, ys \ldots\})))
\]

yields the latter surface evaluation sequence.

This illustrates that there is a trade-off (which we make precise with theorem 4) between Abstraction and Coverage. Because the trade-off depends on goals, we entrust it to the sugar author. In the limit, marking the entirety of each rule as transparent results in an ordinary trace in the core, ignoring all sugar.
4.2.5 Maintaining Hygiene

The implementation discussed in this chapter—Confection—represents variables as strings and is not hygienic (e.g., sugar-introduced variables could accidentally capture user-defined variables). However, in section 4.8, we provide a sketch of how it could be made hygienic.

4.3 Confection at Work

We demonstrate how the techniques just described come together to show surface evaluation sequences in the presence of sugar. Consider the following program, written in the language Pyret [pyret.org], that computes the length of a list:

```pyret
fun len(x):
    cases(List) x:
        | empty() => 0
        | link(_, tail) => len(tail) + 1
    end
end
len([1, 2])
```

This seemingly innocuous program contains a lot of sugar. The `cases` expression desugars into an application of the matchee's `_match` method on an object containing code for each branch; the function declaration desugars into a let binding to a lambda; addition desugars into an application of a `_plus` method; and the list `[1, 2]` desugars into a chain of list constructors. Here is the full desugaring (i.e., the code that will actually be run):

```pyret
len = fun(x):
    temp17 :: List = x
    temp17._match_(
        {"empty" : fun(): 0 end,
         "link" : fun(_, tail):
            len(tail)._plus_(1 end),
         fun(): raise("cases: no cases matched"));
    end
    len(list._link_(1, list._link_(2, list._empty_)))
```

This degree and nature of expansion is not unique to Pyret. It is also found in languages like Scheme, due to the small size of the core, and in semantics like λJS [32], due to both the size of the core and the enormous complexity of the surface language.
Nevertheless, here is the surface evaluation (pretty-)printed by Confection (where \texttt{<func>} denotes a resolved functional):

\[
\begin{align*}
\rightarrow \texttt{<func>}(\{1, 2\}) \\
\rightarrow \texttt{cases(List)} [1, 2]: \\
    | \texttt{empty()} => 0 \\
    | \texttt{link(_, tail)} => \texttt{len(tail)} + 1 \\
\end{align*}
\]

\[
\begin{align*}
\rightarrow \texttt{<func>}(\{2\}) + 1 \\
\rightarrow \texttt{cases(List)} [2]: \\
    | \texttt{empty()} => 0 \\
    | \texttt{link(_, tail)} => \texttt{len(tail)} + 1 \\
\end{align*}
\]

\[
\begin{align*}
\rightarrow \texttt{<func>}(\{\}) + 1 + 1 \\
\rightarrow \texttt{cases(List)} [\]: \\
    | \texttt{empty()} => 0 \\
    | \texttt{link(_, tail)} => \texttt{len(tail)} + 1 \\
\end{align*}
\]

\[
\begin{align*}
\rightarrow 0 + 1 + 1 \\
\rightarrow 1 + 1 \\
\rightarrow 2
\end{align*}
\]

This sequence hides all the complexity of the core language.

4.4 The Transformation System

We will present our system in three parts. First (section 4.4.1), we will describe how our transformation system works, up to the level of performing a single transformation: either expanding or unexpanding a single (instance of a) sugar. Next (section 4.4.2), we will describe how to use tags to fully desugar and re-sugar terms, transforming not just a term but its subterms as well. Finally (section 4.4.3), we will show how to use the transformation system to lift core evaluation sequences to surface evaluation sequences.

4.4.1 Performing a Single Transformation

We begin by describing the form and application of our transformation rules.
Patterns

Since rules are applied both forward and in reverse, we represent their LHSs and RHSs uniformly as patterns. We give a broader definition of patterns here than in section 3.1; here we allow patterns to contain lists, ellipses, and (for the purposes of resugaring) tags:

\[
\begin{align*}
\text{pattern } p & \ ::= \ a \quad \text{pattern variable} \\
& \mid \text{value} \quad \text{primitive value} \\
& \mid (C \ p_1 \ldots p_n) \quad \text{AST node} \\
& \mid [p_1 \ldots p_n] \quad \text{list of length } n \\
& \mid [p_1 \ldots p_n \ p^\ast] \quad \text{list of length } \geq n \ (\text{ellipses}) \\
& \mid x \quad \text{variable} \\
& \mid (\text{Tag } o \ p) \quad \text{origin tag}
\end{align*}
\]

\[
\begin{align*}
\text{term } e & \ ::= \ p \quad \text{pattern w/o pattern variables or ellipses} \\
\text{origin tag } o & \ ::= \ (\text{Head } i \ e) \quad \text{marks topmost rule production} \\
& \mid (\text{Body } \text{bool}) \quad \text{marks each rule production}
\end{align*}
\]

Variables are denoted by a lowercase identifier (we omit the reference/declaration distinction, and the position \(i\), because they are not relevant in this chapter); compound nodes are written in s-expression form; and lists are denoted by a bracketed list of subpatterns. Nodes must have fixed arity, so lists are used when a node needs to contain an arbitrary number of subterms. Ellipses (which we write formally as \(p^\ast\) to distinguish them from metasyntactic ellipses) in a list pattern denote zero or more repetitions of the pattern they follow.

A term \(e\) is simply a pattern without pattern variables or ellipses. Tags and origins \((o)\) are described in section 4.4.2.

Our definition of patterns determines both the expressiveness of the resulting transformation system and the ability to formally reason about it. There is a natural trade-off between the two. We pick a definition similar to that of Scheme syntax-rules-style macros, though without guard expressions.

Formally, these patterns are regular tree expressions [3]. Regular tree expressions \(\text{trx}\) are the natural extension of regular expressions to handle trees: they add a primitive \((C \ \text{trx}_1 \ldots \text{trx}_n)\) for matching a tree node labeled \(C\) with branches matching the regular tree expressions \(\text{trx}_1 \ldots \text{trx}_n\). Whereas regular tree expressions conventionally allow choice, we encode it using multiple rules, making the pattern language simpler.

While we have found this definition of patterns suitably powerful for a wide variety of sugars—including all those discussed in this chapter—our approach
Figure 1: Bindings

is not dependent on the exact definition. The precise requirements for the transformation language are given in section 4.5.3

Matching, Substitution, and Unification

Our transformations are implemented with simpler operations on patterns: matching and substitution.

Matching a term against a pattern induces an environment that binds the pattern’s pattern variables. This environment may be substituted into a pattern to produce another term. Formally, an environment is a mapping from pattern variables α to bindings b, where each binding is either a term e, a list binding \([b_1 ... b_n]\), or an ellipsis binding \([b_1 ... b_n b^*]\). A pattern variable within ellipses is bound to a list binding \([b_1 ... b_n]\) instead of a list term \([b_1 ... b_n]\) because they behave slightly differently under substitution. Ellipsis bindings are similar, but needed only during unification when a pattern variable within an ellipsis is itself bound to an ellipsis pattern.

We will write \(e/p\) to denote matching a term \(e\) against a pattern \(p\), and write \(\gamma \bullet p\) to denote substituting the bindings of an environment \(\gamma\) into a pattern \(p\). We will write \(e \geq p\) to mean that \(e/p\) is defined, and \(\gamma_1 \cup \gamma_2\) for the right-biased union of \(\gamma_1\) and \(\gamma_2\). The matching and substitution algorithms are given in fig. 2 while bindings are defined in fig. 1.

For an example of matching and substitution, consider one of the rules of our running Or example:

\[
\text{Or}([x \ y \ ys \ ...] \rightarrow \\
\quad \text{Let}([\text{Binding("t" \ x)])} \\
\quad \text{If(Id("t") \ Id("t") \ Or([y \ ys \ ...])));}
\]

Matching \(\text{Or}([\text{true \ Not(true) \ false \ true}])\) against \(\text{Or}([x \ y \ ys \ ...])\) produces the environment

\[
\gamma = \{\alpha \rightarrow \text{true}, \beta \rightarrow \text{Not(true)}, \delta \rightarrow [\text{false \ true}]\}
\]

and substituting \(\gamma\) into the rule’s rhs produces
\[
\begin{align*}
\text{value}/\text{value} &= \{\} \\
x/x &= \{\} \\
e/\alpha &= \{\alpha \rightarrow e\} \\
[e_1 \ldots e_n]/[p_1 \ldots p_n] &= \bigcup_{i=1..n}(e_i/p_i) \\
[e_1 \ldots e_n \ldots e_{n+k}]/[p_1 \ldots p_n p^*] &= \bigcup_{i=1..n}(e_i/p_i) \cup \text{merge}([e_{n+i}/p]_{i=1..k}) \\
(C e_1 \ldots e_n)/(C p_1 \ldots p_n) &= \bigcup_{i=1..n}(e_i/p_i)
\end{align*}
\]

\[
\begin{align*}
\gamma \cdot \text{value} &= \text{value} \\
\gamma \cdot x &= x \\
\gamma \cdot [p_1 \ldots p_n] &= [\gamma \cdot p_1 \ldots \gamma \cdot p_n] \\
\gamma \cdot [p_1 \ldots p_n p^*] &= [\gamma \cdot p_1 \ldots \gamma \cdot p_n \text{ ++ split}(\gamma, p)] \\
& \quad \quad \quad \text{(where ++ is concatenation)} \\
\{\ldots, \alpha \rightarrow b, \ldots\} \cdot \alpha &= \text{toTerm}(b) \\
\gamma \cdot (C p_1 \ldots p_n) &= (C \gamma \cdot p_1 \ldots \gamma \cdot p_n)
\end{align*}
\]

\[
\begin{align*}
\text{merge}(\{[\alpha_1 \rightarrow b_{11} \ldots \alpha_n \rightarrow b_{n1}] \ldots [\alpha_1 \rightarrow b_{1k} \ldots \alpha_n \rightarrow b_{nk}]\}) &= \{\alpha_1 \rightarrow [b_{11} \ldots b_{1k}] \ldots \alpha_n \rightarrow [b_{n1} \ldots b_{nk}]\} \\
\text{split}(\{[\alpha_1 \rightarrow [b_{11} \ldots b_{1k}] \ldots \alpha_n \rightarrow [b_{h1} \ldots b_{hk}]\}, p) &= ([\alpha_1 \rightarrow b_{11} \ldots b_{1k}] \ldots \alpha_n \rightarrow b_{h1} \ldots b_{hk}] \cdot p) \\
\text{toTerm}(p) &= p \\
\text{toTerm}([b_1 \ldots b_n]) &= (\text{toTerm}(b_1) \ldots \text{toTerm}(b_n)) \\
\end{align*}
\]

**Figure 2: Matching and substitution**

Let ([Binding("t" true)])

If (Id("t") Id("t") Or([Not(true) false true]))

Later, we will need to compute unifications as well. We omit showing the algorithm; it is straightforward since we disallow duplicate pattern variables (as seen in the next section).

**Well-formedness of Transformations**

The definitions we have given for matching and substitution are not well-behaved for all patterns. Even the crucial property that \((e/p) \cdot p = e\) whenever \(e/p\) exists fails to hold in certain situations, such as when a pattern’s ellipsis contains no pattern variables (e.g., \((3*)\)). For this reason and others, we require a number of criteria to hold for the LHS and RHS of each rule. Except for the last criterion
about ellipses, these requirements were given in section 3.2.1. We repeat them here for convenience:

1. Each pattern variable in the rhs also appears in the lhs. Otherwise the pattern variable would be unbound during expansion.

2. The lhs pattern contains no references or declarations. Rather, these should be bound to its pattern variables during expansion.

3. Each transformation’s lhs must have the form (C p₁ ... pₙ). We will rely on this fact when showing that unexpansion is an inverse of expansion in section 4.5.2.

4. Each pattern variable appears at most once in the lhs and at most once in the rhs. Allowing duplicate pattern variables complicates matching, unification, and proofs of correctness. It also copies code and, in the worst case, can exponentially blow up programs. We therefore disallow duplication, with the sole exception of variables bound to atomic terms.

5. The transformations’ lhs must be disjoint. We explain the reason for this in section 4.4.1.

6. (New) An ellipsis of depth n must contain at least one pattern variable that either appears at depth n or greater on the other side of the rule, or does not appear on the other side of the rule. Otherwise it is impossible to know how many times to repeat its pattern during substitution. (The depth of an ellipsis measures how deeply nested it is within other ellipses; a top-level ellipsis has depth 1, an ellipsis within an ellipsis depth 2, and so forth.) For example, the rule Bad(x) \rightarrow List([x ...]) is invalid, as is Bad(x) \rightarrow List([2 ...]).

The first two restrictions were further justified by our formalization of expansion and unexpansion in Coq (section 4.5.4), where they occurred naturally as preconditions for proofs.

 Applying Transformations

A rulelist rs is an ordered list of desugaring rules \( p_i \rightarrow p'_i \), where each rule is well-formed according to the criteria just described. A term \( e \) can then be expanded with respect to rs by matching \( e \) against each \( p_i \) in turn, and substituting the resulting bindings into \( p'_i \) if successful. In addition, the index \( i \) of the case that was successful must also be returned. This index will be used during unexpansion to know which rule to use, as multiple rules may have similar or identical rhs.

Formally,

\[
\text{exp}_{rs} e = (j, (e/p_j) \bullet p'_j)
\]

for \( j = \min \{ i | e \geq p_i \} \).
Unexpansion proceeds in reverse, matching against $p'_i$ and then substituting into $p_i$. Recall, however, that our well-formedness criteria insisted that the pattern variables in a rule’s rhs pattern be a subset of those in its lhs pattern, but not vice versa. This allows a rule to “forget” information when applied forward. Allowing information to be lost substantially increases the set of desugarings expressible in our system in exchange for breaking the symmetry between expansion and unexpansion. Because pattern variables may be dropped in the rhs, the unexpansion of a term $e'$ takes an additional argument—the term $e$ that was originally expanded—with which to bind pattern variables in $p_i$ that do not appear in $p'_i$. Formally,

\[
\text{unexp}_\text{rs} \ (j, e') \ e \ = \ (((e/p_j) \cup (e'/p'_j)) \cdot p_j
\]

Notice that $e$ contains a good deal of redundant information. Since $p_j$ and $p'_j$ are statically known, it suffices to store only the environment $\gamma = e/p_j$ restricted to the pattern variables not free in $p'_j$. We will say that $\gamma$ stands in for $e$, and overload $\text{unexp}_\text{rs}$ by writing:

\[
\text{unexp}_\text{rs} \ (j, e') \ \gamma \ = \ (\gamma \cup (e'/p'_j)) \cdot p_j
\]

Because unexpansion usually occurs after reduction steps have been taken, in general the term being unexpanded is different from the output of expansion.

**Overlapping Rules**

When multiple rules overlap (i.e., when two different rules could apply to the same term), the Emulation property may be violated. For illustration, suppose a core language contains a MaxAcc primitive that takes a list of numbers and a starting maximum, and in each reduction step pops the list and updates the starting maximum. Furthermore, say we want to extend this language with simple sugar for finding the maximum of a list of numbers, that fails with a runtime exception on empty lists. This could be achieved with the following transformation rules:

\[
\begin{align*}
\text{Max}([]) & \rightarrow \text{Raise("empty list")}; \\
\text{Max}(xs) & \rightarrow \text{MaxAcc}(xs, -\infty);
\end{align*}
\]

These rules are problematic, however, as demonstrated by the evaluation of the surface term $\text{Max}([-\infty])$. It expands to the core term $\text{MaxAcc}([-\infty], -\infty)$, which reduces (in the core) to $\text{MaxAcc}([], -\infty)$, which unexpands by the second rule above to $\text{Max}([])$. Thus, the core sequence is:

\[
\begin{align*}
\text{MaxAcc}([-\infty], -\infty) \\
\rightarrow \text{MaxAcc}([], -\infty)
\end{align*}
\]
and the derived surface evaluation sequence is:

\[ \text{Max}([-\infty]) \]

\[ \rightarrow \text{Max}([]) \]

But the \( \text{Max}([]) \) surface step flagrantly violates the Emulation property! It expands into \( \text{Raise("empty list")} \), which is very different from the core term \( \text{MaxAcc}([], -\infty) \) it purports to represent.

Fortunately, the \( \text{Max} \) sugar becomes safe with the following minor rewrite to make apparent the fact that the second rule only applies to non-empty argument lists:

\[ \text{Max}([]) \rightarrow \text{Raise("Max: given empty list")}; \]
\[ \text{Max}([x, xs ...]) \rightarrow \text{MaxAcc}([x, xs ...], -\infty); \]

The scenario just described now plays out differently. The initial expansion and core reduction step remain the same, but when \( \text{MaxAcc}([], -\infty) \) is unexpanded, that unexpansion fails because the term does not match the \( \text{rhs} \) pattern \( \text{MaxAcc}([x, xs ...], -\infty); \) thus this step is safely skipped.

\text{Confection} implements a static check that admits the second definition but not the first. It checks that the \( \text{lhs} \)s of the rules are pairwise disjoint. This ensures that after unexpansion, only the same rule that was unexpanded applies. We formally state the rule and what it gains us in section 4.5.1.

4.4.2 Performing Transformations Recursively

We have described how to perform a single transformation. We will now describe how to use tags to keep track of which rule each core term came from, and how to use this information to perform recursive expansion and unexpansion of terms, a.k.a. desugaring and resugaring.

Tagging

We define two kinds of tags: \textbf{Head} tags mark the outermost term constructed by a rule application, and \textbf{Body} tags mark each non-atomic term constructed by a rule application. \textbf{Body} tags serve to distinguish rule-generated code from user-written code, thereby maintaining Abstraction. They are automatically inserted into each rule’s \( \text{rhs} \) during parsing. Crucially, these tags can be considered simply part of the \( \text{rhs} \) pattern, so they do not interfere with the definitions of rule expansion and unexpansion.

As noted in section 4.2.4, it is sometimes desirable to make sugar-produced terms transparent to the user. \text{Confection} allows sugar authors to do so by
prefixing a term with ‘!’. Each Body tag contains a boolean indicating whether it was made transparent in this way; we will call these tags transparent or opaque, as appropriate.

Head tags serve a dual role. First, they store the index of the rule which was applied, thus ensuring that only that rule may be applied in reverse during resugaring; this is necessary to maintain Emulation. Second, when the rhs of a rule contains fewer pattern variables than the lhs, Head tags store the bindings $\gamma$ for those pattern variables present in the lhs but not in the rhs.

While Head tags mark which rule they originated from, Body tags do not. In principle, this simplification would allow one rule to successfully unexpand using chunks of code produced by another rule. In practice, it is hard to construct scenarios in which this actually occurs and, in any case, it does not affect our goal properties.

### Recursive Expansion and Unexpansion

We have defined how to non-recursively expand and unexpand a term with respect to a rulelist, and will now define recursive expansion and unexpansion, a.k.a. desugaring and resugaring. To desugar a complete core term, recursively traverse it in-order, applying $\text{exp}_{rs}$ at each node:

\[
\begin{align*}
\text{desugar}_{rs} \text{ value} & = \text{value} \\
\text{desugar}_{rs} x & = x \\
\text{desugar}_{rs} (C \, e_1, ..., e_n) & = \text{desugar}_{rs} (\text{Tag} (\text{Head} \, i \, \gamma) \, e') \\
& \text{where } \gamma \text{ stands in for } (C \, e_1, ..., e_n)/p_i \\
& \text{when } \text{exp}_{rs} (C \, e_1, ..., e_n) = (i, e') \\
\text{desugar}_{rs} (C \, e_1, ..., e_n) & = (C \, \text{desugar}_{rs} e_1, ..., \text{desugar}_{rs} e_n) \\
& \text{otherwise} \\
\text{desugar}_{rs} (e_1 ... e_n) & = (\text{desugar}_{rs} e_1 ... \text{desugar}_{rs} e_n) \\
\text{desugar}_{rs} (\text{Tag} \, o \, e) & = (\text{Tag} \, o \, \text{desugar}_{rs} e)
\end{align*}
\]

Resugaring can be performed by traversing a term, this time performing $\text{unexp}_{rs} (i, e) \, \gamma$ for any term $e$ tagged with $(\text{Head} \, i \, \gamma)$. Thus $\text{resugar}_{rs}$ identifies the specific sugars that need to be unexpanded by finding Head tags, and delegates the sugar-specific unexpansions—which include eliminating Body tags—to $\text{unexp}_{rs}$.

If the unexpansion of any particular term fails, then resugaring as a whole fails, since the tagged term in question cannot be accurately represented as the result of an expansion nor shown as-is. Furthermore, resugaring should fail
4.4 The Transformation System

if any opaque Body tags remain. This ensures that code originating in sugar (and therefore wrapped in Body tags) is never exposed, guaranteeing Abstraction.

\[
\begin{align*}
\text{resugar}_{rs} e &= R'_{rs} e & \text{when } R'_{rs} e \text{ has no opaque tags} \\
\text{resugar}_{rs} e &= \bot & \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
R'_{rs} \text{ value} &= value \\
R'_{rs} x &= x \\
R'_{rs} (\text{Tag} (\text{Body} b) e) &= (\text{Tag} (\text{Body} b) R'_{rs} e) \\
R'_{rs} (\text{Tag} (\text{Head} i \gamma) e') &= \text{unexp}_{rs} (i, R'_{rs} e') \gamma \\
R'_{rs} (C e_1, ..., e_n) &= (C R'_{rs} e_1, ..., R'_{rs} e_n) \\
R'_{rs} (e_1 ... e_1) &= (R'_{rs} e_1 ... R'_{rs} e_n)
\end{align*}
\]

4.4.3 Lifting Evaluation

We can now put the pieces together to see how Confection works as a whole.

We have defined desugaring and resugaring with respect to terms expressed in our pattern language. Real languages’ source terms do not start in this form, so we will require functions for converting between syntax in the surface and core languages and terms in our pattern language. We will call these \(s \rightarrow e\), \(e \rightarrow s\), \(c \rightarrow e\), and \(e \rightarrow c\), using \(s\), \(c\), and \(e\) as abbreviations for surface, core, and term respectively. With these functions, we can define functions to fully desugar and resugar terms in the language’s syntax:

\[
\begin{align*}
D &= s \rightarrow e ; \text{desugar}_{rs} ; e \rightarrow c \\
R &= c \rightarrow e ; \text{resugar}_{rs} ; e \rightarrow s
\end{align*}
\]

A surface reduction sequence for a deterministic language can now be computed as follows:

\[
\begin{align*}
def \text{showSurfaceSequence}(s): \\
&\text{let } c = D(s) \\
&\text{while } c \text{ can take a reduction step:} \\
&\quad \text{let } s' = R(c) \\
&\quad \text{if } s': \text{ emit}(s') \\
&\quad c := \text{step}(c)
\end{align*}
\]

Implementing this requires a step relation; though most languages don’t provide one natively, section 4.6 describes how to obtain one.

For a nondeterministic language, the aim is to lift an evaluation tree instead of an evaluation sequence. The set of nodes in the surface tree can be found by keeping a queue of as-yet-unexplored core terms, initialized to contain just \(\text{desugar}(s)\), and repeatedly dequeuing a core term and checking whether it can
be resugared. If it can, add its resugaring to the node set, and either way add
the core terms it can step to to the end of the queue. The tree structure can be
reconstructed with additional bookkeeping.

We have a complete implementation of Confection, in which all examples
from this chapter were run. It uses a user-written grammar file that specifies gram-
mars for both the core and surface syntax, and a set of rewrite rules. Though
the grammars and rewrite rules mimic the syntax used by Stratego [6], the rules
obey the semantics described in this chapter. The rules are also checked against
the well-formedness criteria of section 4.4.1 thus ensuring that our results hold.
Confection is available at:

http://cs.brown.edu/research/plt/dl/resugaring/v1/

4.5 Formal Justification

We will now justify many of our design decisions in terms of the formal proper-
ties they yield, and ultimately prove the Emulation and Abstraction properties
relative to some reasonable assumptions about the underlying language.

4.5.1 Transformations as Lenses

We have found it helpful to view our transformation rules from the perspective
of lenses [27]. In particular, the disjointness condition that prevents the Max
problem of section 4.4.1 can be seen as a precondition for the lens laws, and the
proof that our system obeys the Emulation property rests upon the fact that its
transformations form lenses.

A lens has two sets C and A, together with partial functions get : C \rightarrow A and
put : A \times C \rightarrow C that obey the laws,

\[
\begin{align*}
\text{put} (\text{get} (c, c)) &= \bot \text{ or } c & \forall c \in C & \text{GetPut} \\
\text{get} (\text{put} (a, c)) &= \bot \text{ or } a & \forall a \in A, c \in C & \text{PutGet}
\end{align*}
\]

Taking C = e and A = (\mathbb{N}, e) gives exp_{rs} and unexp_{rs} the signatures of get and
put, respectively. Thus if they additionally obey the two laws, they will form
a lens. We will give a necessary and sufficient condition for the laws to hold,
and later show that when they do hold, the Emulation property is preserved by
resugaring.
The GetPut Law

The GetPut law applied to our transformations states that whenever it is well-defined,
\[ \text{unexp}_{rs} (\text{exp}_{rs} e) e = e \]

Expanding the definitions produces:
\[ ((e/p_i) \cup ((e/p_i) \bullet p'_i/p'_i)) \bullet p_i = e \]

This law can be shown to hold without further preconditions.

**Lemma 1.** The GetPut law holds whenever it is well-defined.

**Proof.** Clearly \((e/p_i) \bullet p'_i/p'_i \subseteq e/p_i\). Thus \((e/p_i) \cup ((e/p_i) \bullet p'_i/p'_i) = e/p_i\), and \((e/p_i) \cup ((e/p_i) \bullet p'_i/p'_i)) \bullet p_i = (e/p_i) \bullet p_i\). And since \(e\) is closed, \((e/p_i) \bullet p_i = e\). \(\Box\)

The PutGet Law

The PutGet law states that whenever it is well-defined,
\[ \text{exp}_{rs} (\text{unexp}_{rs} (j,e')) = (j,e') \]

Expanding the definitions gives that,
\[ (i,((e/p_j) \cup (e'/p'_j)) \bullet p_j/p_i) \bullet p'_i = (j,e') \]
\[ \text{for } i = \min \{ i | ((e/p_j) \cup (e'/p'_j)) \bullet p_j \geq p_i \} \]

This law, however, does not hold for all possible rulelists. In fact, we saw a situation in which it fails—the Max sugar in section 4.4.1—as well as the alarming consequences of the failure. In that section we introduced the disjointness condition. Forcing the LHSs of rules to be disjoint ensures that the surface representation of a core term, which was obtained by unexpanding that term through some rule, could only expand via the same rule, thereby obtaining the core term it is supposed to represent.

We can now say precisely what the disjointness check gains us: it is both necessary and sufficient for the PutGet law to hold. We will see later that the PutGet law ensures Emulation. The reverse is not true, however, so the disjointness check is sufficient but not necessary to achieve Emulation, and a tighter test could be found (although it would almost certainly have to make stronger assumptions about evaluation in the core language than we do).

**Definition 1.** The disjointness condition for a rulelist \(rs = p_1 \rightarrow p'_1, \ldots, p_n \rightarrow p'_n\) states that \(p_i \vee p_j = \perp\) for all \(i \neq j\).
**Theorem 1.** For any rulelist \( rs \), the PutGet law holds iff the disjointness condition holds.

**Proof Sketch.** The law states that:

\[
(i, ((e/p_j) \cup (e'/p'_j)) \cdot p_j/p_i) \cdot p'_i = (j', e')
\]

for \( i = \min \{ i | ((e/p_j) \cup (e'/p'_j)) \cdot p_j \geq p_i \} \)

First, note that the law always holds when \( i = j \), so it is sufficient to consider \( i < j \). Let \( \gamma_1 = (e/p_j) \cup (e'/p'_j) \). If the PutGet law does not hold, then \( \gamma_1 \cdot p_j / p_i \) exists, so \( p_i \lor p_j \) exists. On the other hand, if \( p_i \lor p_j \) exists for some \( i < j \), then \( e \) and \( e' \) can be chosen such that \( (\gamma_1 \cdot p_j) / p_i \) is guaranteed to be well-defined, forcing the law to not hold.

**Confection** statically checks that the rulelist obeys the well-formedness criterion from section 4.4.1 and the disjointness criterion, thereby ensuring that the lens laws will hold. We will next show that these lens laws imply that desugaring and resugaring are inverses of each other, which is the crux of the Emulation property.

4.5.2  **Desugar and Resugar are Inverses**

We show that \( \text{desugar} \) and \( \text{resugar} \) are inverses of each other, after noting that surface and core terms have slightly different shapes.

**Definition 2.** A surface term is a term without any tags \( (\text{Tag} o e) \).

**Definition 3.** A core term is a term that contains no construct \( C \) that appears in the outermost position of any \( \text{LHS} \) of the rulelist.

As expected, desugaring produces core terms, and resugaring produces surface terms.

**Lemma 2.** If \( \text{desugar}_{rs} e = e' \), then \( e' \) is a core term. And if \( \text{resugar}_{rs} e' = e \), then \( e \) is a surface term.

**Proof.** By induction over the term.

Further, \( \text{desugar} \) and \( \text{resugar} \) are idempotent over core and surface terms, respectively.

**Lemma 3.** Whenever \( e \) is a surface term, \( \text{resugar}_{rs} e = e \). And whenever \( e' \) is a core term, \( \text{desugar}_{rs} e' = e' \).

**Proof.** By induction over the term.
Theorem 2. Assume that the lens laws hold for all transformations. Then for all surface terms e, \( \text{desugar}_{rs} e = e' \) implies \( \text{resugar}_{rs} e' = e \). And for all core terms e', \( \text{resugar}_{rs} e' = e \) implies \( \text{desugar}_{rs} e = e' \).

Proof. For both cases, proceed by induction over the term. The two nontrivial cases are \( \text{resugar}_{rs} (\text{desugar}_{rs} (C e_1, ..., e_n)) \) and \( \text{desugar}_{rs} (\text{resugar}_{rs} (\text{Tag} (\text{Head} i \gamma) e')) \). For brevity, call \( \text{desugar}_{rs} \) Des, call \( \text{exp}_{rs} \) E, call \( \text{resugar}_{rs} \) Res, and call \( \text{unexp}_{rs} \) U.

In the first case,

\[
\begin{align*}
\text{Res} (\text{Des} (C e_1, ..., e_n)) &= \text{Res} (\text{Des} (\text{Tag} (\text{Head} i \gamma) e')) \\
&= \text{Res} (\text{Tag} (\text{Head} i \gamma) (\text{Des} e')) \\
&= U (i, \text{Res} e') (C e_1, ..., e_n) \\
&= U (i, e') (C e_1, ..., e_n) \quad \text{(by I.H.)} \\
&= (C e_1, ..., e_n) \quad \text{(by GetPut)} \\
\end{align*}
\]

In the second case,

\[
\begin{align*}
\text{Des} (\text{Res} (\text{Tag} (\text{Head} i \gamma) e')) &= \text{Des} (U (i, \text{Res} e') \gamma) \\
&= \text{Des} (C e_1, ..., e_n) \quad \text{(using w.f.)} \\
&= \text{Des} (\text{Tag} (\text{Head} i \gamma) (\text{Res} e')) \quad \text{(by PutGet)} \\
&= (\text{Tag} (\text{Head} i \gamma) (\text{Des} (\text{Res} e'))) \\
&= (\text{Tag} (\text{Head} i \gamma) e') \quad \text{(by I.H.)} \\
\end{align*}
\]

4.5.3 Ensuring Emulation and Abstraction

We now precisely state and prove the Emulation and Abstraction properties, making use of the results of the last section.

Theorem 3 (Emulation). Given a well-formed rulelist rs, each surface term in the generated surface evaluation sequence desugars into the core term which it represents, so long as:

- \( e \rightarrow c \ (c \rightarrow e \ c) = c \) for all c
- \( s \rightarrow e \ (e \rightarrow s \ e) = e \) for all e
- \( (c \rightarrow e \ c) \) is a core term for all c
The disjointness condition holds for rs

Proof. In the stepping algorithm, $s'$ represents $c$ in each iteration, so we would like to show that if $s'$ occurs in the surface evaluation sequence then $D(s') = c$. If $s'$ occurs in the surface sequence, then resugaring must have succeeded with $R(c) = s'$. Thus we simply need to show that $D(R(c)) = c$ for all terms $c$ in the core language, i.e., that

$$e -> c (\text{desugar}_rs (s -> e (e -> s (\text{resugar}_rs (c -> e c)))))) = c$$

The preconditions of theorem 2 are satisfied. This expression then consists of three pairs of functions and their inverses, so the equation holds. □

To state Abstraction precisely, we must first define the origin of a term. Since it is possible for two different surface evaluation sequences to contain terms which are identical up to tagging but have different origins, the origin of a term must be defined with respect to a surface evaluation sequence (and the corresponding core evaluation sequence).

**Definition 4.** The origin of an occurrence of a term within a given evaluation sequence is defined by:

- Atomic terms have no origin.
- All subterms of the original input term have user origin.
- When a transformation rule is applied to a term (either forward or in reverse), terms bound to pattern variables retain their origins, but all other terms on the rhs have sugar origin, and all other terms on the lhs have user origin.
- Terms maintain their origin through evaluation.

Our use of Body tags purposely mimics this definition, so that Abstraction is nearly true by construction.

**Theorem 4** (Abstraction 1). The surface-level representation of a term $e$ contains only subterms of user origin, except as explicitly allowed by transparency marks ($!$).

**Proof Sketch.** Check that the application of transformation rules both forward and in reverse preserves the invariant that a term has sugar origin iff it is tagged with at least one Body tag, and user origin otherwise. Now see that resugar$_rs$ always fails if any opaque Body tags remain. □

**Theorem 5** (Abstraction 2). Terms of user origin are never hidden by unexpansion.

**Proof Sketch.** Each subterm in the rhs of a rule is wrapped in a Body tag; thus only terms tagged with a Body tag can match against it to be unexpanded. As argued above, only terms of sugar origin may be tagged with a Body tag. □
Notice that theorems 2 and 3, which together prove Emulation, work for any definition of rule expansion and unexpansion that obeys the lens laws. Consequently, our expansion/unexpansion mechanism does not need to be defined solely through the pattern-matching rules we have presented; it can be replaced by a different one that (i) obeys the lens laws, and (ii) retains a tagging mechanism for guaranteeing Abstraction.

4.5.4 Machine-Checking Proofs

We have made substantial progress formalizing our transformation system in the Coq proof assistant [89]. We have formalized a subset of our pattern language, as well as matching, substitution, unification, expansion, and unexpansion, and the disjointness condition. Atop these definitions we have constructed formal proofs that:

1. Matching is correct with respect to substitution.
2. Unification is correct with respect to substitution and matching.
3. Expansion and unexpansion of well-formed rules (as defined in Section 4.4.1) that pass the first static check obey the lens laws.

This formalization helped us pin down the (sometimes subtle) well-formedness criteria of section 4.4.1.

Our formalization does not, however, address tags or ellipsis patterns. It would be straightforward to add tags. Handling ellipses, though, would require significantly more work: when patterns may contain ellipses, substitution is no longer compositional on p. For instance, γ • [p*] is not a function of γ • p.

4.6 Obtaining Core-Language Steppers

Confection assumes it has access to the sequence of core-language terms produced by evaluation, each ornamented with the tags produced by the initial desugaring—but typical evaluators provide neither! Fortunately this information can be reconstructed with little or no modifications to the evaluator, even if it compiles to native code. We now describe in general terms how this can be accomplished. In what follows, we will use the term stepper [125] for an evaluator that, instead of just producing an answer, produces the sequence of core terms generated by evaluation.

The essence of reconstructing each term is simple: it is the current continuation at that point of evaluation. Therefore, we need to be able to capture, and present, the continuation as source. (The tags are introduced statically, so the
process of reconstructing the code can reconstitute these tags alongside.) Either the evaluator can be modified to reconstruct the source as it runs, or a pre-compilation step may be introduced that does so in the host language itself. We have used both approaches.

To construct the source term at an evaluation step, we have multiple options. For instance, we can convert the code to continuation-passing style, with each continuation parameter represented as a pair: the closure that runs, combined with a function to produce a core language representation of the closure.

Instead, our steppers use a more efficient transformation based on A-normalization—to obtain a representation of each stack frame. To traverse the stack and accumulate these representations, we have two choices. In languages with generalized stack inspection features like continuation marks, or ways of emulating them (as discussed by Pettyjohn, et al.), we can exploit these existing run-time system features. In other cases, our steppers simply instrument the code to maintain a global stateful stack onto which they push and pop frames.

In addition, our core steppers instrument the code so that it pauses at every evaluation step to emit the representation of the current continuation. This can be done by using resumable exceptions, native continuations, and so forth, but even in languages without such features, it is easy to achieve: simply pause execution to print the continuation, before resuming computation.

Using this combination of techniques, we have created steppers for Racket, Pyret, and PLT Redex (a tool for studying language semantics). In the process we have used both the continuation mark and “shadow stack” strategies. The Racket stepper is notable because although Racket already has a stepper, it is much weaker than ours. For example, it does not handle state, continuations, or any user-defined macros. Obtaining a core stepper from PLT Redex is trivial because the tool already provides a function that performs a single evaluation step.

Performance Our prototype core steppers for Racket and Pyret induce a 5-40% overhead, depending on how large the stack grows and the relative mix of instrumented and uninstrumented calls. In addition, we must pay for serialization and context-switching because the Confection implementation is an external process. This additional cost can obviously be eliminated by implementing Confection inside the host language runtime.
In this section we describe sugars we implemented to test the expressiveness of our system. (Section 4.3 shows a non-trivial outcome.) In what follows, we manually verified that each of the implemented sugars showed the expected surface steps.

4.7.1 Building on the Lambda Calculus

To see how far we could push building a useful surface language atop a small core, we constructed a simple stateful language in PLT Redex. It contains only single-argument functions, application, if statements, mutation, sequencing, and amb (which nondeterministically chooses among its arguments), and some primitive values and operations. Atop this we defined sugar for multi-argument functions, Thunk, Force, Let, Letrec, multi-arm And and Or, Cond; and atop these, a complex Automaton macro \[48\]. All of these behave exactly as one might expect other than Letrec and Automaton, discussed below.

The Letrec sugar does not show any intermediate steps in which some but not all branches have been evaluated; thus the surface evaluation shows the branches all evaluating in one step. For instance, \[(\text{letrec } ((x y) (y 2)) (+ x y))\] steps directly to \[(+ 2 2)\]. Though this initially surprised us, it is actually the correct representation of the semantics of letrec; from our perspective, showing intermediate steps would necessarily be inaccurate and violate Emulation.

The Automaton macro had the same problem until we made some small, semantics-preserving refactorings: lifting some identifiers into Let bindings, and adding \! on recursive annotations. Figure 3 shows a run in Redex’s evaluation visualizer; the underlying core evaluation took 264 steps\[1\].

4.7.2 Return

Having first-class access to the current continuation is a powerful mechanism for defining new control flow constructs. Racket does so with the built-in function call/cc, that takes a function of one argument and calls it with the program’s

---

\[1\] It may be concerning that this figure appears to contain unbound identifiers, such as \texttt{M} and \texttt{init}. These are actually the names of function values, as determined by Racket. (These names can be obtained by calling \texttt{object-name} on the function.) We could instead have simply displayed function values generically—e.g. as \texttt{#<procedure> }—but that would have resulted in a more opaque evaluation sequence. A better method might have been to explicitly show the environment, in which \texttt{M} and \texttt{init} are bound. Finding the best way to present the surface evaluation sequence is an interesting research question and outside the scope of this thesis.
Figure 3: Automaton macro execution example
current continuation. Using it, we can define a `return` sugar that returns early from a function:

```plaintext
Return(x) ->
    Let([Bind("%RES", x)],
        [Apply(Id("%RET"), [Id("%RES")])]);

Function(args, body) ->
    Lambda(args, Apply(Id("call/cc"),
        [Lambda(["%RET"], body)]));
```

(The definition of `function` is necessary to mark the point that `return` should return to.) With this definition in place, we can see evaluation sequences such as:

```
(+ 1 ((function (x) (+ 1 (return (+ x 2)))) (+ 3 4)))
   --> (+ 1 ((function (x) (+ 1 (return (+ x 2)))) 7))
   --> (+ 1 (+ 1 (return (+ 7 2))))
   --> (+ 1 (+ 1 (return 9)))
   --> (+ 1 9)
   --> 10
```

This example illustrates that our approach is robust enough to work even in the presence of dynamic control flow.

### 4.7.3 Pyret: A Case Study

Pyret, shown in section 4.3, is a new language. It makes heavy use of syntactic sugar to emulate the syntax of other programming languages like Python. This sugar was implemented by people other than this thesis's author, and written as a manual compiler, not as a set of rules; it was also implemented without any attention paid to the limitations of this work. Thus, the language makes for a good case study for the expressiveness of our work.

We restricted our attention to sugar relevant to evaluation. Pyret has built-in syntactic forms for writing tests, and can run code both in a “check” mode that only runs these tests, or in “normal” mode that runs code. We focused on “normal” mode since it is most relevant to evaluation. There were two pieces of sugar we were unable to express and one that required modification to show ideal surface steps; we describe these in more detail below. As fig. 4 shows, we
Figure 4: Syntactic sugar in normal-mode Pyret

were able to handle almost all of Pyret’s sugar. An example of the result in action was shown in section 4.3.

We were unable to fully handle algebraic datatype declarations because they splice one block of code into another in a non-compositional manner; we believe these could be expressed by adding a block construct that does not introduce a new scope (akin to Scheme’s `begin`).

We were also unable to handle `graph`, which constructs cyclic data. It has a complex desugaring that involves creating and updating placeholder values and compile-time substitution. This could be solved either by expanding the expressiveness of our system or by adding a new core construct to the language. There is always a trade-off between the complexity of the core language and the complexity of the desugaring; when a feature can only be implemented through a highly non-compositional sugar like this, it may make sense to instead add the feature to the core language.

Finally, the desugaring for binary operators needed to be modified to show helpful surface evaluation sequences. The desugaring follows a strategy similar to that of Python, by applying the `_plus` method of the left subexpression to the right subexpression:

```
Op(s, "+", x, y) ->
App(s, Bracket(s, x, Str(s, "+\plus")), [y]);
```
Given the term \(1 + (2 + 3)\), we would expect evaluation to step first to \(1 + 5\) and then to 6. Unfortunately, Confection shows only this surface evaluation sequence:

\[
1 + (2 + 3) \rightarrow 6
\]

The core evaluation sequence reveals why:

\[
1. ["._plus"](2.["._plus"](3))
\rightarrow <\text{func}>(2.["._plus"](3))
\rightarrow <\text{func}>(<\text{func}>(3))
\rightarrow <\text{func}>(5)
\rightarrow 6
\]

(<\text{func}> denotes a resolved functional). To show the term \(1 + 5\), Emulation requires that it desugar precisely into one of the terms in the core sequence; but it desugars to \(1.["._plus"](5)\), which has a different shape than any of the core terms.

The fundamental problem is the order of evaluation induced by this desugaring: first the left subexpression is evaluated, then the \(_\text{plus}\) field is resolved, then the right subexpression is evaluated, then the “addition” is performed. We can obtain a more helpful surface sequence by instead choosing a desugaring that forces the left and right subexpressions to be evaluated fully before resolving the operation, as shown in fig. 5.

This desugaring constructs a temporary object \(\{\text{left}: x, \text{right}: y\}\), and then computes \(\text{temp}.\text{left}._\text{plus}(\text{temp}.\text{right})\). Notice that this desugaring slightly changes the semantics of binary operators; the difference may be seen when the
right subexpression mutates the _plus field of the left subexpression. In exchange, we obtain the expected surface evaluation sequence:

$$1 + (2 + 3) \rightarrow 1 + 5 \rightarrow 6$$

4.8 Obtaining Hygiene

The implementation discussed in this chapter—Confection—is not hygienic. (For a quick introduction to hygiene, see section 2.1.7) In this section, we sketch out how Confection could be modified to be hygienic. More specifically, we show how it can be modified such that variables defined by users are never bound by variables defined by sugars, and vice-versa.

The required modification is simple. Suppose that:

- $\text{desugar}_{rs}$ gives fresh names to introduced variables (that is, variables on the rhs).
- $\text{resugar}_{rs}$ allows any variable name to match against an introduced variable.

Then both $\text{desugar}_{rs}$ and $\text{resugar}_{rs}$ will be hygienic: user-defined variables and sugar-introduced variables will never be bound by one another.

During desugaring, sugar-introduced variables are those that appeared as variables in the rhs of a sugar. By assumption, all of these variables are given fresh names; thus no user-defined variable can be bound by them, and vice-versa. Why is this proof so easy, when significant effort has gone into defining hygienic macro expansion algorithms? It is because we are assuming that sugars are defined externally to the language, so they aren’t intermingled with user-defined code.

During resugaring, it may look like there are no sugar-introduced variables, since the lhs of desugaring rules cannot contain variables. However, it is possible for the lhs of a rule to drop a pattern variable when that pattern variable is bound to a variable, thus indirectly dropping the variable. For example, take the desugaring rule:

$$\text{desugaring rule: } (\text{Lambda } \alpha. \beta \ast 0) \Rightarrow (\text{Lambda } \alpha. 0)$$

Using this rule, the term $(\text{Lambda } x. x \ast 0)$ will desugar into $(\text{Lambda } x. 0)$. Resugaring would restore the pattern variable $\beta$, thereby re-introducing the $x$ in the body of the Lambda.

Is this re-introduced variable bound by the outer $x$? Surprisingly, it is not. The key point is that in the absence of surface-language scoping rules, where a variable is bound in a surface term is defined by the desugaring of that term. Since the inner
x is dropped during desugaring, it has no binding site. Thus unexpansion is hygienic: every variable that is re-introduced during resugaring has no binding site, so there can be no accidental variable capture. The textual representation of the surface term is misleading in this case. If CONFECTION made the binding structure of the surface term explicit by, e.g., drawing arrows to show where variables were bound, then it would be visibly clear that the inner x was not bound by the outer x.

This may seem like a cop-out, but it is the best that we can do when the scope of the surface language is defined only implicitly through desugaring. In the next chapter, we will show how to resugar core language scope rules to obtain scope rules for the surface language, which will lead to a stronger notion of hygiene.

4.9 RELATED WORK

There is a long history of trying to relate compiled code back to its source. This problem is especially pronounced in debuggers for optimizing compilers, where the structure of the source can be altered significantly [34]. Most of this literature is based on black-box transformations, unlike ours, which we assume we have full control over. As a result, this work tends to be very different in flavor from ours: some of it is focused on providing high-level representations of data on the heap, which is a strict subproblem of ours, or of correlating back to source expression locations, which again is weaker than reconstructing a source term. For this reason, this work is usually also not accompanied by strong semantic guarantees or proofs of them.

One line of work in this direction is SELF’s debugging system [36]. Its compiler provides its debugger with debugging information at selected breakpoints by (in part) limiting the optimizations that are performed around them. This is a sensible approach when the code transformation in question is optimization and can be turned off, but does not make sense when the transformation is a desugaring which is necessary to give the program meaning.

Another line of work in this direction is the compile-time macro error reporting developed by Culpepper, et al. [14]. Constructing useful error messages is a difficult task that we have not yet addressed. It has a different flavor than the problem we address, though: akin to previous work in debugging, any source terms mentioned in an error appear directly in the source, rather than having to be reconstructed.

Deursen, et al. [83] formalize the concept of tracking the origins of terms within term rewriting systems (which in their case represent the evaluator, not the syntactic sugar as in our case). They go on to show various applications, includ-
ing visualizing program execution, implementing debugger breakpoints, and locating the sources of errors. Their work does not involve the use of syntactic sugar, however, while our work hinges on the interplay between syntactic sugar and evaluation. Nevertheless, we have adopted their notion of origin tracking for our transformations.

Krishnamurthi, et al. develop a macro system meant to support a variety of tools, such as type-checkers and debuggers. Tools can provide feedback to users in terms of the programmer’s source using source locations recorded during transformation. The system does not, however, reconstruct source terms; it merely point out relevant parts of the original source. The source tracking mechanisms are based on Dybvig, et al.’s macro system.

Clements implements an algebraic stepper (similar to ours) for Racket—a language that has macros—and thus faces precisely the same problem we address in this chapter. That work, however, side-steps these issues by handling a certain fixed set of macros specially (those in the “Beginner Student” language) and otherwise showing only expanded code. On the other hand, it proves that its method of instrumenting a program to show evaluation steps is correct (i.e., the instrumented program shows the same evaluation steps that the original program produces), while we only show that the lifted evaluation sequence is correct with respect to the core stepper. Thus its approach could be usefully composed with ours to achieve stronger guarantees.

Fisher and Shivers develop a framework for defining static semantics that connect the surface and core languages. They show how to effectively lower a static semantics from a surface language to its core language. This is complementary to our work, which shows how to lift a dynamic semantics from core to surface. This exposes a fundamental difference in starting assumptions: they assume the surface language has a static semantics, while we assume its semantics is defined by desugaring.

In a similar vein, there has been work on ensuring that syntactic extensions respect scope rules and are type sound. We discuss this work in chapter and chapter, respectively.

Model-driven software engineering also draws heavily on bidirectional transformation, because systems are expected to be written in a collection of domain-specific languages that are transformed into implementations. These uses tend to be static, rather than addressing the inverse-mapping problem in the context of system execution (see the survey by Stevens). When the problem we address does arise in this area, it is typically the case that either (i) both the source and target models have implementations, so that surface-level execution traces can be obtained by evaluating in the surface language directly, or (ii) the surface information sought is more limited than the reduction sequence we pro-
vide (in the same ways as for debuggers for optimizing compilers, as described earlier). Applying our results to this area is future work.
In this chapter, 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 $\alpha$-equivalence even though scoping rules have been provided only for the core language.

This chapter comes from work published in ICFP 2017 under the title *Inferring Scope through Syntactic Sugar* (co-authored with Shriram Krishnamurthi and Mitchell Wand) [70].

The next chapter presents type rule resugaring, which subsumes scope resugaring: type rules include the type environment $\Gamma$, and thus also describe scope. Why not use this instead? There are two reasons why it may still be preferrable to resugar scope rules: (i) your language may be untyped; and (ii) scope resugaring is more general and admits a wider variety of constructs. In addition, this chapter’s hygiene theorem (theorem 8) may be of independent interest.

5.1 Introduction

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, refactorters need to know binding structure to perform correct transformations. These tools are hard 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 [7] 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 sup-
posed 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 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 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 7). 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 (such as lifting type-checking rules, as we do in the next chapter) though of course the mechanics of doing so will depend heavily on the notion of semantics to be considered.

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 5.3, and a language for specifying scope in section 5.4.)

2. Infer surface language scoping rules from the core scoping rules. (We give a scope inference algorithm in section 5.5, and show how to make it hygienic in section 5.7)

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 [35, 70], type systems [53], and formal semantics [22]. 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 5.3, we give a formal description of scope as a preorder (which we motivate through examples in section 5.2). 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 5.4, 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 5.5, 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 5.6, and prove that—given reasonable assumptions—desugaring after scope inference will be hygienic in section 5.7.

### 5.2 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 5.4 describes the full version, and sections 5.6.1 and 5.6.3 provide examples. (While the full version of the technique is sometimes important, it has no effect on the examples of this section.)

#### 5.2.1 Example: Single-arm Let

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

\[ e ::= (\text{Let } x^p e_1 e_2) \quad \text{“Let } x^p \text{ equal } e_1 \text{ in } e_2” \]

(Here the superscript \( p \) 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:

\[ (\text{Let } \alpha \beta \delta) \Rightarrow (\text{Apply } (\text{Lambda } \alpha \delta) \beta) \]

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:

\[
\begin{align*}
\text{Apply} &\quad \text{Lambda} \\
\alpha &\quad \beta \\
\delta &\quad \alpha
\end{align*}
\]

where the dotted lines show the tree structure of the AST, and where the teal/-solid arrow means that the Lambda’s parameter (α) can be used in its body (δ). 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:

\[
\begin{align*}
\text{Apply} &\quad \text{Lambda} \\
\alpha &\quad \beta \\
\delta &\quad \alpha
\end{align*}
\]

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 pattern variable to another in the LHS iff there is a similar path in the RHS. If there was a path from α to β in the LHS but not in the RHS, that would mean that a variable (in α) that used to be bound (by β) could become unbound. Likewise, if there was a path between two pattern variables in the RHS but not in the LHS, that could result in unwanted variable capture.

Thus, since there is a path from δ to α in the rule’s RHS, there must also be a path from δ to α in the LHS. This gives:

\[
\begin{align*}
\text{Let} &\quad \alpha \\
\beta &\quad \delta
\end{align*}
\]
In English, this arrow says that the variable declared at $\alpha$ is in scope at the Let’s body $\delta$, as expected.

There are still some missing arrows, however: there should be up 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 pattern variable on the rhs, there should be a similar path on the lhs. Since on the rhs there are paths to each pattern variable from the root, the same should hold true on the lhs:

\[
\text{(lhs)}
\]

This gives a complete scoping rule for this Let construct.

### 5.2.2 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:

\[
e ::= \text{(Let* } b e) \quad "\text{Let-bind } b \text{ in } e"
\]

\[
b ::= \text{(Bind } x^\beta e b) \quad "\text{Bind } x^\beta \text{ to } e, \text{ and export } b's \text{ bindings}"
\]

\[
| \quad \text{EndBinds} \quad "\text{No more bindings}"
\]

This grammar separates out the Let’s bindings into nested subterms. 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:

\[
\text{(Let* (Bind } \alpha \beta \delta) e) \Rightarrow (\text{Apply (Lambda } \alpha (\text{Let* } \delta e)) \beta)
\]

\[
\text{(Let* EndBinds } \alpha) \Rightarrow (\text{Begin } \alpha)
\]

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

\[
\text{(Let* (Bind } x^\beta 1 (\text{Bind } y^\beta 2 \text{EndBinds}) (\text{Plus } x^x y^x))\)
\]

\[
\Rightarrow (\text{Apply (Lambda } x^\beta (\text{Apply (Lambda } y^\beta (\text{Plus } x^x y^x)) 2)) 1)
\]

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:
Unlike in the previous example, this diagram is not (necessarily) complete, since we don’t yet know the scoping rule for Let*. (This will happen when desugaring rules use recursion.) 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 δ to α and from ε to α, the same must be true in the lhs (labeling the arrows for reference):

Notice that we drew the path from ε to α 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 $e$ to $\delta$, however, that needs to be reflected back at the LHS!

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.

5.2.3 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: 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 $c$. 

We make the meaning of arrows precise in section 5.3.2.
These are also precisely the properties that define a preorder. Thus, we will transcribe scope diagrams as preorders, writing \( a \leq b \) 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: \( e \leq \text{Bind} \leq \delta \leq \alpha \)

5.3 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.

5.3.1 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.

- Scoping rules will be as local as possible, only relating a term to its immediate children. Longer relationships will be achieved by transitivity.

- We work on an AST, instead of directly on the surface syntax. As mentioned in section 3.1, variable references (use sites) and declarations (binding sites) must be 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 are already reflected in our definition of (AST) terms in section 3.1. For convenience, we repeat that definition here:

\[
\text{constructor } C ::= \text{name} \quad \text{syntactic construct name} \\
\text{term } e ::= \text{value} \quad \text{primitive value} \\
| (C \ e_1 ... e_n) \quad \text{AST node} \\
| x^p_i \quad \text{variable reference at position } i \\
| x^d_i \quad \text{variable declaration at position } i
\]
To reiterate, 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. \( x_i \). Likewise, we will omit the position subscript i when there is no ambiguity. We will also sometimes write C in place of \((C e_1 ... e_n)\) when there is no danger of ambiguity.

By the last assumption above, we do not support lists \([e_1 ... e_n]\) (which have indefinite arity). Instead we require them to be encoded into fixed-arity constructs.

5.3.2 Basic Definitions

We define scope in terms of a preorder. A preorder \((\leq)\) is a reflexive and transitive relation. It need not be anti-symmetric, however, so it is possible that \(a \leq b\) and \(b \leq a\) for distinct \(a\) and \(b\). (This happens, for example, in letrec.) We capture scope as a preorder on a term \(e\) as follows:

**Definition 5 (Scope).** A scope preorder on a term \(e\) is a preorder \((\leq)\) on the references and declarations in \(e\) such that references are least in this preorder (i.e., nothing is ever smaller than a reference).

**Definition 6.** A reference \(x^R_i\) is in scope of a declaration \(x^D_j\) when \(x^R_i \leq x^D_j\).

**Definition 7.** A declaration \(x^D_i\) is more specific than another \(x^D_j\) when \(x^D_i \leq x^D_j\).

Note that these definitions rely on the existence of a preorder \((\leq)\), but don’t say how to determine it for a given term. We will present scoping rules to do so in section 5.4. These definitions therefore provide very little on their own, but they can be built upon to define some common concepts:

**Definition 8 (Bound).** A reference is bound by the most specific declaration(s) that it has the same name as and is in scope of. More formally, we write:

\[
x^R \mapsto x^D \triangleq x^D \in \min \{x^D_i \mid x^R \leq x^D_i\}
\]

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

\[
\min S \triangleq \{a \in S \mid \forall b \in S. \ b \leq a \text{ and } a \not\leq b\}
\]

**Definition 9 (Unbound).** A reference is unbound (or free) when it is not bound by any declaration.

**Definition 10 (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 by both of them. We also capture the idea of shadowing, where a more specific declaration hides a less specific declaration:

**Definition 11 (Shadowing).** A declaration shadows the most specific declarations that it has the same name as and is more specific than. Formally, \( x^p_i \) shadows \( x^p_j \) when:

\[
 x^p_i \mapsto x^p_j \triangleq x^p_j \in \min \{ x^p_k \mid x^p_i \leq x^p_k \} \text{ and } x^p_j \nleq x^p_i
\]

5.3.3 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 three forms.

First, we prove a simple lemma below showing that shadowing behaves as one would expect. Second, we show (section 5.3.4) that the notion of scope used in “Binding as Sets of Scopes” [24] obeys our scope-as-a-preorder definitions, for an appropriate choice of preorder \( (\leq) \). Finally, in section 5.3.5 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 4 (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 \( x^p_j \) shadows \( x^p_i \) (\( x^p_j \) is the shadowing declaration and \( x^p_i \) is the shadowed declaration), and \( x^p_k \) is in scope of \( x^p_j \). By definition, \( x^p_k \) will be bound by \( \min \{ x^p_l \mid x^p_k \leq x^p_l \} \). But \( x^p_k \leq x^p_j \leq x^p_i \), and \( x^p_j \nleq x^p_i \), so \( x^p_i \) cannot be in this set, and \( x^p_k \) cannot be bound by \( x^p_i \).

5.3.4 Relationship to “Binding as Sets of Scopes”

Scope-as-a-preorder aligns with the notion of scope expressed by Flatt [24]. 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” [24] pp. 3.

This can be expressed in terms of scope-as-a-preorder. Take the preorder \( (\leq) \) to be (the least relation such that):
\[ x^r_i \leq x^r_j \]
\[ x^r_i \leq y^p_i \text{ iff } \text{scope-set}(x^r_i) \supseteq \text{scope-set}(y^p_i) \]
\[ x^p_i \leq y^p_i \text{ iff } \text{scope-set}(x^p_i) \supseteq \text{scope-set}(y^p_i) \]

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.

5.3.5 Axiomatizing Scope as Sets

In this section, we describe an alternative axiomatization of scope, called scope-as-sets. In scope-as-sets, instead of there being a preorder over variables, each declaration has a scope, which is the part of the program in which it can be referenced. For example, the scope of a function’s parameter is that function’s body. The axioms for a term \( e \) are then:

**Scope** Each declaration in \( e \) has a scope—written \( S(x^d) \)—given by the parts of the program in which it can be referenced. We will say that:

- A reference \( x^r \) is in scope of a declaration \( x^p \) when \( x^r \in S(x^p) \).
- One declaration \( x^p \) is more specific than another \( y^p \) when \( S(x^p) \subseteq S(y^p) \).

**Binding** A reference is bound to the most specific declaration that it has the same name as and is in scope of, provided such a unique element exists. Again, if there is no (unique) most specific declaration, we will say that the reference is ambiguously bound by those declarations.

These axioms differ from the scope-as-a-preorder definitions in that the relations “in scope of” and “more specific than” are defined differently. However, all of the other definitions of section 5.3 (including binding, shadowing, ambiguous declarations, etc.) are based on the relations “in scope of” and “more specific than”. Thus all of those definitions apply just as well to scope-as-sets.

Furthermore, the two axiomatizations of scope we have presented are closely related. In fact, scope-as-a-preorder is simply a normalized form of scope-as-sets: there can be more than one scope set that leads to the same scope preorder (with the same binding and shadowing). To begin with, either form can be converted to the other.

**Convert:** From Scope-as-a-Preorder to Scope-as-sets

Given a preorder \( (\leq) \), define its conversion into a scope-as-sets function \( S \) by:

\[
S(y^p) \triangleq \{ x^r | x^r \leq y^p \} \cup \{ x^p | x^p \leq y^p \}
\]
**Conv2: From Scope-as-Sets to Scope-as-a-Preorder** Given a scope-as-sets function $S$, define its conversion into a preorder ($\leq$) by letting ($\leq$) be the least relation such that:

\[ \begin{align*} 
  x^r_i &\leq x^r_i & \text{when} & \quad & i = j \\
  x^r &\leq y^p & \text{when} & \quad & x^r \in S(y^p) \\
  x^d &\leq y^p & \text{when} & \quad & S(x^d) \subseteq S(y^p) 
\end{align*} \]

Both of these conversions preserve all of the binding concepts of section 5.3.2.

**Lemma 5** (Binding Preservation). Both $Conv_1$ and $Conv_2$ preserve “in scope of” and “more specific than” in both directions (and thus all of the concepts of section 5.4).

*Proof.* Under both conversions, “in scope of” is preserved:

\[ x^r \text{ in scope of } y^p \text{ (under sas)} \iff x^r \in S(y^p) \iff x^r \leq y^p \iff x^r \text{ in scope of } y^p \text{ (under sap)} \]

And under both conversions, “more specific than” is preserved:

\[ x^d \text{ more specific than } y^p \text{ (under sas)} \iff S(x^d) \subseteq S(y^p) \iff x^d \leq y^p \iff x^d \text{ more specific than } y^p \text{ (under sap)} \]

Furthermore, these conversions are inverses in one direction: from scope-as-a-preorder to scope-as-sets back to scope-as-a-preorder. This implies that there is a normal form for scope-as-sets, given by $Conv_1(Conv_2(S))$, such that the conversions are exact inverses whenever this normal form is used.

We first show that the conversions are inverses in one direction:

**Lemma 6** (Inverses1). For every scope preorder ($\leq$), $Conv_2(Conv_1(\leq)) = (\leq)$

\[ \begin{align*} 
  x^r_i &\leq x^r_j & \text{when} & \quad & i = j \\
  x^r &\leq y^p & \text{when} & \quad & x^r \in S(y^p) \\
  x^d &\leq y^p & \text{when} & \quad & S(x^d) \subseteq S(y^p) 
\end{align*} \]

*Proof.*

\[ \begin{align*} 
  i = j \iff x^r_i \in \{ x^r \mid x^r \leq y^p \} \\
  x^r \leq y^p \iff x^r \leq y^p \\
  S(x^d) \subseteq S(y^p) \iff x^d \leq y^p 
\end{align*} \]

And they’re inverses in the other direction when $S$ is in a *normal form*. Specifically, we will say that a scope-as-sets function $S$ is in *normal form* when:
Lemma 7 (Inverses2). For any scope-as-sets function \( S \) in normal form, \( \text{Conv}_1(\text{Conv}_2(S)) = S \).

Proof. \( S(y^p) = \{ x^r \mid x^r \leq y^p \} \cup \{ x^p \mid x^p \leq y^p \} \) (Conv1)
= \{ x^r \mid x^r \in S(y^p) \} \cup \{ x^p \mid S(x^p) \subseteq S(y^p) \} \) (Conv2)
= \{ x^r \mid x^r \in S(y^p) \} \cup \{ x^p \mid x^p \in S(y^p) \} \) (by (ii))
= \( S(y^p) \) (by (i))

The normal form of a scope-as-sets scope function can be computed as:

\[
\text{Norm}(S) = \text{Conv}_1(\text{Conv}_2(S))
\]

Lemma 8. For any scope-as-sets scope function \( S \), \( \text{Norm}(S) \) is in fact in normal form.

Proof. The first requirement—that the range of \( S \) only contains variables—is immediately fulfilled by \( \text{Conv}_1 \). The second requirement follows from the definitions of \( \text{Conv}_1 \) and \( \text{Conv}_2 \):
\[
x^p \in S(y^p) \text{ iff } x^p \leq y^p \text{ iff } S(x^p) \subseteq S(y^p)
\]

Putting a scope-as-sets scope function in normal form preserves its binding structure. Furthermore, once it is in normal form, converting it to scope-as-a-preorder (and back) have no effect.

Lemma 9. \( \text{Norm} \) preserves “in scope of” and “more specific than” (and thus all of the concepts of section 5.4).

Proof. Follows directly from lemma [5]

This concludes the demonstration that scope-as-a-preorder is simply a normalized version of scope-as-sets, and can effectively be used in its place. The axioms of scope-as-sets are basic enough that they ought to apply in basically any lexically-scoped setting; thus scope-as-a-preorder should too. We use Scope-as-a-preorder in this thesis, however, because it is more canonical (per normalization) and because it more closely aligns with the binding language described next.

5.4 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 \textit{scoping rules} that, given a term, determine a scoping preorder over its variables.

5.4.1 \textit{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., \texttt{Lambda} or \texttt{Bind}) should specify a \textit{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 \((\texttt{Lambda}\ x^d_1\ (\texttt{Plus}\ x^r_2\ 3))\). To find the bindings of this term, we must know the scoping rules for \texttt{Lambda} and \texttt{Plus}. A sensible rule for \texttt{Plus} is that a term \((\texttt{Plus}\ \alpha\ \beta)\) has preorder \(\alpha \leq (\texttt{Plus}\ \alpha\ \beta)\) and \(\beta \leq (\texttt{Plus}\ \alpha\ \beta)\), meaning that whatever a \texttt{Plus} term is in scope of, its children are too. For brevity, we will typically write \(\alpha \leq \texttt{Plus}\) and \(\beta \leq \texttt{Plus}\) instead. Likewise, a sensible rule for \texttt{Lambda} is that a term \((\texttt{Lambda}\ \alpha\ \beta)\) has preorder \((\beta \leq \alpha \leq \texttt{Lambda})\), meaning that whatever a \texttt{Lambda} term is in scope of, its children are too, and that \(\beta\) (its body) is in scope of \(\alpha\) (its declaration). Put together, and applied to the example term, these rules give that:

\[(\texttt{Lambda}\ x^d_1\ (\texttt{Plus}\ x^r_2\ 3))\]

has preorder:

\[
x^r_2, 3 \leq \texttt{Plus} \leq x^d_1 \leq \texttt{Lambda}
\]

Thus \(x^r_2 \mapsto x^d_1\) by definition [8], as it should be.

5.4.2 \textit{A Problem}

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

\[(\texttt{Lambda}\ x^d_1\ (\texttt{Let}\*\ (\texttt{Bind}\ x^d_2\ x^r_3\ \texttt{EndBinds})\ x^r_4))\]

What will these scoping rules look like? Whatever they are, they should cause \(x^d_2\) to shadow \(x^d_1\), \(x^r_3\) to be bound by \(x^d_1\), and \(x^r_4\) to be bound by \(x^d_2\). Formally, we should have:

\[
x^d_2 \mapsto x^d_1\text{ and }x^r_3 \mapsto x^d_1\text{ and }x^r_4 \mapsto x^d_2
\]

which implies that, at a minimum:

\[
x^d_2 \leq x^d_1\text{ and }x^r_3 \leq x^d_1\text{ and }x^r_4 \leq x^d_2
\]

This places a set of requirements on the scoping rules for \texttt{Lambda}, \texttt{Let}\*, and \texttt{Bind} (assuming per section [5.3.1] that scoping rules are as local as possible). For
instance, \( x_3^R \leq x_1^R \) can only be achieved if \( x_3^R \leq \text{Bind} \leq \text{Let*} \leq x_1^R \). Continuing this way gives the requirements (shown both pictorially and textually):

However, this puts \( x_3^R \) in scope of \( x_2^R \), and as a result, \( x_3^R \) will be bound by \( x_2^R \)!

The problem is that \text{Bind} is trying to provide \( x_2^R \), to make it available in the body of \text{Let*}, but in doing so it incidentally makes it available in the \text{Bind}'s definition (to \( x_3^R \)). 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.

5.4.3 The Solution

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

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

A. \texttt{bind i in j}: A term may make all declarations \textit{exported} by its \textit{i}th child be \textit{imported} by its \textit{j}th child.

B. \texttt{import i}: A term’s \textit{i}’th child may import its parent’s declarations. If so, it \textit{imports} the declarations \textit{imported} by its parent. (This is almost universal: declarations in scope at a node in an \textit{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. \texttt{export i}: A term’s \textit{i}’th child may export its declarations to its parent. If so, the term \textit{exports} child \textit{i}’s \textit{exports}.

D. \texttt{re-export}: A term may take the declarations it \textit{imported}, and \textit{export} them. (We do not know of any examples of this in the wild, but we offer it for completion.)
These four kinds of paths may be represented graphically, showing imports as ↑ and exports as ↓:

In a potentially confusing change, these port arrows are drawn in the opposite direction as the port arrows in the paper this chapter is based on. Why the change?

In the paper, imports were drawn as ↓ and exports as ↑. As a result, in the diagram the up arrow B connected two downward-pointing import arrows, and the down arrow C connected two upward-pointing export arrows. Eelco Visser pointed out how confusing this was, so we switched it.

Relatedly, why do our (solid/green) binding arrows go in the direction they do? Some considerations suggest that they should go from reference to declaration (which is what we settled on):

- They should go the intuitive way, and flow from the natural question to ask (where is this reference bound?) to its answer (here is the binding declaration).
- They should be consistent with related work, and go in the same direction as the arrows in Scope Graphs.

Other considerations suggest that they go in the opposite direction (from declaration to reference):

- They should go the intuitive way, and flow in the same direction as values do: first a value is bound to a declaration, and then later that value is propagated to its reference.
- They should be consistent with related work, and match the arrows that the DrRacket editor draws between references and declarations.

It was a difficult choice, and could reasonably have gone in the other direction.

With these new bindings in mind, the requirements for the example from the previous subsection become:
Under this new preorder, \( x_3^k \mapsto x_1^p \) and \( x_4^k \mapsto x_2^p \) as desired.

5.4.4 Scoping Rules: Uncompressed

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):

\[
d \ ::= \ \uparrow \text{(import)} \mid \downarrow \text{(export)}
\]

A port, then, pairs a term \( e \) with a sign:

\[
a, b, c \ ::= \ \uparrow e \mid \downarrow e \text{ (port)}
\]

A set of scope rules \( \Sigma \) gives a relation for each term constructor \( C \) that describes the scoping relationships between a term constructed with \( C \) and its subterms:

**Definition 12.** A set of scope rules \( \Sigma \) is a partial map from term constructors \( C \) of arity \( n \) to binary relations over \( \{1, \ldots, n, r^\uparrow, r^\downarrow\} \), such that:

- The relation is transitive.
- \( r^\uparrow \) is a least element (\( \not\exists a. (a, r^\uparrow) \in \Sigma[C] \))
- \( r^\downarrow \) is a greatest element (\( \not\exists a. (r^\downarrow, a) \in \Sigma[C] \))

Here \( i \) represents the \( i \)th child term, \( r^\uparrow \) represents the parent term’s exports, and \( r^\downarrow \) represents the parent term’s imports. We will call pairs in the relation (e.g., \( (1, r^\uparrow) \)) facts, and will denote them with their description in our binding language (so that \( (1, r^\downarrow) = \text{import } 1 \)). 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 [5.4.3](#). We will write \( e' \sqsubseteq e \) to mean that \( e' \) is a subterm of \( e \), and write \( a \sqsubseteq e \) to mean \( \exists e'. (a = \uparrow e' \text{ or } a = \downarrow e') \) and \( e' \sqsubseteq e \).
As an example of scope rules, the rules for Lambda are:

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

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 13.** The scoping of a full term $$e$$ under scoping rules $$\Sigma$$ is the set of judgments of the form $$\Sigma, e \vdash a \leq b$$ defined by the “Declarative Rules” and “Shared Rules” of fig. 6.

The judgments in the figure have the form $$\Sigma, e \vdash a \leq b$$, which means that “$$a \leq b$$ in term $$e$$ using scoping rules $$\Sigma$$”. A judgment is well formed when $$a, b \subseteq e$$.

(Later, we will also use judgments of the form $$\Sigma, p \vdash a \leq b$$; these are governed by identical rules, allowing each term $$e$$ to instead be a pattern $$p$$.)

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. S-Decl says that declarations 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 6 (Algorithmic Scope Checking).** The declarative and algorithmic scope checking rules (fig. 6) [with shared rules common to both] are equivalent.

Proof. We will show that the Algorithmic (SA-) and Declarative (SD-) rules are equivalent by giving translations in both directions (omitting the symbol $$\Sigma$$ throughout for brevity). figs. 9 and 12 give the translations. The translations make use of the fact that the scoping rule relations are transitive.

The conversion from SA to SD is straightforward. However, the conversion from SD to SA is more difficult, as it has to handle the many ways the SD-Trans rule can be used. It proceeds by recursively pushing SD-Trans toward the leaves of the derivation.

The following table shows which inference rules can occur above SD-Trans (and are thus included in figs. 9 and 11), and which are impossible (and thus not included):

<table>
<thead>
<tr>
<th>left \ right</th>
<th>re-export</th>
<th>export</th>
<th>import</th>
<th>bind</th>
<th>lift</th>
</tr>
</thead>
<tbody>
<tr>
<td>re-export</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>export</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>import</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>bind</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lift</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Declerative Rules

\[
\begin{align*}
\Sigma, e &\vdash a \leq b \\
\Sigma, e &\vdash b \leq c \\
\Sigma, e &\vdash a \leq c
\end{align*}
\]

SD-Trans

SD-Import

\[
\Sigma, (C e_1 ... e_n) \vdash \uparrow e_i \leq \uparrow (C e_1 ... e_n)
\]

SD-Export

\[
\Sigma, (C e_1 ... e_n) \vdash \downarrow (C e_1 ... e_n) \leq e_i
\]

SD-Bind

\[
\Sigma, (C e_1 ... e_n) \vdash \uparrow e_i \leq \downarrow e_j
\]

Shared Rules (Declarative & Algorithmic)

\[
\begin{align*}
\Sigma, e &\vdash \uparrow e \leq \uparrow e \\
\Sigma, e &\vdash \downarrow e \leq \downarrow e
\end{align*}
\]

S-Refl1

S-Refl2

\[
\Sigma, e_i \vdash a \leq b
\]

\[
\Sigma, (C e_1 ... e_n) \vdash a \leq b
\]

S-Lift

\[
\Sigma, x^B \vdash \downarrow x^B \leq \uparrow x^B
\]

S-Decl

\[
\Sigma, (C e_1 ... e_n) \vdash \downarrow (C e_1 ... e_n) \leq \uparrow (C e_1 ... e_n)
\]

S-ReExport

Algorithmic Rules

\[
\begin{align*}
\Sigma, e_i \vdash \uparrow e_i &\text{ import } i \in \Sigma[C] \\
\Sigma, (C e_1 ... e_n) \vdash a \leq \uparrow (C e_1 ... e_n)
\end{align*}
\]

SA-Import

\[
\begin{align*}
\Sigma, e_i \vdash \downarrow e_i &\text{ export } i \in \Sigma[C] \\
\Sigma, e_j \vdash \downarrow e_j \leq a
\end{align*}
\]

SA-Export

\[
\Sigma, (C e_1 ... e_n) \vdash \downarrow (C e_1 ... e_n) \leq a
\]

\[
\begin{align*}
\Sigma, e_i \vdash \uparrow e_i &\text{ bind } j \text{ in } i \in \Sigma[C] \\
\Sigma, e_j \vdash \downarrow e_j \leq b
\end{align*}
\]

SA-Bind

\[
\Sigma, (C e_1 ... e_n) \vdash a \leq b
\]

Figure 6: Scope Checking

These scope checking rules say how to find a preorder over all of the ports in a term. However, section 5.3 is based only on preorders over the variables in a term. 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.
This is obtained as the restriction of the entire preorder to variables, as captured by the following rule:

$\Sigma, e \vdash x \leq y \quad \text{S-Var} \quad \Sigma, e \vdash x_i \leq y_j$

The definitions for binding and shadowing (definitions 8 and 11) can then be expressed as inference rules:

$\Sigma, e \vdash x \mapsto y \quad \text{S-Bound} \quad x^p \in \min\{x^p_i | \Sigma, e \vdash x^r \leq x^p_i\}$

$\Sigma, e \vdash x^r \mapsto x^p_j \quad \text{S-Shadow} \quad x^p_j \in \min\{x^p_i | \Sigma, e \vdash x^p_i \leq x^p_j\}$

These definitions form a scope preorder:

**Lemma 10.** For any set of scoping rules $\Sigma$ and term $e$, the relation $\{(x_i, x_j) | \Sigma, e \vdash x_i \leq x_j\}$ is a scope preorder satisfying the requirements of definition 5.

**Proof sketch.** The relation is a preorder by the derivation rules S-Ref1, S-Ref2, and SD-Trans. We must also show that references are least. Suppose instead that $x_i \leq x^p_i$ for some $i \neq j$. Then $\uparrow x_i \leq \uparrow x^p_i$ (by S-Var), which is syntactically impossible to achieve by the declarative judgements.

### 5.4.5 Well-Boundedness

Definition 8 (on being bound) can be used to define $\alpha$-equivalence. Two terms are $\alpha$-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 $x^r \mapsto x^p$ 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 $(e \bowtie e')$ that checks that $e$ and $e'$ have the same shape and finds a bijection between their variable occurrences as a witness to this fact:

$\begin{align*}
x^p_i \bowtie y^p_j & = \{x^p_i \leftrightarrow y^p_j\} \\
x^r_i \bowtie y^p_j & = \{x^r_i \leftrightarrow y^p_j\} \\
const & \bowtie const = \emptyset \\
(C \ e_1 \ldots e_n) & \bowtie (C \ e'_1 \ldots e'_n) = \bigcup_{j \in 1..n} e_j \bowtie e'_j \\
e & \bowtie e' = \text{undefined otherwise}
\end{align*}$

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 14 (Conflicting Declarations). Two variable declarations $x_i^p$ and $x_j^p$ conflict in a term $e$ when:

$$\Sigma, e \vdash a \leq x_i^p \quad \Sigma, e \vdash a \leq x_j^p$$

\[\min_{x_i^p, x_j^p} \{x_i^p, x_j^p\} = \{x_i^p, x_j^p\}\]

\[\Sigma, e \vdash x_i^p \text{ conflicts } x_j^p\]

(If a variable reference is ambiguously bound (definition 10), then its bindings declarations must be in conflict.)

A term $e$ is well-bound with respect to scoping rules $\Sigma$ when every reference is bound by exactly one declaration, and there are no conflicting declarations:

$$\forall x^p \in e. \exists! x^p \in e. \Sigma, e \vdash x^p \mapsto x^p$$

$$\Sigma, e \vdash x_i^p \text{ conflicts } x_j^p$$

The definition of $\alpha$-equivalence with respect to the scoping rules $\Sigma$ is then:

Definition 15 ($\alpha$-equivalence).

$$\Sigma \vdash \text{wb } e \quad \Sigma \vdash \text{wb } e' \quad e \equiv e' = \psi$$

\[\forall x^p, x^p', \Sigma, e \vdash x^p \mapsto x^p' \iff \Sigma, e' \vdash \psi(x^p) \mapsto \psi(x^p)\]

\[\Sigma \vdash e =_{\alpha} e'\]

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

In section 5.5.5 we show a catalog of scoping rules that can be expressed in our binding language.

5.5 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 $\alpha$-equivalence. More precisely, given a core language with scoping rules $\Sigma_{\text{core}}$ and a desugaring $D$, our algorithm finds scoping rules $\Sigma_{\text{surf}}$ that preserves $\alpha$-equivalence (theorem 8), so that:

$$\Sigma_{\text{surf}} \vdash e =_{\alpha} e' \quad \text{implies} \quad \Sigma_{\text{core}} \vdash D(e) =_{\alpha} D(e')$$

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

The general algorithm for scope inference is given in fig. 7. The next three subsections explain our assumptions about desugaring, and then the algorithm.

5.5.1 Assumptions about Desugaring

To briefly recap, we assume that desugaring is given by a set of rewrite rules of the form \( p \Rightarrow p' \), where \( p \) and \( p' \) are patterns:

\[
\text{pattern } p \ ::= \ \alpha \quad \text{pattern variable} \\
| \ \text{value} \quad \text{primitive value} \\
| \ (C \ p_1 \ldots p_n) \quad \text{AST node} \\
| \ x^r_i \quad \text{variable reference} \\
| \ x^d_i \quad \text{variable declaration}
\]

Additionally, references and declarations in the \texttt{rhs} of a rule must be given fresh names during expansion. Finally, desugaring must obey the requirements of section 3.2.1 with one exception: the \texttt{lhs}s of rules need not be disjoint.

When desugaring, there may be more than one rewrite rule that applies to a given term. None of the results of this chapter depend on the order of the rewrites; even a non-deterministic desugaring is allowed. A more typical choice is to apply rules in outside-in order, as described in section 3.2 and as is done by Scheme-style \texttt{syntax-rules} macros \[42\].

In general, a rewrite will look like:

\[ p_0[p[e_1, \ldots, e_n]] \Rightarrow p_0[p'[e_1, \ldots, e_n]] \]

where \( p_0 \) and \( p \) are patterns, and \( p[e_1, \ldots, e_n] \) denotes replacing the \( n \) pattern variables of pattern \( p \) with terms \( e_1, \ldots, e_n \). (In section 5.2, \( p \) was called the \texttt{lhs}, and \( p' \) the \texttt{rhs}.) The outer pattern \( p_0 \) 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, \( p_0 \) might be \((\texttt{Lambda } x^p \ a)\), and \( x^r \) inside the pattern variable may be unbound without it.

5.5.2 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 \( p \Rightarrow p' \). It is important that this rewrite does not change the binding of any variable \textit{outside} of \( p \). To achieve this, it will suffice that the preorder on the boundary of \( p \) is the same as the preorder among the boundary of \( p' \). The boundary, here, is the set of pattern variables in \( p \), together with the
inferScope(Σ, \{p_i \Rightarrow p'_i\}_{i \in 1..n}) \triangleq \text{Let } \Sigma_{surf} = solve(\Sigma, \{\text{genConstrs}(p_i \Rightarrow p'_i)\}_{i \in 1..n})
checkScope(\Sigma_{surf}, \{p_i \Rightarrow p'_i\}_{i \in 1..n})
Return \Sigma_{surf}
genConstrs(p \Rightarrow p') \triangleq \{\text{genConstr}(a \leq b, p \Rightarrow p')\}_{a,b \in H}
where \(H = \text{pattern-vars}(p) \cup \text{pattern-vars}(p') \cup \{r\}\) and where
the omitted port signs on \(a\) and \(b\) are filled in as described in
definition 16

genConstr(a \leq b, p) \triangleq \text{genConj}(a \leq b, p) \text{ iff genConj}(a \leq b, p')
(This is a constraint of the form \(f_1 \land \ldots \land f_m \text{ iff } f'_1 \land \ldots \land f'_m\).)

genConj(a \leq b, p) \triangleq \text{Find a derivation } D \text{ of } \Sigma, p \vdash a \leq b \text{ where } 
\Sigma \text{ is unknown, using the Algorithmic Scope Checking rules (fig. 6). This can be accomplished since the rules are syntax-directed over } p.
Return the conjunction of the premises of \(D\).
(These premises will be facts of the form \((i, j) \in \Sigma[C]\) for some \(i, j,\) and \(C,\) and their conjunction is both necessary and sufficient for \(\Sigma, p \vdash a \leq b\) to hold.)
solve(\Sigma_{core}, \text{constraints}) \triangleq \text{Initialize } \Sigma_{surf} = \Sigma_{core}
Until fixpoint:
\begin{itemize}
  \item If a constraint has the form \(f_1 \land \ldots \land f_m \text{ iff } f'_1 \land \ldots \land f'_m\) and some \(f_i\) (or \(f'_i\)) is true in \(\Sigma_{surf}\):
    Delete \(f_i\) (or \(f'_i\)) from the constraint
  \item If a constraint has the form \(f_1 \land \ldots \land f_m \text{ iff true, or the form true iff } f_1 \land \ldots \land f_m\) (where \text{true} is the empty conjunction):
    Delete the constraint from \text{constraints}
    Add \(f_i\) to \(\Sigma_{surf}\) for each \(f_1, \ldots, f_m\)
    (maintaining transitive closure)
  \item If at any point \((i, j) \in \Sigma_{surf}[C],\) where \(C\) is in the core language but \((i, j) \notin \Sigma_{core}[C]\):
    ERROR: Reject this sugar
\end{itemize}
Return \(\Sigma_{surf}\)

checkScope(\Sigma, \{p_i \Rightarrow p'_i\}_{i \in 1..n}) \triangleq \text{For each rule } p \Rightarrow p':
  \begin{itemize}
    \item Check that if \(\Sigma, p \vdash a \leq b\) and \(a \in p'\) then \(b \in p'\)
      (otherwise ERROR)
    \item Check that each reference \(x^a \in p'\) is bound by a unique declaration \(x^a \in p'\)
  \end{itemize}

Figure 7: Scope Inference Algorithm
root (i.e., the whole term). For example, in $p_0[p[e_1, ..., e_n]]$, $a_i$ bounds $e_i$, and $p$ (the root) bounds $p_0$. In general, we will call this property scope-equivalence:

**Definition 16** (Scope-equivalence of patterns).

$\Sigma \vdash p \equiv p'$ means that $\forall a, b \in \{a_1, ..., a_n, r\}$.

$$\Sigma, p \vdash a \leq b \iff \Sigma, p' \vdash a \leq b$$

where $r$ (“root”) stands in for $p$ or $p'$, as appropriate, and omitted port signs are determined by what our binding language allows:

$$\begin{align*}
\Sigma, p \vdash a_i \leq a_j & \triangleq \Sigma, p \vdash \uparrow a_i \leq \downarrow a_j \\
\Sigma, p \vdash a_i \leq r & \triangleq \Sigma, p \vdash \uparrow a_i \leq \uparrow p \\
\Sigma, p \vdash r \leq a_j & \triangleq \Sigma, p \vdash \downarrow p \leq \downarrow a_j \\
\Sigma, p \vdash r \leq r & \triangleq \Sigma, p \vdash \downarrow p \leq \uparrow p
\end{align*}$$

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

**Definition 17** (Scope-preservation). A rewrite

$$p_0[p[e_1, ..., e_n]] \Rightarrow p_0[p'[e_1, ..., e_n]]$$

preserves scope relative to a set of scoping rules $\Sigma$ if $\forall a, b \subseteq p_0, e_1, ..., e_n$ (i.e., each of $a$ and $b$ lies in one of $p_0, e_1, ..., e_n$):

$$\Sigma, p_0[p[e_1, ..., e_n]] \vdash a \leq b \iff \Sigma, p_0[p'[e_1, ..., e_n]] \vdash a \leq b$$

**Lemma 11** (Scope-equivalent patterns preserve scope). If $\Sigma \vdash p \equiv p'$, then any rewrite $p_0[p[e_1, ..., e_n]] \Rightarrow p_0[p'[e_1, ..., e_n]]$ preserves scope.

**Proof.** We will prove the forward implication of the iff in scope-preservation; the reverse is symmetric. View the $(\leq)$ preorder as a directed graph. Our given is that there is a path from $a$ to $b$ in $p_0[p[e_1, ..., e_n]]$, where neither $a$ nor $b$ lies in $p$. Some subpaths of this path may traverse $p$; these subpaths are bounded by $\uparrow e_1, \downarrow e_1, ..., \uparrow e_n, \downarrow e_n, \uparrow p, \downarrow p$. The fact that $\Sigma \vdash p \equiv p'$ means that these subpaths can be converted to subpaths in $p'$, bounded instead by $\uparrow e_1, \downarrow e_1, ..., \uparrow e_n, \downarrow e_n, \uparrow p', \downarrow p'$. Replace these subpaths. Now the whole path goes from $a$ to $b$ in $p_0[p'[e_1, ..., e_n]]$. \qed

We can use scope-equivalence to turn a rewrite rule $p \Rightarrow p'$ 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:

$$f_1 \land f_2 \land ... f_n \iff f_1' \land f_2' \land ... f_m'$$
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 $\Sigma, p \vdash a \leq b$ hold iff the premises of the derivation $\Sigma, p' \vdash a \leq b$ hold. These derivations are guaranteed to be unique, and can found efficiently, because the algorithmic scope-checking rules (fig. 6) are syntax-directed.

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

$$(Let \ a \ \beta \ \delta) \Rightarrow (Apply \ (Lambda \ a \ \delta) \ \beta)$$

One of the necessary constraints says that:

$$\Sigma, (Let \ a \ \beta \ \delta) \vdash \uparrow a \leq \downarrow \beta$$

iff $$\Sigma, (Apply \ (Lambda \ a \ \delta) \ \beta) \vdash \uparrow a \leq \downarrow \beta$$

Each side of this “iff” has a unique derivation using the algorithmic scope-checking rules (fig. 6). Replacing each side with the premises of its derivation yields the constraint:

bind 2 in 1 $\in \Sigma [Let]$ iff bind 2 in 1 $\in \Sigma [App] \land import 1 \in \Sigma [Lam]$

Since the boundary has size four ($a$, $\beta$, $\delta$, and $r$), continuing this way leads to a total of $4^2 = 16$ constraints:

bind 1 in 1 $\in \Sigma [Let]$ iff bind 1 in 1 $\in \Sigma [Lam]$
bind 2 in 1 $\in \Sigma [Let]$ iff bind 2 in 1 $\in \Sigma [App] \land import 1 \in \Sigma [Lam]$
bind 3 in 1 $\in \Sigma [Let]$ iff bind 2 in 1 $\in \Sigma [Lam]$
import 1 $\in \Sigma [Let]$ iff import 1 $\in \Sigma [App] \land import 1 \in \Sigma [Lam]$
bind 1 in 2 $\in \Sigma [Let]$ iff bind 1 in 2 $\in \Sigma [App] \land export 1 \in \Sigma [Lam]$
bind 2 in 2 $\in \Sigma [Let]$ iff bind 2 in 2 $\in \Sigma [App]$
bind 3 in 2 $\in \Sigma [Let]$ iff bind 1 in 2 $\in \Sigma [App] \land export 2 \in \Sigma [Lam]$
import 2 $\in \Sigma [Let]$ iff import 2 $\in \Sigma [App]$
bind 1 in 3 $\in \Sigma [Let]$ iff bind 1 in 2 $\in \Sigma [Lam]$
bind 2 in 3 $\in \Sigma [Let]$ iff bind 2 in 1 $\in \Sigma [App] \land export 2 \in \Sigma [Lam]$
bind 3 in 3 $\in \Sigma [Let]$ iff bind 2 in 2 $\in \Sigma [Lam]$
import 3 $\in \Sigma [Let]$ iff import 1 $\in \Sigma [App] \land import 2 \in \Sigma [Lam]$
export 1 $\in \Sigma [Let]$ iff export 1 $\in \Sigma [App] \land export 1 \in \Sigma [Lam]$
export 2 $\in \Sigma [Let]$ iff export 2 $\in \Sigma [App]$
export 3 $\in \Sigma [Let]$ iff export 1 $\in \Sigma [App] \land export 2 \in \Sigma [Lam]$
re-export $\in \Sigma [Let]$ iff re-export $\in \Sigma [App]$

We have just described how to generate constraints—covering the gen functions in fig. 7—and the previous lemma shows that the constraints generated this way capture our aim in scope inference. We now turn to solving these constraints.
5.5.3 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):

\[(\text{LambdaFF Flip } a_1 a_2) \Rightarrow (\text{Lambda } a_1 a_2)\]

\[(\text{LambdaFF Flop } a_1 a_2) \Rightarrow (\text{Lambda } a_2 a_1)\]

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 ∈ Σ[LambdaFF], and from the second rule, we can learn that bind 3 in 2 ∈ Σ[LambdaFF]. Applying either of these facts to the other rule gives that bind 2 in 1 ∈ Σ[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 /∈ Σ[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.

5.5.4 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 \textit{checkScope} in fig. 7

- Any references introduced on the \texttt{rhs} of a sugar must be bound. For instance, a sugar could not simply expand to \texttt{x^k}, because that would be unbound.

- A sugar cannot delete a pattern variable that might contain a bound declaration. For instance, it could not rewrite \texttt{(lambda a b)} to \texttt{b}, because \texttt{b} might contain a reference bound by a declaration in \texttt{a}. In general, if a sugar deletes any pattern variable, then it must also delete all smaller pattern variables (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 \textit{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 \texttt{x^i} and then uses \texttt{x^k} inside a sugar, and the sugar locally binds \texttt{x^j}, then \texttt{x^k} should not be bound by \texttt{x^i}. 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 ².

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 \textit{\alpha}-equivalent, then they will still be \textit{\alpha}-equivalent after a desugaring \texttt{D}:

\[
\Sigma_{surf} \vdash e =_\alpha e' \implies \Sigma_{core} \vdash D(e) =_\alpha D(e')
\]

(Recall from definition 15 that \textit{\alpha}-equivalence is parameterized by \Sigma. Therefore, in the above antecedent and consequent, \textit{\alpha}-equivalence is respectively defined by \Sigma_{surf} and \Sigma_{core}.)

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

Most hygiene papers don’t mention this criterion for a simple reason: \texttt{=} is not defined on their surface language, so they cannot even state the requirement. Recent exceptions to this rule ³⁶⁻²⁻ 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 5.7 we will show that they do, so long as inference was successful and scopeCheck passed.

This covers the solve algorithm in fig. 7 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.

5.5.5 Discussion of the Hygiene Property

One may wonder how useful of a property preserving α-equivalence is. To put it strongly: what good is it for desugaring to preserve α-equivalence, when α-equivalence for the surface language was made up? In fact, there is a situation in which preserving α-equivalence is useless. Suppose that \( e_1 =_\alpha e_2 \) in the surface language was defined to mean that \( D(e_1) =_\alpha D(e_2) \) in the core language, where \( D \) is a naive, scope-unaware desugaring. Then desugaring will preserve α-equivalence, despite not being hygienic!

It is therefore crucial that our binding language cannot express this (rather insane) notion of surface-language α-equivalence. It is the weakness (i.e., sanity) of our binding language that leads to the strength of our hygiene property. While “sanity” is a somewhat subjective notion, we can at least show that our binding language passes one basic litmus test. Any set of scope rules in our binding language will lead to a notion of α-equivalence that is invariant under permuting variable names (that is, it is equivariant [29]):

**Lemma 12.** If \( \sigma \) is a permutation of variable names and \( \Sigma \vdash e =_\alpha e' \), then \( \Sigma \vdash \sigma(e) =_\alpha \sigma(e') \).

*Proof sketch.* A formula \( \phi(a_1, ..., a_n) \) is equivariant if for all arguments \( a_1, ..., a_n \), and for all permutations \( \sigma \) of variable names, \( \phi(a_1, ..., a_n) \iff \phi(\sigma(a_1), ..., \sigma(a_n)) \). We wish to show that α-equivalence in our binding language is equivariant. The only parts of α-equivalence (definition 15) that inspect variable names are the various rules that implicitly check that two variables have the same name (such as S-Decl and S-WB). Luckily, equality checking is equivariant. Furthermore, formulas that do not inspect or modify names are equivariant, and any composition of equivariant formulas is itself equivariant. Thus α-equivalence is equivariant.

The “insane” notion of α-equivalence above—that defines surface language α-equivalence in terms of a naive desugaring—does not respect variable permutation, and thus cannot be expressed in our binding language.
5.5.6  Correctness and Runtime

The inferScope algorithm correctly solves the constraints:

**Theorem 7** (Rewrites preserve scope).

Let \( \Sigma_{surf} = \text{inferScope}(\Sigma_{core}, \{ p_i \Rightarrow p_i' \}) \). Then any rewrite of the form \( p_0[p_i[e_1, ..., e_n]] \Rightarrow p_0[p_i'[e_1, ..., e_n]] \) will preserve scope. Furthermore, \( \Sigma_{surf} \) is least (it is contained in every other set of scoping rules that would be preserved).

**Proof.** By construction, the constraints generated by inferScope ensure that \( \Sigma_{surf} \vdash p_i \bowtie p_i' \) for each \( i \). By lemma 11 this implies that any rewrite \( p_0[p_i[e_1, ..., e_n]] \Rightarrow p_0[p_i'[e_1, ..., e_n]] \) will preserve scope. Thus it suffices to show that solve does, in fact, find the least solution to the constraints given \( \Sigma_{core} \).

solve works with scope signatures \( \Sigma_{core} \) and \( \Sigma_{surf} \), and a set of constraints \( C \):

\[
F_1' \land F_2' \land ... \land F_n' \land F_1'' \land F_2'' \land ... \land F_n''
\]

At any point during the evaluation of solve, let the meaning \( \phi \) of \( \Sigma_{core}, \Sigma_{surf} \), and Cs be:

\[
\bigwedge_{F \in \Sigma_{surf}} F \land \bigwedge_{F \in \Sigma_{core}} F \land \bigwedge_{F \in \Sigma_{core}} \neg F \land \bigwedge_{i} \left( \bigwedge_{j} F_{ij}' \iff \bigwedge_{k} F_{ik}'' \right)
\]

That is,

1. Every fact in \( \Sigma_{surf} \) is true
2. Every fact in \( \Sigma_{core} \) is true
3. Every fact in the core language but not in \( \Sigma_{core} \) is false (i.e., we assume that \( \Sigma_{core} \) lists all scoping rules for the core language)
4. The constraints hold.

Upon initialization during solve, \( \phi \) follows from \( \Sigma_{core} \) and \( C \). Furthermore, every step of solve maintains \( \phi \). There are three kinds of steps to consider, and each follows from a logical equivalence:

- “If a fact \( F \) in a constraint is in \( \Sigma_{surf} \): Delete \( F \) from the constraint.”
  \[
  F \land (F \land F_2 \land ... \iff F_1' \land F_2' \land ...) \equiv F \land (F_2 \land ... \iff F_1' \land F_2' \land ...)
  \]

- “If one side of a constraint is empty: Delete the constraint; Add the other side to \( \Sigma_{surf} \) (maintaining trans. closure).”
  \[
  (\text{true} \iff F_1 \land ... \land F_n) \equiv F_1 \land ... \land F_n
  \]

- “If any fact \( F \) in \( \Sigma_{surf} \) is in the complement of \( \Sigma_{core} \): ERROR”
  \[
  F \land \neg F \equiv \text{false}
  \]
Finally, when solve halts, it is because the facts in \( C \) and \( \Sigma_{surf} \) are disjoint, and every constraint in \( C \) has at least one fact on each side. Therefore it is valid to obtain a minimal set of facts by setting every fact in \( \Sigma_{surf} \) to \textit{false}; doing so will satisfy \( C \) by making both sides of every remaining constraint false. Thus we have shown that the final \( \Sigma_{surf} \):

\[
\bigwedge_{F \in \Sigma_{surf}} F \land \bigwedge_{F \in \Sigma_{surf}} \neg F
\]

is a minimal solution to the initial \( \Sigma_{core} \) and \( C \):

\[
\bigwedge_{F \in \Sigma_{core}} F \land \bigwedge_{F \in \Sigma_{surf}} \neg F \land \bigwedge_{i} \left( \bigwedge_{j} F'_{ij} \text{ iff } \bigwedge_{k} F''_{ik} \right)
\]

Thus the surface language scoping rules \( \Sigma_{surf} \) found by solve are valid and minimal, given the core language scoping rules \( \Sigma_{core} \) plus the constraints \( C \).

\[\square\]

**Corollary 1** (Desugaring preserves scope).

Let \( \Sigma_{surf} = \text{inferScope}(\Sigma_{core}, \{p_i \Rightarrow p'_i\}_{i \in 1..n}) \). Then desugaring with the rules \( \{p_i \Rightarrow p'_i\}_{i \in 1..n} \) will preserve scope.

\[\square\]

**Proof.** Induct on the number of rewrites performed.

Furthermore, scope inference runs in time \( O(\Sigma_{C \in surfarity}(C)^3) \):

**Lemma 13.** \( \text{inferScope}(\Sigma, C) \) runs in time \( O(\text{size}(C) + \Sigma_{C \in surfarity}(C)^3) \).

\[\square\]

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

\[\square\]

The cubic parameter is concerning, but not a problem in practice for a number of reasons. First, \( \text{arity}(C) \) tends to be small. Second, this algorithm is run offline, and once per language. Finally, as we discuss in section \[5.6\] in practice the running time is extremely small.
5.6 Implementation and Evaluation

We have implemented the scope inference algorithm. Beyond what is shown in this chapter, 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 pattern variable, where the pattern variable contains a declaration and the copy is meant to be a reference of the same name. The implementation is available at https://github.com/brownplt/scope-graph.

Besides the examples shown earlier, 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 [42]: 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` constructors and used those to express the equivalent looping.) `letrec` required one further adjustment to successfully infer scope. 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 18ms on a generic desktop, of which 8ms 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.

5.6.1 Case Study: Pyret for Expressions

Consider the “for expressions” of the Pyret language:

---

1 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.
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:

```
In the textual representation of the scoping rules, we give names to a node’s children. Formally, these should be indices.
```

```
5.6.2 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 [61 section 3.11]: “List comprehensions satisfy these identities, which may be used as a translation into the kernel:”

```
[ 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 <- 1, 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 qualifier, and \( Q \) over sequences of qualifiers.”

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

\[
[n \mid n \leftarrow [1..], \text{let } d = \text{divisors } n, \text{sum } d == n]
\]

Our system successfully infers the scope of these sugars. We will describe them one at a time. First, list comprehensions \([e \mid Q]\) 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 \( e \):

\[
\begin{align*}
\uparrow e \downarrow & \quad \text{import } e \in \Sigma[\text{ListComprehension}] \\
\uparrow Q \downarrow & \quad \text{import } Q \in \Sigma[\text{ListComprehension}] \\
\quad & \quad \text{bind } Q \text{ in } e \in \Sigma[\text{ListComprehension}]
\end{align*}
\]

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:

\[
\begin{align*}
\uparrow b, Q \downarrow & \quad \text{import } b \in \Sigma[\text{LC_Guard}] \\
\uparrow b \downarrow & \quad \text{import } Q \in \Sigma[\text{LC_Guard}] \\
\quad & \quad \text{export } Q \in \Sigma[\text{LC_Guard}]
\end{align*}
\]

A generator expression \( p \leftarrow l, Q \) binds elements of list \( l \) to pattern \( p \). \( p \) is bound in \( Q \), and the declarations of both \( p \) and \( Q \) are exported:

\[
\begin{align*}
\uparrow p \leftarrow l, Q \downarrow & \quad \text{import } p \in \Sigma[\text{LC_Generator}] \\
\uparrow p \downarrow & \quad \text{import } l \in \Sigma[\text{LC_Generator}] \\
\uparrow l \downarrow & \quad \text{import } Q \in \Sigma[\text{LC_Generator}] \\
\quad & \quad \text{bind } p \text{ in } Q \in \Sigma[\text{LC_Generator}] \\
\quad & \quad \text{export } p \in \Sigma[\text{LC_Generator}] \\
\quad & \quad \text{export } Q \in \Sigma[\text{LC_Generator}]
\end{align*}
\]

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

\[
\begin{align*}
\uparrow \text{let } \text{decls}, Q \downarrow & \quad \text{import } \text{decls} \in \Sigma[\text{LC_Let}] \\
\uparrow \text{decls} \downarrow & \quad \text{import } Q \in \Sigma[\text{LC_Let}] \\
\quad & \quad \text{bind } \text{decls} \text{ in } Q \in \Sigma[\text{LC_Let}] \\
\quad & \quad \text{export } \text{decls} \in \Sigma[\text{LC_Let}] \\
\quad & \quad \text{export } Q \in \Sigma[\text{LC_Let}]
\end{align*}
\]
5.6.3 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 \((\text{let } (x \text{ val}) \ldots) \text{ body}\), and binds each declaration \(x\) to the corresponding \(\text{val}\) in body. The scope of this variant can be inferred similarly to how we inferred the scope of \(\text{let}\) in section 5.2.

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

\[
(\text{define } (\text{reverse list})
(\text{let rev } ([\text{unreversed list}] [\text{reversed empty}])
(\text{if } (\text{empty? unreversed})
  \text{reversed}
  (\text{rev } (\text{cdr unreversed})
    (\text{cons } (\text{car unreversed}) \text{ reversed})))))
\]

Named-let desugars by the rule:

\[
(\text{define-syntax-rule}
  ;; The named let sugar:
  (\text{let proc-id } ([\text{arg-id init-expr}] \ldots) \text{ body})
  ;; Desugars into:
  (\text{letrec } ([\text{proc-id } (\text{lambda } (\text{arg-id} \ldots) \text{ body})])
    (\text{proc-id init-expr} \ldots)))
\]

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

\[
e := (\text{Let } x^0 \ b \ e) \quad \text{“Named-let: bind initial values } b\text{ and recursive function } x^0\text{ in } e\”
| \ldots
b := (\text{Bind } x^0 \ e \ b) \quad \text{“Bind } x^0\text{ to } e,\text{ and bind } b\”
| \text{EndBinds} \quad \text{“No more bindings”}
\]

Translating the desugaring to use this grammar, our system correctly infers the binding structure:
While this correctly reflects the scoping of named-let, observe that it permits the let-bindings to shadow the function name. This follows because \((\text{arg-id} \ldots)\) can shadow \(\text{proc-id}\) 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.

5.6.4 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.

\[
\text{(do ((sum 0)}
  (i 0 (+ 1 i))
  ((= i 3) (display "The sum is: ") (display sum) (newline))
  (set! sum (+ sum (string->number (read-line)\)))))
\]

In general, do binds a list of variables \([\text{sum} \text{and i}]\) to initial values \([0 \text{and 0}]\), and then repeatedly evaluates the body of the loop \([\text{set! sum ...}]\) 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 \([\text{display "The sum is: "} (\text{display sum}) \text{newline}]\).

The desugaring of do is given by \([42]\) derived forms:

\[
\text{(define-syntax do (syntax-rules (}
  ((do ((var init step ...)) ...)
    (test expr ...))
\)
\]
We will focus on the scope of the binding list, as scope inference fails on it. Its correct scope is:

```
import var ∈ Σ[DoBind]
import init ∈ Σ[DoBind]
import step ∈ Σ[DoBind]
import binds ∈ Σ[DoBind]
export var ∈ Σ[DoBind]
export binds ∈ Σ[DoBind]
bind var in step ∈ Σ[DoBind]
bind binds in step ∈ Σ[DoBind]
```

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.

---

2 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), i in j for all i and j? How does scope inference handle these cases, while still being correct, fast, and hygienic?
5.6.5 Catalog of Scoping Rules (Extended)

In fig. 8 we demonstrate the expressiveness of our scoping rule language by showcasing a small catalog of scoping rules. They are shown diagrammatically. Some arrows have been omitted when they are implied by transitivity (e.g., the arrow between Lambda and its (body) is omitted). The names written for the children of each term (such as “(arg)” and “X^D”) are only expository, but meant to suggest a grammar the language might follow.

Single-binding Lambda and Let shows scoping rules for Lambda and Let when they have only a single binding position. Multi-argument Lambda extends this to functions of more than one argument. Notice that the scoping rule for Lambda is the same in each case: the only difference is that one is meant to be used with a single declaration while the other is meant to use a Param.

The next three rules describe the binding for Scheme-style Let, Let*, and Letrec let-bindings. Akin to Lambda, the rules for various Lets are identical; only their bindings differ.

Pattern Matching shows a rule for (nestable) pattern matching (or deconstructive assignment to) a pair. Its “left” and “right” children would typically be variable declarations, although they could themselves be nested pattern matches. Its “MatchPair” construct can be used in any of the binding sites of the other scoping constructs.

Finally, we show the (correctly) inferred scope for the three sugars in Pyret that bind values:

• For “loops” are actually syntactic sugar for the application of a higher-order function (like map).

• Pyret has both statement and expression forms of let. The statements desugar into let expressions, which then merge with adjacent lets to form a single binding block.

• Pyret function declarations desugar into lambdas with explicit recursive bindings.

5.7 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 D is hygienic when it preserves α-equivalence:

\[ \Sigma_{surf} \vdash e =_a e' \quad \text{implies} \quad \Sigma_{core} \vdash D(e) =_a D(e') \]
Single-binding
Lambda and Let

Multi-argument
Lambda

Multi-arm Let

Multi-arm Let* (Pyret)

Multi-arm Letrec

Pattern Matching

For Loops (Pyret)

Functions (Pyret)

Figure 8: Catalog of Scoping Rules (Arrows that follow from transitivity omitted)
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 $\alpha$-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 $D(e)$ are either New (fresh) or Copied from $e$, and variables in $e$ are either Used (if they were copied) or Unused otherwise. Formally, let $\phi$ be the mapping from copied variables in $D(e)$ to their sources in $e$, and:

\[
\begin{align*}
\text{Used} & \triangleq \text{range}(\phi) \\
\text{Unused} & \triangleq \text{vars}(e) - \text{range}(\phi) \\
\text{Copied} & \triangleq \text{domain}(\phi) \\
\text{New} & \triangleq \text{vars}(D(e)) - \text{domain}(\phi)
\end{align*}
\]

(where $\text{vars}(e)$ is the set of all variables in $e$).

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

Property 1.

\[
\begin{align*}
\forall x^R \in \text{Copied}. \quad \forall x^D \in \text{New}. & \quad \Sigma, D(e) \not\vdash x^R \mapsto x^D \\
\forall x^R \in \text{New}. \quad \forall x^D \in \text{Copied}. & \quad \Sigma, D(e) \not\vdash x^R \mapsto x^D
\end{align*}
\]

Second, $D$ must preserve binding structure among the variables it copies:

Property 2.

\[
\forall x^R, x^D \in \text{Copied}. \quad \Sigma, D(e) \vdash x^R \mapsto x^D \iff \Sigma, e \vdash \phi(x^R) \mapsto \phi(x^D)
\]

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

Property 3.

\[
\begin{align*}
\forall x^R \in \text{Used}. \quad \forall x^D \in \text{Unused}. & \quad \Sigma, e \not\vdash x^R \mapsto x^D \\
\forall x^R \in \text{New}. & \quad \Sigma, D(e) \vdash x^R \mapsto x^D
\end{align*}
\]

While these three properties are clearly necessary, it is by no means clear that they are sufficient to guarantee $\alpha$-equivalence preservation. However, the following theorem shows that they are both necessary and sufficient to ensure that $D$ preserves $\alpha$-equivalence:
Theorem 8 (Fundamental Hygiene Theorem). Let $\mathcal{D}$ be a desugaring function over terms with respect to scoping rules $\Sigma$ that obeys the assumptions of section 5.3.1. Then $\mathcal{D}$ respects $\alpha$-equivalence iff for every (well-bound) input term $e$ (numbering by property number):

1. $\forall x^R \in \text{Copied. } \forall x^\varphi \in \text{New. } \Sigma, \mathcal{D}(e) \not\models x^R \mapsto x^\varphi$
2. $\forall x^R \in \text{New. } \forall x^D \in \text{Copied. } \Sigma, \mathcal{D}(e) \not\models x^R \mapsto x^D$
3. $\forall x^R \in \text{Copied. } \forall x^\varphi \in \text{Copied. } \Sigma, \mathcal{D}(e) \models x^R \mapsto x^\varphi$

iff $\Sigma, e \vdash \phi(x^R) \mapsto \phi(x^D)$

where $\phi$ is the mapping from copied variables in $\mathcal{D}(e)$ to their sources in $e$.

Proof. $\mathcal{D}$ respects $\alpha$-equivalence iff (i) $\mathcal{D}$ maps well-bound terms to well-bound terms, and (ii) $\alpha$-renaming any declaration $x^D$ in any well-bound term $e$ does not change the binding structure of $\mathcal{D}(e)$.

Address parts (i) and (ii) in reverse order.

For part (ii), do a case analysis on $x^D$. In both cases, let $e$ be the input term.

1. $(x^D = \text{Unused})$ $\alpha$-varying $x^D$ renames both $x^D$ and also $x^R$ for every $x^R \mapsto x^D$ in $e$. Renaming $x^D$ itself is fine: since it doesn’t appear in $\mathcal{D}(e)$, it cannot change the binding structure of $\mathcal{D}(e)$. Now do case analysis on every such $x^R$:

   a) $(x^R \in \text{Unused})$ Similarly, this case is OK because $x^R$ does not appear in $\mathcal{D}(e)$.

   b) $(x^R \in \text{Used})$ This case is problematic: renaming $x^R$ in $e$ will rename $\text{image}(x^R)$ in $\mathcal{D}(e)$, which will cause its binding to change. Thus:

   $\forall x^R \in \text{Unused. } \forall x^D \in \text{Unused. } \Sigma, e \not\models x^R \mapsto x^D$ $\quad$ (1)

2. $(x^D = \text{Used})$ $\alpha$-varying $x^D$ renames both $x^D$ and $x^R$ for every $x^R \mapsto x^D$ in $e$. Renaming $x^D$ is problematic iff $\exists x^D$ s.t. $\Sigma, \mathcal{D}(e) \models x^R \mapsto x^D$ where $x^D \in \text{image}(x^D)$ but $x^R \notin \text{image}(x^R)$ for some $x^R \mapsto x^D$ (hence $x^R$’s binding will change when $x^D$ is renamed). Thus:

   If $\Sigma, \mathcal{D}(e) \models x^R \mapsto x^D$ and $\phi(x^D)$ exists, then $\phi(x^D)$ exists and $\phi(x^R) \mapsto \phi(x^D)$ $\quad$ (2)

That dealt with the direct effects of renaming $x^D$. Now consider the effects of renaming $x^R$ for some $\Sigma, e \models x^R \mapsto x^D$. Do case analysis on every such $x^R$:

   a) $(x^R \in \text{Unused})$ Renaming $x^R$ in $e$ is OK since it does not appear in $\mathcal{D}(e)$.
b) \((x^r \in \text{Used})\) Consider some \(x^r_i \in \text{image}(x^r)\). Let \(x^r_i \mapsto x^d_i \in \mathcal{D}(e)\) (it must be bound, since \(\mathcal{D}(e)\) is well-bound). For the binding of \(x^r_i\) to not change when \(x^d\) (and \(x^r\)) is renamed, \(x^d_i\) must be in \(\text{image}(x^d)\). Thus:

\[
\text{If } \sum, \mathcal{D}(e) \vdash x^r_i \mapsto x^d_i \text{ and } \phi(x^r_i) \text{ exists, then } \phi(x^d_i) \text{ exists, and } \phi(x^r_i) \mapsto \phi(x^d_i)
\]

For part (i), \(\mathcal{D}(e)\) must be well-bound. The only bindings not covered by the above cases are bindings from \text{New} to \text{New}. Thus:

\[
\forall x^r \in \text{New}. \exists! x^d \in \text{New}. x^r \mapsto x^d \in \mathcal{D}(e)
\]

This gives four equations that hold iff \(\mathcal{D}\) is hygienic. Equation (1) corresponds to the first part of the theorem’s Property (3). Equations (2) and (3) together correspond to Properties (1) and (2). Finally, equation (4) corresponds to the second part of Property (3).

We can now see that \text{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 \(7\), bindings between variables must not change. Finally, property 3 is exactly what \text{checkScope} checks.

### 5.8 Related Work

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 chapter is novel.

#### 5.8.1 Hygienic Expansion

The real goal of hygienic expansion is to preserve \(\alpha\)-equivalence: \(\alpha\)-renaming a program should not change its meaning. Typically, however, \(\alpha\)-equivalence is only defined for the core language. Thus, traditional approaches to hygiene have had to focus on avoiding specific issues like variable capture \([45]\). Recent work by Adams advances the theory by giving an algorithm-independent set of issues to avoid \([2]\). However, even this work lacks the ground truth of \(\alpha\)-equivalence preservation to base its claims on.

In contrast, Herman et al. advocate that sugar specify the binding structure of the constructs they introduce, and build a system that does so \([35]\). Stansifer et al. follow with a more powerful system called Romeo based on the same
approach [78]. (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 $\alpha$-equivalence without requiring scope annotations on sugars.

Erdweg et al. put forward an interesting alternative approach to hygiene with the name-fix algorithm [18]. 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 $\alpha$-equivalence, but for a weaker definition of $\alpha$-equivalence than ours that doesn’t include well-boundedness (thus allowing desugaring to produce unbound variables).

This chapter’s 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 $\alpha$-equivalence preservation (theorem 8), 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.

5.8.2 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 5.3) falls in this category. Second, “Binding Specification Languages” discusses ways in which scope can be determined for a given term. Our binding language (section 5.4) falls in this category.

**Modeling scope** Our description of scope-as-a-preorder is similar to the view expressed by Flatt in “Binding as Sets of Scopes” [24]. In fact, Flatt’s notion of scope can be expressed as a preorder, as we show in section 5.3.4.

Neron et al. describe scope graphs, which are based on a similar view, but are more general. For example, they include mechanisms for handling module scope, which gives them the ability to model both modules and also other constructs like objects and field lookup. While our scope-as-a-preorder binding language can be extended to handle modules, doing so breaks the transitivity assumption that we need to infer scope, so we have left it out of this chapter and consider this a problem for future work.
Our preorder-based binding specification language is novel, but similar in expressiveness to many others. It is perhaps most similar to Stansifer et al.’s Romeo [78]. The primary difference between the two is that Romeo has slightly more expressive power: given two declarations \( x_1 \) and \( x_2 \), it is possible in Romeo for \( x_1 \) to shadow \( x_2 \) in one part of a term, but \( x_2 \) to shadow \( x_1 \) in a different part of a term.\(^3\) 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. present a semantics engineering workbench called Ott, which includes a comparable binding specification language [74]. Like Romeo, Ott would allow two declarations to each shadow one another in different places. Furthermore, it gives additional power, 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. present a binding specification language called Unbound, which can be expressed using scope-as-a-preorder (and hence is no more expressive than it) [85]. They implement Unbound in Haskell, and give language-agnostic implementations of operations such as constructing and deconstructing terms, determining \( \alpha \)-equivalence, and performing substitution. In Unbound, binding is specified via a set of binding combinators. These binding combinators can be expressed as a preorder.\(^4\)

There are many more binding specification languages [1 71 47]. We have chosen what we believe to be a representative sample for comparison. We have shown that our binding specification language has a comparable level of expressiveness with most such systems, while simultaneously being simple enough to enable scope inference.

There is a close analogy between our ports and attributes in attribute grammars [44]; 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

\(^3\) In Romeo, this would be expressed using the \( \llbracket \) combinator, as \( \beta_1 \llbracket \beta_2 \) and \( \beta_2 \llbracket \beta_1 \).

\(^4\) The translation of Unbound to scope-as-a-preorder is as follows. \( \text{Name} \) constructs a declaration or reference, depending on whether it is a term or Pattern. Patterns \( P \) have scoping rules that state \( \text{export } i \in \Sigma[C] \) and \( \text{import } i \in \Sigma[C] \) for every \( i \). Terms \( e \) have scoping rules that state \( \text{import } i \in \Sigma[e] \) 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:

\[
\begin{align*}
\text{Bind } P & \quad \{ \text{bind } 1 \in 2 \in \Sigma[\text{Bind}] \} & \text{Embed } e & \quad \emptyset \\
\text{Rebind } P & \quad \{ \text{bind } 1 \in 2 \in \Sigma[\text{Rebind}] \} & \text{Rec } P & \quad \{ \text{bind } 1 \in 1 \in \Sigma[\text{Rec}] \}
\end{align*}
\]
and synthesized attributes that are allowed in attribute grammars. Most algorithms for evaluating attribute grammars disallow cycles, however, while our preorders allow them.

5.9 Discussion 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 5.3 and our binding language for specifying the scope of a programming language in section 5.4. The case studies in section 5.6 show that all of the aspects of this work are able to deal with many interesting scoping constructs from real languages.

There are a number of weaknesses in our binding language that it would be useful to ameliorate:

- Support for ellipses in sugar definitions would make writing sugars easier.
- Allowing named imports and exports—à la Ott [74]—would make sugars like `do inferable. It could also help with the fact that value bindings and type bindings often flow differently. For example, in Java field names are visible in subclasses while type parameters are not. With named imports, each syntactic construct could effectively have two scope rules—one for values and one for types—and they could be inferred separately.
- References are not always bound to a single declaration. For example, if a method is overloaded, then a reference to that method is bound to both method definitions, with the actual binding determined at runtime. In definition 10 we define a reference that is bound to multiple declarations to be ambiguously bound, and we consider this to be an error. It would be useful to be able to relax this restriction in certain circumstances, and infer during scope inference when it should be relaxed for a particular sugar.
- The ability to group several bindings together into a single namespace—à la module declarations in Scope Graphs [57]—is necessary for inferring scope for modules and for classes. This change is a relatively straightforward extension to our binding language, but is a research question when applied to scope inference.
Figure 9: Proof of theorem 6
\[ \text{SD } \rightarrow \text{SA} \]

\[ \begin{array}{ll}
\text{SD-Export} & \text{export } i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash (C e_1 \ldots e_n) \leq e_i & \Rightarrow \text{SA-Export} \quad \text{export } i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash (C e_1 \ldots e_n) \leq e_i & \\
\text{SD-Import} & \text{import } i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash \uparrow e_i \leq \uparrow (C e_1 \ldots e_n) & \Rightarrow \text{SA-Import} \quad \text{import } i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash \uparrow e_i & \leq \uparrow (C e_1 \ldots e_n) \\
\text{SD-Trans} & \text{bind } j \in i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash \uparrow e_i \leq e_j & \Rightarrow \text{SA-Bind} \quad \text{bind } j \in i \in \Sigma[C] \\
(C e_1 \ldots e_n) \vdash e_j & \leq e_i \\
\end{array} \]
<table>
<thead>
<tr>
<th>SD → SA (cont.)</th>
<th>SD-Trans</th>
<th>S-Lift</th>
<th>SD-Bind</th>
<th>intermediate 1</th>
<th>SD-Trans</th>
<th>S-Lift</th>
<th>SD-Bind</th>
<th>intermediate 1</th>
<th>SD-Trans</th>
<th>S-Lift</th>
<th>SD-Bind</th>
<th>intermediate 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD-Trans</td>
<td>(C_1 \ldots C_n) \models \mathcal{T}_x \models \mathcal{V}_x \models b \leq b \leq a \leq b \leq c</td>
<td>S-Lift</td>
<td>(C_1 \ldots C_n) \models \mathcal{T}_x \models \mathcal{V}_x \models b \leq b \leq a \leq b \leq c</td>
<td>SD-Trans</td>
<td>(C_1 \ldots C_n) \models \mathcal{T}_x \models \mathcal{V}_x \models b \leq b \leq a \leq b \leq c</td>
<td>S-Lift</td>
<td>(C_1 \ldots C_n) \models \mathcal{T}_x \models \mathcal{V}_x \models b \leq b \leq a \leq b \leq c</td>
<td>SD-Bind</td>
<td>(C_1 \ldots C_n) \models \mathcal{T}_x \models \mathcal{V}_x \models b \leq b \leq a \leq b \leq c</td>
<td>intermediate 1</td>
<td>SD-Trans</td>
<td>S-Lift</td>
</tr>
</tbody>
</table>

Figure 11: Proof of theorem 6 (continued)
\[ \begin{align*}
\text{SD} \rightarrow \text{SA (cont.)} \\
\text{S-Lift} & \quad \frac{e_i \vdash a \leq b}{(C \ e_1 \ldots e_n) \vdash a \leq b} \\
\text{SA-Import} & \quad \frac{e_i \vdash \uparrow e_j \quad \text{import } i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash b \leq \uparrow(C \ e_1 \ldots e_n)} \\
\text{SD-Trans} & \quad \frac{e_i \vdash a \leq b \quad e_j \vdash b \leq \uparrow e_j \quad \text{import } i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash a \leq \uparrow(C \ e_1 \ldots e_n)} \\
\text{SA-Bind} & \quad \frac{e_i \vdash \uparrow e_j \quad \text{bind } j \ in \ i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash a \leq b} \\
\text{SA-Import} & \quad \frac{e_j \vdash \uparrow e_j \quad \text{import } j \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash b \leq \uparrow(C \ e_1 \ldots e_n)} \\
\text{SD-Trans} & \quad \frac{e_i \vdash a \leq \uparrow e_j \quad \text{bind } j \ in \ i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash a \leq \uparrow(C \ e_1 \ldots e_n)} \\
\text{SA-Bind} & \quad \frac{e_j \vdash \downarrow e_j \quad \text{bind } j \ in \ i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash a \leq b} \\
\text{SA-Export} & \quad \frac{\text{export } i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash \downarrow(C \ e_1 \ldots e_n) \leq a} \\
\text{SD-Trans} & \quad \frac{\text{export } i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash \downarrow(C \ e_1 \ldots e_n) \leq b} \\
\text{SA-Export} & \quad \frac{\text{export } j \ in \ i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash \downarrow e_j \leq b} \\
\text{SA-Export} & \quad \frac{\text{export } j \ in \ i \in \Sigma[C]}{(C \ e_1 \ldots e_n) \vdash \downarrow e_j \leq b}
\end{align*} \]
RESUGARING TYPES

Type systems and syntactic sugar are both valuable to programmers, but sometimes at odds. While sugar is a valuable mechanism for implementing realistic languages, the expansion process obscures program source structure. As a result, type errors can reference terms the programmers did not write (and even constructs they do not know), baffling them. The language developer must also manually construct type rules for the sugars, to give a typed account of the surface language. In this chapter, we address these problems by presenting a process for automatically reconstructing type rules for the surface language using rules for the core. We have implemented this theory, and show several interesting case studies.

This chapter comes from work published in PLDI 2018 under the title Inferring Type Rules for Syntactic Sugar (co-authored with Shriram Krishnamurthi) [69].

6.1 INTRODUCTION

While both desugaring and type checking are valuable, they typically interact poorly. Type checking occurs either before or after desugaring, and there can be major problems with each.

Suppose type-checking occurs on the desugared code. This has the virtue of keeping the type-checker’s target language more tractable. However, errors are now going to be generated in terms of desugared code, and it is not always clear how to report these in terms of the surface language. This is further complicated when the code violates implicit type assumptions made by the sugar, which likely results in a confusing error message.

Alternatively, suppose we type-check surface code. This too is problematic. It turns syntactic sugar into a burden by forcing the type-checker to expand with the size of the surface language. This is especially bad in languages with macro-like facilities, because the macro author must now also know how to extend a type-checker. This destroys a valuable division of labor: macro authors may be experts in a domain but not in programming language theory. Furthermore, the
enlarged type-checker must respect desugaring: i.e., every program must type
in exactly the same way in the surface as it would have after desugaring.

INFERRING A SURFACE TYPE SYSTEM  We offer a way out of this dilemma.
Given typing rules for the core language, and syntactic sugar written as pattern-
based rules, we show how to infer type rules for the surface language.

Notice that this is not a complete solution to the problem: we provide type
rules, but not a full type checker with quality error messages. This could be
done automatically or manually. Automatically extending a type checker while
maintaining good error messages is (we believe) an open, and independently
valuable, problem. Alternatively, the type rules can (as usual) be added to the
type checker by hand.

Whichever method is used, these new rules can be added to the documenta-
tion for the language, providing a typed account of the surface. These rules are
also a useful diagnostic, enabling the author of the sugar, or an expert on the lan-
guage’s types, to confirm that the inferred typing rules are expected; when they
are not, these suggest a flaw in the desugaring. This diagnostic comes very early
in the pipeline: it relies only on the sugar definition, and so is available before a
sugar is ever used.

This approach depends crucially on a particular guarantee, which our system
will provide:

A surface program has a type in the inferred surface type system
iff its desugaring has that type in the core type system.

Thus, a well-typed program under the inferred surface rules will desugar into
a well-typed program under the original core rules. As a result, an ill-typed
program will always be caught in the surface type system, and an ill-typed sugar
will be rejected by our algorithm at definition time rather than having to wait until
it is used. Since the inferred type rules are guaranteed to be correct, they become
a valid documentation of the surface language’s type structure.

6.2 TYPE RESUGARING

Our overall aim is to be able to generate type judgments for the surface lan-
guage given desugaring rules and judgments for the core, i.e., to perform type
resugaring. Type resugaring, in which type rules are inferred through syntactic
sugar, should be distinguished from ordinary type inference, in which types are
inferred within a program. We wish to obtain type rules for the surface language
that are faithful to the core language type rules: type checking using resugared
type rules should produce the same result as first desugaring and then type
checking using the core type rules. Specifically, if $I_{\text{core}}$ are the core language type rules and $I_{\text{surf}}$ are the resugared surface type rules, then

**Goal 1.**

$$I_{\text{surf}} \vdash \Gamma \vdash e : t \text{ iff } I_{\text{core}} \vdash \Gamma \vdash D(e) : t$$

where $T \vdash J$ means that judgment $J$ is provable by inference rules $T$, and $D(e)$ means the desugaring of expression $e$.

Notice the assumption implicit in this equation: the right-hand-side says $t$, rather than $D(t)$. We are handling desugaring of expressions, but not of types. It is sometimes desirable to introduce a new type by way of translation into an existing type: for instance, introducing Booleans and implementing them in terms of Integers. We leave this more general problem—resugaring type rules, when types can contain sugars—to future work.

To see how type resugaring might proceed, let us work through an example. Take a simple and sugar, defined by:

$$\alpha \text{ and } \beta \Rightarrow \text{if } \alpha \text{ then } \beta \text{ else false}$$

Our goal is to construct a type rule for and that is faithful to the core language, meaning that (using goal 1):

$$I_{\text{surf}} \vdash \Gamma \vdash (\alpha \text{ and } \beta) : t \text{ iff } I_{\text{core}} \vdash \Gamma \vdash D(\alpha \text{ and } \beta) : t$$

Expanding out the sugar:

$$I_{\text{surf}} \vdash \Gamma \vdash (\alpha \text{ and } \beta) : t \text{ iff } I_{\text{core}} \vdash \Gamma \vdash D(\alpha) \text{ then } D(\beta) \text{ else false} : t$$

It is seemingly straightforward to obtain this property. We just have to add this inference rule to $I_{\text{surf}}$.

$$\frac{\Gamma \vdash (\text{if } D(\alpha) \text{ then } D(\beta) \text{ else false}) : t}{\Gamma \vdash (\alpha \text{ and } \beta) : t}$$

and perhaps also its converse:

$$\frac{\Gamma \vdash (\alpha \text{ and } \beta) : t}{\Gamma \vdash (\text{if } D(\alpha) \text{ then } D(\beta) \text{ else false}) : t}$$

The rule $\text{t-and}^\rightarrow$ can be read as “to prove that $(\alpha \text{ and } \beta)$ has type $t$ under type environment $\Gamma$ in the surface language, prove that its desugaring has type $t$ under $\Gamma$ in the core language”. This is useful because it provides a way to prove a type in the surface language by way of the core language type rules.

Its converse $\text{t-and}^\leftarrow$, however, is not helpful: there is no need to use the surface language when trying to prove a type in the core language. Furthermore,
\(t \text{-} \text{and}^+\) is actually redundant: since \(t \text{-} \text{and}^-\) is the only type rule mentioning \text{and}, \(t \text{-} \text{and}^+\) is admissible. Therefore, we only need \(t \text{-} \text{and}^-\).

In this particular case, we have added only the rule \(t \text{-} \text{and}^-\), but in general we would add one such rule for each sugar. This could be called the \textit{augmented} type system: it is the core language type system, plus one extra rule per sugar, such that we obtain a type system for the surface language.

Type checking in this augmented type system is akin to desugaring the program and type checking in the core language. For example, the program \texttt{true} and \texttt{false} has the type derivation:

\[
\frac{\vdash \text{true} : \text{Bool} \quad \vdash \text{false} : \text{Bool} \quad \vdash \text{false} : \text{Bool}}{\vdash (\text{if true then false else false}) : \text{Bool}}
\]

\[
\frac{\vdash \text{false} : \text{Bool}}{\vdash (\text{true and false}) : \text{Bool}}
\]

Since the extension type rules (like \(t \text{-} \text{and}^-\)) always succeed, any type errors will be found in the core language. For example, if the first argument to \texttt{and} was not a boolean, this will be discovered by the \(t \text{-} \text{if}\) rule, not by the \(t \text{-} \text{and}^-\) rule!

Thus, while the augmented type system technically obeys goal [1] it breaks the abstraction that ought to be provided by syntactic sugar. Type errors made in the surface language should be reported with respect to surface language constructs. This can be achieved with a second goal:

**Goal 2.** Type rules for surface constructs should not mention core constructs.

Let us see how we can accomplish this. The essential insight is that every type derivation of \texttt{and} will share a common form. It will always follow the template:

\[
\frac{D_\alpha \quad D_\beta}{\vdash \text{if } \alpha \text{ then } \beta \text{ else false} : \text{Bool}}
\]

\[
\frac{\vdash \alpha : \text{Bool} \quad \vdash \beta : \text{Bool}}{\vdash (\alpha \text{ and } \beta) : \text{Bool}}
\]

where the sub-derivations \(D_\alpha\) and \(D_\beta\) depend on \(\alpha\) and \(\beta\). Notice that the rest of the derivation is constant: every type-derivation of \texttt{and} has this form. Thus there is no reason to re-derive it every time we type-check. Instead, we can remove this “cruft” to obtain a simpler type rule for \texttt{and}:

\[
\frac{\vdash \alpha : \text{Bool} \quad \vdash \beta : \text{Bool}}{\vdash (\alpha \text{ and } \beta) : \text{Bool}}
\]

This type rule now satisfies our two goals, and is a valid and useful type rule for the surface language. Indeed, it hides the implementation of \texttt{and} and instead focuses just on its (expected) type structure.

The important step was determining the “template” derivation. We presented it above without fanfare, but how can it automatically be discovered? Let us look into this with a slightly more complex example, an \texttt{or} sugar:

\begin{itemize}
\item The \texttt{let} in the desugaring of \texttt{or} prevents the duplicate evaluation of \(\alpha\).
\end{itemize}
As before, we want to find a derivation for the sugar’s RHS (right-hand-side). That is, we should search for a derivation of the judgment:

\[ \Gamma \vdash (\text{let } x = \alpha \text{ in if } x \text{ then } x \text{ else } \beta) : t \]

We can begin by applying the core type rules, obtaining a partial derivation, shown at the top of fig. 13. However, the core type rules (unsurprisingly) cannot prove the judgments about pattern variables (marked with ?). Each pattern variable stands for an unknown surface term, so its derivation will vary between different uses of the or sugar. Since we do not know what type it will have, we will assign it a globally fresh type variable, using the rule t-premise:

\[ \text{t-premise } \Gamma \vdash \alpha : A \]

We write this rule with a dashed line because it is in a sense incomplete: it serves as a placeholder for a subderivation that would be filled in if the pattern variable were instantiated. Using this rule finishes the derivation, giving the bottom derivation in fig. 13. Notice that t-premise can be used to prove \( \Gamma \vdash \alpha : A \), because \( \alpha \) is a pattern variable, but cannot be used to prove \( \Gamma, x : A \vdash x : A \), because \( x \) is not a pattern variable.

As seen, pattern variables introduce type variables. Solving for these type variables in general requires unification. We therefore split the search for a derivation: first we find a potential derivation with equality constraints (as in fig. 13), then we solve these constraints (via an ordinary unification algorithm). Solving the constraints of fig. 13 gives the substitution \( \{ A = \text{Bool}, B = \text{Bool} \} \).

Finally, gathering the premises and conclusion of the derivation and applying the substitution to them produces the type rule for or:

\[ \text{t-or } \Gamma \vdash \alpha : \text{Bool} \quad \Gamma \vdash \beta : \text{Bool} \]

\[ \Gamma \vdash \alpha \text{ or } \beta : \text{Bool} \]
OUR OVERALL APPROACH  Putting all this together, we can describe our type resugaring algorithm. For each desugaring rule, such as the or sugar from above:

1. Construct a generic type judgment from the sugar’s RHS,
   e.g. \( \Gamma \vdash (\text{let } x = \alpha \text{ in if } x \text{ then } x \text{ else } \beta) : t \)

2. Search for a derivation of this judgment using the core language type rules plus the \texttt{t-premise} rule from above. Fail if no derivation, or if multiple derivations, are found. For example, this will find the derivation shown in fig. [13]

3. Gather the equality constraints from the derivation. Additionally, if multiple premises (i.e., judgments proved by the \texttt{t-premise} rule) are of the same expression, add equality constraints that these expressions have the same type. Solve the unification problem. (If there are any unconstrained variables, they become free variables in the type rule.)

   For example, in \texttt{or}, there are two equality constraints: \( A = \text{Bool} \) and \( A = B \). The \texttt{t-premise} rule is used only once for \( \alpha \) and once for \( \beta \), so no additional constraints are needed. The solution is \{ \( A = \text{Bool}, B = \text{Bool} \) \}.

4. Form a type rule whose premises are the judgments proved by \texttt{t-premise} from the derivation in step (2), and whose conclusion is a generic type judgment from the sugar’s LHS. Apply the unification from step (3). This is the resugared surface type rule.

We have implemented a prototype of this approach, called SweetT. SweetT is written in Racket [26] (racket-lang.org), and makes use of the semantics engineering tool Redex [19]. All of the examples in this chapter run in SweetT, albeit with a different, more parenthetical, syntax.

6.3 THEORY

In this section, we describe the assumptions that type resugaring will rely on, and then prove that it obeys goal \( \text{goal}^1 \) and goal \( \text{goal}^2 \) given these assumptions. Roughly speaking, these assumptions are:

- Desugaring rules must obey the basic “Well-Formedness Criteria” of section \[3.2.1\]

- Desugaring rules must be defined using pattern-based rules, and their LHSs must be disjoint (section \[6.3.1\]).

(This is the second “Restriction on Desugaring Rules” from section \[3.2.1\]. We relax the first restriction given there: pattern variables may be used more than once in a pattern.)
• The type system used to resugar must support pattern variables and partial derivations, and they must obey obvious laws (section 6.3.2).

• The core language type rules must be syntax directed (also section 6.3.2). This will fail, for instance, on a type system with non-algorithmic subtyping.

• Our implementation of SweetT must be correct (section 6.3.3). (As must Redex, which we use to find derivations.)

• Finally, SweetT’s unification algorithm must be able to handle the sugars given. Section 6.4.5 gives an example of extending it. Additionally, our implementation supports directly recursive sugars (as shown in section 6.4.2), but not sugars that are defined in terms of other sugars.

The rest of this section describes these assumptions in more detail. As a prelude, fig. [14] provides a guide to the notation we will use throughout the chapter.

6.3.1 Requirements on Desugaring

First, we require that desugaring rules be pattern-based. Each desugaring rule has a LHS and a RHS, which are terms $p$ that may contain pattern variables. Desugaring proceeds by recursively expanding these rules, replacing the LHS with the RHS. Formally:

\[
D(e) = e \\
D(value) = value \\
D(x) = x \\
D(\gamma \bullet p) = (D(\gamma)) \bullet (L[p]) \\
D(C e_1 \ldots e_n) = (C D(e_1) \ldots D(e_n)) \\
\]

if $p = (C p_1 \ldots p_n)$ and $C$ is in the surface language

if $C$ is in the core language

where $D(\cdot)$ is desugaring, $L$ represents the desugaring rules, $L[p]$ is the RHS of the desugaring rule whose LHS is $p$, and desugaring a substitution $\gamma$ means desugaring its expressions: $D(\{x \mapsto e, \ldots\}) = \{x \mapsto D(e), \ldots\}$.

Likewise, desugaring can be extended in the obvious way to desugar judgments and type environments:

\[
D(\Gamma \vdash p : t) = \Gamma \vdash D(p) : t \\
D(\{x \rightarrow p, \ldots\}) = \{x \rightarrow D(p), \ldots\} \\
\]

Unsurprisingly, substitution distributes over pattern-based desugaring:
Our approach relies on being able to use \( p \) in two different ways: (i) from one perspective, \( p \) is one side of a syntactic sugar rule, and any \( \alpha \) inside is a pattern variable; (ii) from the other perspective, \( p \) is an expression inside a type rule, in which \( \alpha \) is a metavariable. The convention of the first perspective is to call \( p \) as \( C \), but we choose instead to use \( p \) to emphasize the other perspective.

In addition, to the above notation, we will also write:

\[
\text{\( \Gamma \vdash J \) to mean that judgment \( J \) is provable under inference rules \( \Gamma \) (i.e., there is a derivation that proves \( J \)).}
\]

\[
\text{\( \top \vdash J_1 \ldots J_n \rightarrow J \) to mean that there is a derivation that proves \( J \) with unproven leaves \( J_i \).}
\]

\[
(\gamma \bullet p) \text{ to denote applying substitution } \gamma \text{ to pattern } p.
\]

**Figure 14: Notation explanation**

---

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e ::= )</td>
<td>( value ) (primitive value)</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>( p ::= )</td>
<td>( \alpha ) (pattern variable)</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>( t ::= )</td>
<td>( type ) (type)</td>
</tr>
<tr>
<td>( \Gamma ::= )</td>
<td>( \cdot \mid \Gamma, x : t ) (type environment)</td>
</tr>
<tr>
<td>( J ::= \top \vdash \alpha : t ) (type judgment)</td>
<td></td>
</tr>
<tr>
<td>( I ::= )</td>
<td>( J_1 \ldots J_n / I ) (inference rule)</td>
</tr>
<tr>
<td>( \Gamma ::= )</td>
<td>( I_1 \ldots I_n ) (set of inference rules)</td>
</tr>
<tr>
<td>( \gamma ::= )</td>
<td>( {\alpha \rightarrow e, \ldots} ) (substitution)</td>
</tr>
<tr>
<td>( \mathcal{R} ::= )</td>
<td>( {p \Rightarrow p', \ldots} ) (desugaring rules)</td>
</tr>
</tbody>
</table>
Lemma 14 (Distributivity of Substitution and Desugaring).

\[ \mathcal{D}(\gamma \cdot J) = \mathcal{D}(\gamma) \cdot \mathcal{D}(J) \]

Proof. Let \( J = \Gamma \vdash p : t \). By definition, \( \mathcal{D}(\Gamma \vdash p : t) = \Gamma \vdash \mathcal{D}(p) : t \). Induct on \( p \).

Base case: \( e \) is a primitive value:

\[ \mathcal{D}(\gamma \cdot \text{value}) = \text{value} = \mathcal{D}(\gamma) \cdot \text{value}. \]

Base case: \( e \) is a variable: likewise.

Base case: \( e \) is a pattern variable \( \alpha \):

\[ \mathcal{D}(\gamma \cdot \alpha) = \mathcal{D}(\gamma[\alpha]) = (\mathcal{D}(\gamma))[\alpha] = \mathcal{D}(\gamma) \cdot \alpha = \mathcal{D}(\gamma) \cdot \mathcal{D}(\alpha). \]

Inductive case: \( e \) is a compound term \( \{\alpha_1 \mapsto e_1, \ldots\} \cdot p \):

\[
\begin{align*}
\mathcal{D}(\gamma \cdot \{\{\alpha_1 \mapsto e_1, \ldots\} \cdot p\}) &= \mathcal{D}(\{\{\alpha_1 \mapsto (\gamma \cdot e_1), \ldots\} \cdot p\}) \quad \text{(substitution)} \\
&= \{\alpha_1 \mapsto \mathcal{D}(\gamma \cdot e_1), \ldots\} \cdot p' \quad \text{where } \mathcal{L}[p] = p' \\
&= \{\alpha_1 \mapsto (\mathcal{D}(\gamma) \cdot \mathcal{D}(e_1)), \ldots\} \cdot p' \quad \text{(I.H.)} \\
&= \mathcal{D}(\gamma) \cdot \mathcal{D}(\{\{\alpha_1 \mapsto e_1, \ldots\} \cdot p\}) \quad \text{(substitution)} \\
&= \mathcal{D}(\gamma) \cdot \mathcal{D}(\{\{\alpha_1 \mapsto e_1, \ldots\} \cdot p\}) \quad \text{(desugar)}
\end{align*}
\]

We also assume that the LHSs of each desugaring rule are disjoint, so that there is never any ambiguity as to which resugaring rule to apply (this is the last assumption of section 3.2.1). That is:

Assumption 1 (Unique Desugaring). For every pair \( p_1 \) and \( p_2 \) of sugar LHSs, there are no substitutions \( \gamma_1 \) and \( \gamma_2 \) such that \( \gamma_1 \cdot p_1 = \gamma_2 \cdot p_2 \).

This is everything we need of desugaring.

6.3.2 Requirements on the Type System

Let us now change focus to the type system. In the and example in section 6.2, we made implicit assumptions about the core type system. We stated that every type derivation of \( (\alpha \text{ and } \beta) \) must share a common template, and we implicitly assumed that this template could not depend on \( \alpha \) or on \( \beta \). This is certainly not true of every conceivable type system. Type resugaring will rely on three assumptions about the type system in order to make the approach we outlined work.

Before we describe these assumptions, notice that the type derivations found by resugaring (e.g., in fig. 13) contain pattern variables. Thus the type system used by resugaring is not exactly the language’s type system: it is an extension of the type system that handles pattern variables (and partial derivations, discussed
shortly). It is this extended type system we will be discussing in this section. With that said, we can state the assumptions.

First, we will assume that the type system supports pattern variables: it must be possible to search for type derivations of a judgment whose term contains pattern variables. Furthermore, a judgment with pattern variables must hold iff that judgment holds under all substitutions for those pattern variables:

**Assumption 2** (Substitution into Derivations). A derivation (possibly containing pattern variables) is provable iff it is provable under all substitutions:

\[
\Gamma \vdash J_1 \ldots J_n \rightarrow J \iff \forall \gamma. \Gamma \vdash \gamma \cdot J_1 \ldots \gamma \cdot J_n \rightarrow \gamma \cdot J
\]

Likewise for rules:

\[
\Gamma \vdash J_1 \ldots J_n / J \iff \forall \gamma. \Gamma \vdash \gamma \cdot J_1 \ldots \gamma \cdot J_n / \gamma \cdot J
\]

Next, we assume that the type system supports partial derivations that may contain unjustified judgments in their leaves, which we will call their premises. If a partial derivation is provable, and its premises are provable, then its conclusion must also be provable:

**Assumption 3** (Composition of Derivations). The composition of provable derivations is provable:

\[
\text{If } \Gamma \vdash J_1 \ldots J_n \rightarrow J \text{ and } \forall i. \Gamma \vdash J_i, \text{ then } \Gamma \vdash J.
\]

Finally, we would like the core type system to be deterministic in a particular way. Say that a judgment is abstract if it contains pattern variables, or concrete otherwise. We would like that if an abstract partial derivation \( J_1 \ldots J_n \rightarrow J \) applies to a concrete judgment \( \gamma \cdot J \) that can be proven, then the proof of \( \gamma \cdot J \) must use \( J_1 \ldots J_n \rightarrow J \), and thus prove as intermediate steps \( \gamma \cdot J_i \) for each \( i \in 1..n \).

Formally, we define determinism as:

**Definition 18** (Determinism). A set of inference rules \( \Gamma \) is deterministic when, for any concrete judgment \( \gamma \cdot J \):

\[
\text{If } \Gamma \vdash \gamma \cdot J \text{ and } \Gamma \vdash J_1 \ldots J_n \rightarrow J \text{, then } \Gamma \vdash \gamma \cdot J_i \text{ for each } i \in 1..n.
\]

Instead of assuming outright that the core language is deterministic, we can prove it from a more conservative assumption. We will assume that there is never any ambiguity as to which type rule applies to a concrete judgment \( J \), i.e., that the type system is syntax directed:

**Assumption 4** (Syntax Directedness). At most one type rule in \( \Gamma_{\text{core}} \) ever applies to a concrete judgment \( J \).
Under this assumption, the core language can be proven deterministic. This will be essential for our proof of goal 1.

**Lemma 15 (Determinism).** Suppose that at most one type rule in $T$ ever applies to a concrete judgment $J$. Then $T$ is deterministic.

**Proof.** Suppose that $T \vdash \gamma \cdot J$ and $T \vdash J_1 \ldots J_n \to J$. We aim to show that $T \vdash \gamma \cdot J_i$ for each $i \in 1..n$.

Induct on the derivation $T \vdash J_1 \ldots J_n \to J$. Let the bottommost step in the derivation be $T \vdash J'_1 \ldots J'_m / J$, and call this rule $R$. By assumption 2 (substitution), $T \vdash \gamma \cdot J'_1 \ldots \gamma \cdot J'_m / \gamma \cdot J$. Since, by assumption 4 (unique-rule), only one rule can apply to the judgment $\gamma \cdot J$, no rule other than $R$ may apply. Hence the derivation of $\gamma \cdot J$ must have $T \vdash \gamma \cdot J'_1 \ldots \gamma \cdot J'_m / \gamma \cdot J$ as the bottommost step. Thus for each $i \in 1..m$:

- $T \vdash \gamma \cdot J'_i$, and
- There is a subset $J_{i_1} \ldots J_{i_l}$ of $J_1 \ldots J_n$ such that $T \vdash J_{i_1} \ldots J_{i_l} \to J'_i$. Since each judgment $J_1 \ldots J_n$ must be used in the derivation $T \vdash J$, the union of these subsets must be the full set $J_1 \ldots J_n$.

For each $i \in 1..m$, by the inductive hypothesis, $T \vdash \gamma \cdot J_{i_1} \ldots \gamma \cdot J_{i_l}$. Since the union of these sets is $\gamma \cdot J_1 \ldots \gamma \cdot J_n$, we are done.

(Note that in the base case, $n = 0$, and the result is vacuously true.)

**Corollary 2 (Core Determinism).** If a core language $T_{core}$ obeys assumption 4 (unique-rule), then it is deterministic.

**Proof.** Follows from the lemma, together with assumption 4 (unique-rule).

### 6.3.3 Requirements on Resugar

Our final set of requirements is on the behavior of the type resugar algorithm. Thus it is essentially a specification for our implementation: SweetT is correct iff it obeys the requirements of this subsection.

Let us look at what it means to successfully resugar a desugaring rule $p_{LHS} \Rightarrow p_{RHS} \in \mathcal{L}$. Resugarin will search for a partial derivation of the sugar's RHS:

$$\overline{T_{core}} \vdash J_1 \ldots J_n \to J_{RHS}$$

where $J_1 \ldots J_n$ are provable using the $t$-premise rule and $J_{RHS}$ has the form $J_{RHS} = \Gamma \vdash p_{RHS} : t$. (Our implementation uses Redex's build-derivations function to perform this search.) If such a derivation is found, and is unique, then we will write:

$$\text{resugar}(\overline{T_{core}}, p_{LHS} \Rightarrow p_{RHS}) = J_1 \ldots J_n / J_{LHS}$$
where $J_{\text{LHS}} = \Gamma \vdash p_{\text{LHS}} : t$, and we will add the type rule $J_1 \ldots J_n / J_{\text{LHS}}$ to $\bar{I}_{\text{surf}}$. Therefore:

**Assumption 5 (Resugaring).** Suppose that
\[
\text{resugar}(\bar{I}_{\text{core}}, p_{\text{LHS}} \Rightarrow p_{\text{RHS}}) = J_1 \ldots J_n / (\Gamma \vdash p_{\text{LHS}} : t).
\]
Then:
\[
\bar{I}_{\text{core}} \vdash J_1 \ldots J_n \rightarrow D(\Gamma \vdash p_{\text{LHS}} : t)
\]
This is the correctness criterion for resugaring.

For the upcoming proof, we will also need that the surface language be deterministic in the sense of definition 18. This is provable using assumption 1 (unique-sugar):

**Lemma 16 (Surface Determinism).** If resugaring succeeds, then $I_{\text{surf}}$ is deterministic. Repeating the definition of determinism, this means that:

If $I_{\text{surf}} \vdash \gamma \cdot J$ and $I_{\text{surf}} \vdash J_1 \ldots J_n \rightarrow J$, then $I_{\text{surf}} \vdash \gamma \cdot J_i$ for each $i \in 1..n$.

**Proof.** To start, we will show that at most one resugared type rule may apply to a concrete judgment $J$. Suppose, for the sake of contradiction, that two distinct rules apply, with conclusions $J_1$ and $J_2$. Let the expressions in $J$, $J_1$, and $J_2$ be $e$, $e_1$, and $e_2$ respectively. Since both rules can be applied to $J$, there must be substitutions $\gamma_1$ and $\gamma_2$ such that $\gamma_1 \cdot J_1 = \gamma_2 \cdot J_2 = J$. Thus $\gamma_1 \cdot e_1 = \gamma_2 \cdot e_2 = e$. However, this contradicts assumption 1 (unique-sugar). Thus at most one type rule in $I_{\text{surf}}$ may apply to a concrete judgment.

Then, by lemma 15, $I_{\text{surf}}$ is deterministic.

### 6.3.4 Main Theorem

We now prove that—given the requirements of this section—type resugaring obeys goal 1. This theorem applies to the core of our approach, but ignores some details for simplicity: for example, it ignores the fact that type rules may have side conditions, and it assumes by fiat that our derivation search process works correctly (see assumption 5).

**Theorem 9.** Grant assumptions 1–5 from this section, let $L = p_{\text{LHS}} \Rightarrow_{\text{RHS}} \ldots$, and suppose that
\[
\text{resugar}(\bar{I}_{\text{core}}, p_{\text{LHS}} \Rightarrow_{\text{RHS}}) = \ldots = \bar{I}_{\text{surf}}.
\]
Then for all surface type judgments $J_{\text{surf}}$:
\[
\bar{I}_{\text{surf}} \vdash J_{\text{surf}} \iff \bar{I}_{\text{core}} \vdash D(J_{\text{surf}})
\]

**Proof.** Given in fig. 15

Furthermore, resugaring obeys goal 2 essentially by construction:
Proof. Split on the “iff”.

Forward implication (“soundness”): Induct on the derivation proving that \( \text{surf} \vdash I_s\). Let \( I_1 \ldots I_n / I_0 = \text{resugar}(\text{core} \ldots) \) be the rule in \( \text{surf} \) used to prove \( I_s\), and let \( \gamma \) be the substitution such that \( I_s = \gamma \cdot I_0 \). Then:

\[
\begin{align*}
\text{surf} & \vdash I_s \quad \text{assumption} \\
\text{surf} & \vdash \gamma \cdot I_0 \quad \text{equality} \\
\text{core} & \vdash \gamma \cdot I_i \quad \text{by lemma}^{16} \text{(surface determinism)} \\
\text{surf} & \vdash \text{D}(\gamma \cdot I_i) \quad \text{inductive hypothesis}
\end{align*}
\]

Also:

\[
\begin{align*}
\text{core} & \vdash I_1 \ldots I_n / \text{D}(I_0) \quad \text{by assumption}^{5} \text{(resugaring)} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_1 \ldots \text{D}(\gamma) \cdot I_n / \text{D}(\gamma) \cdot \text{D}(I_0) \quad \text{by assumption}^{2} \text{(substitution)} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_1 \ldots \text{D}(\gamma) \cdot I_n / \text{D}(\gamma) \cdot I_0 \quad \text{by lemma}^{14} \text{(distributivity)} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_1 \ldots \text{D}(\gamma) \cdot I_n / \text{D}(\text{surf}) \quad \text{equality}
\end{align*}
\]

Thus \( \text{surf} \vdash \text{D}(\text{surf}) \) by assumption\(^3\) (composition).

Reverse implication (“completeness”): Induct on the derivation proving that \( \text{core} \vdash \text{D}(\text{surf}) \). Let \( I_1 \ldots I_n / I_0 = \text{resugar}(\text{core} \ldots) \) be the rule in \( \text{surf} \) for the (outermost) sugar in \( I_s\)’s expression, and let \( \gamma \) be the substitution such that \( I_s = \gamma \cdot I_0 \). Then:

\[
\begin{align*}
\text{core} & \vdash \text{D}(\text{surf}) \quad \text{assumption} \\
\text{core} & \vdash \gamma \cdot I_0 \quad \text{equality} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_0 \quad \text{by lemma}^{14} \text{(distributivity)} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_1 \ldots \text{D}(\gamma) \cdot I_n \quad \text{by corollary}^{2} \text{(core determinism)} \\
\text{core} & \vdash \text{D}(\gamma) \cdot I_1 \ldots \text{D}(\gamma) \cdot I_n \quad \text{by lemma}^{14} \text{(distributivity)} \\
\text{surf} & \vdash \gamma \cdot I_1 \ldots \gamma \cdot I_n \quad \text{inductive hypothesis}
\end{align*}
\]

Also:

\[
\begin{align*}
\text{surf} & \vdash I_1 \ldots I_n / I_0 \quad \text{by assumption}^{5} \text{(resugaring)} \\
\text{surf} & \vdash \gamma \cdot I_1 \ldots \gamma \cdot I_n / \gamma \cdot I_0 \quad \text{by assumption}^{2} \text{(substitution)} \\
\text{surf} & \vdash \gamma \cdot I_1 \ldots \gamma \cdot I_n / \gamma \cdot I_s \quad \text{equality}
\end{align*}
\]

Thus \( \text{surf} \vdash I_s \) by assumption\(^3\) (composition). \( \square \)

Figure 15: Proof of theorem\(^9\)
Lemma 17. Resugaring obeys goal 2: type rules for surface constructs never mention core constructs.

Proof. Let \( \text{resugar}(I_{\text{core}}, p_{\text{LHS}} \Rightarrow p_{\text{RHS}}) = J_1 \ldots J_n / J_{\text{LHS}} \) be any surface rule. We aim to show that \( J_1 \ldots J_n \) and \( J_{\text{LHS}} \) do not mention core constructs \( C \). By assumption 5 (resugaring), \( I_{\text{core}} \vdash J_1 \ldots J_n \rightarrow D(J_{\text{LHS}}) \), where \( J_1 \ldots J_n \) are all provable using t-premise. We gave the t-premise rule in section 6.2 and generalize it in sections 6.4.3 and section 6.4.4. However, in all of its versions, the judgment must be over a surface term. Thus \( J_1 \ldots J_n \) do not mention core constructs.

Finally, the expression in \( J_{\text{LHS}} \) is the LHS of a desugaring rule, and is thus by definition a surface term. Therefore, given our assumptions listed in this section, resugaring obeys goal 2. \( \square \)

6.4 Desugaring Features

There are several important features of desugaring that make the above story more interesting. We describe them in this section.

6.4.1 Calculating Types

Consider the desugaring of \( \text{let} \) into the application of a lambda:

\[
\text{let } x = a \text{ in } \beta \Rightarrow (\lambda x : ? \cdot \beta)(a)
\]

What is the missing type? It needs to match the type of \( a \), but there is no way to express this using the kind of desugaring rules we have presented so far. We therefore extend the desugaring language with a feature called calc-type. In this example, it can be used as follows:

\[
\text{let } x = a \text{ in } \beta \Rightarrow \text{calc-type } a \text{ as } X \text{ in } (\lambda x : X \cdot \beta)(a)
\]

This binds the type variable \( X \) to the type of \( a \) in the rest of the desugaring.

In general, calc-type may be used in expression position on the RHS of a desugaring rule, and its meaning is that:

\[
\text{calc-type } p_1 \text{ as } t \text{ in } p_2
\]

desugars to \( p_2 \), in which the type \( t \) has been unified with the type of \( p_1 \), thus allowing the free type variables of \( t \) to be used in \( p_2 \). Notice that this requires \( \text{calc-type} \) can also be used to force a more specific surface type rule than would be inferred. For example, \( \text{calc-type } a \text{ as List<X> in } \ldots \) will lead to a surface type rule that enforces that \( a \) is a list. This is used in the Haskell list comprehension example of section 6.6.2 and in the or example of section 6.4.5.
desugaring and type checking to be interspersed. This is not surprising, since the desugaring of `let` involves determining a type.

This feature needs to be reflected in our type system. We do so with the type rule:

\[
\Gamma \vdash p_1 : t_1 \quad t_1 = t \quad \Gamma \vdash p_2 : t_2 \\
\Gamma \vdash (\text{calc-type } p_1 \text{ as } t \text{ in } p_2) : t_2
\]

With this type rule, we can find a type derivation for `let`, shown in fig. [16]. It leads to the type rule:

\[
\Gamma \vdash \alpha : A \\
\Gamma \vdash \alpha : A \\
\Gamma, x : A \vdash \beta : B \\
\Gamma \vdash \text{let } x = \alpha \text{ in } \beta : B
\]

Here we can see an advantage of type resugaring. As noted above, to type check `let` in the core, type checking and desugaring must be interspersed. However, to type check `let` in the surface, only this resugared rule is needed.

### 6.4.2 Recursive Sugars

Consider boolean guards in Haskell list comprehensions, which are defined by the desugaring rule (in Haskell’s syntax):

\[
[\alpha | \beta, \gamma] \Rightarrow \text{if } \beta \text{ then } [\alpha | \gamma] \text{ else } []
\]

This sugar, unlike those we have seen up to this point, is defined recursively: its `rhs` contains a list comprehension. Our resugaring algorithm, as described so far, will fail to find a type derivation for this sugar. It will get to the judgment `\[\alpha | \gamma\]` but lack any way to prove this judgment, because the `t-premise` rule does not match.

Our solution is to generalize the `t-premise` rule to allow any judgment about a surface term to be accepted as a premise. Notice that the term `[\alpha | \gamma]` is a surface term: when desugaring, pattern variables such as `\alpha` and `\gamma` will only ever be bound to surface terms, and thus they themselves should be considered part of the surface language. We therefore refine the `t-premise` rule as:
Furthermore, this is the most general rule we can make: goal 2 states that surface type rules must never mention core constructs, so t-premise can allow judgments over surface terms but nothing more.

6.4.3 Fresh Variables

Take the sugar const, which produces a constant function:

\[
\text{const } \alpha \Rightarrow \lambda x : \text{Unit. } \alpha
\]

It is important that \( x \) be given a fresh name, or else this sugar might accidentally capture a user-defined variable called \( x \) which is used in \( \alpha \). This is easy to add to desugaring: each desugaring rule will specify a set of “capturing” variables that are not freshly generated, and all other introduced variables will be given fresh names. (We use a capturing rather than fresh set to choose hygiene by default.)

This feature must also be reflected in the surface type system. First, let \( F \) be the set of introduced variables that are not marked as captured. We then add the type rule:

\[
\Gamma \vdash p : t \\
\frac{\text{\( x \) is a surface term \( x_1\ldots x_n \in F \)}}{\Gamma, x_1 : t_1\ldots x_n : t_n \vdash p : t}\]

to remove unnecessary fresh variables from the type environment, and by modifying t-premise to only work on judgments so limited:

\[
\frac{p \text{ is a surface term}}{\Gamma, x \not\in F \quad \forall x \in \Gamma. x \not\in F \quad \text{fresh } x'}{\Gamma \vdash p : x'}
\]

What exactly is t-fresh saying? It is a form of weakening, but with two extra restrictions. First, the variables being weakened are variables that will be given fresh names during desugaring. Second, the expression \( e \) is a surface term. Together, these imply that \( e \) cannot contain \( x_1\ldots x_n \), so it should be safe to remove them from \( \Gamma \). One way this could fail is if the language does not admit weakening, for example if it has a linear type system. We therefore assume that:

Assumption 6. The rule:

\[
\frac{x \not\in \Gamma \quad x \not\in e}{\Gamma, x : t' \vdash e : t}
\]

is admissible in the core type system.

This rule can be used to “reverse” any use of t-fresh, so if it is admissible then applying t-fresh greedily can never lead a derivation into a dead end.
6.4.4 **Globals**

Sugars may rely on library functions. For instance, Haskell’s list comprehension sugar makes use of the library function `concatMap` (which is `map` followed by list concatenation). We therefore allow the declaration of “global” names, together with their type, with the understanding that this name will be available to the desugared code (with the given type).

The declared globals effectively form a primordial type environment, available in conjunction with the ordinary type environment. For example, if + desugars into a call to a global `plus`, the type rule for + is actually (using \( N \) as shorthand for `Number`):

\[
\begin{align*}
\text{\textbf{plus}} & : N, N \rightarrow N, \Gamma \vdash a : N \\
\text{\textbf{plus}} & : N, N \rightarrow N, \Gamma \vdash \beta : N \\
\text{\textbf{plus}} & : N, N \rightarrow N, \Gamma \vdash a + \beta : N
\end{align*}
\]

However, this is both verbose and unusual, so we opt to leave the \( N, N \rightarrow N \) implicit. We do so by adding the type rule:

\[
\frac{}{\Gamma \vdash x : t}
\]

which allows `plus` to be left out of \( \Gamma \).

6.4.5 **Variable Arities**

We support syntactic constructs with variable arity by having a sort called \( e^* \) that represents a sequence of expressions:

\[
e^* ::= \epsilon \quad \text{empty sequence} \\
| \text{(cons } e e^*) \quad \text{nonempty sequence} \\
| a \quad \text{pattern variable}
\]

SweetT supports these sequences by providing:

- The above grammar production, allowing a language’s grammar to refer to \( e^* \).
- Proper handling of sequences in the unification algorithm, allowing them to be resugared.
- Built-in operations for accessing the \( n \)'th element of a sequence, and for asserting that a type judgment holds for all expressions in a sequence.

SweetT likewise supports sequences of types, \( t^* \), and records of both expressions and types.

Using this feature, a simple variable-arity or sugar can have production rule (or \( e^* \)), and desugaring rules:
(or (cons α e)) ⇒ α
(or (cons α (cons β δ))) ⇒ if α then true else (or (cons β δ))

Type resugaring produces one type rule for each desugaring rule:

\[
\begin{align*}
\text{sugar-or-1} & : \quad \Gamma \vdash \alpha : A \\
\Gamma & \vdash (\text{or} \ (\text{cons} \ \alpha \ \epsilon)) : A
\end{align*}
\]

\[
\begin{align*}
\text{sugar-or-2} & : \quad \Gamma \vdash (\text{or} \ (\text{cons} \ \beta \ \delta)) : \text{Bool} \\
\Gamma & \vdash \alpha : \text{Bool} \\
\Gamma & \vdash (\text{or} \ (\text{cons} \ \alpha \ (\text{cons} \ \beta \ \delta))) : \text{Bool}
\end{align*}
\]

The first rule may appear to be too general, but it accurately reflects the sugar as written: (or (cons 3 e)) is a synonym for 3 and has type Number. However, we can statically restrict the singleton or to accept only booleans using calc-type:

(\text{or} \ (\text{cons} \ \alpha \ e)) ⇒ \text{calc-type} \ \alpha \ \text{asBool} \ \text{in} \ \alpha

at which point the resugared type rule becomes:

\[
\begin{align*}
\text{sugar-or-1} & : \quad \Gamma \vdash \alpha : \text{Bool} \\
\Gamma & \vdash (\text{or} \ (\text{cons} \ \alpha \ e)) : \text{Bool}
\end{align*}
\]

as probably desired.

### 6.5 Implementation

We have implemented a prototype of our tool in PLT Redex \([19]\), a semantics engineering tool. It can be found at [cs.brown.edu/research/plt/dl/pldi2018/](https://cs.brown.edu/research/plt/dl/pldi2018/).

Among other features, Redex allows one to define judgment forms, and given a judgment form can search for derivations of it.

SweetT takes as input:

- The syntax of a language, given as a grammar in Redex.
- Core language type rules, defined as a judgment form in Redex. We require that these rules be written using equality constraints: if two premises in a type rule would traditionally describe equality by repeating a type variable, SweetT instead requires that the rule be written using two different type variables, with an equality constraint between them—thus making the unification explicit\(^2\)
- Desugaring rules, given by a \textit{lhs} and \textit{rhs}. Each rule has a capture list of variables to be treated unhygienically, as described in section \[6.4.3\] and the rhs of a rule may make use of \textit{calc-type}, as described in section \[6.4.1\].

\(^2\) This is necessary because re-using the same type variable would invoke Redex’s pattern-matching algorithm. This is usually sufficient, because Redex is meant to type a complete term. However, we are typing a partial term, and instead need a more general unification algorithm. So instead, SweetT gathers equations and performs unification itself.
• Type definitions of globals, as described in section 6.4.4

SweetT then provides a \texttt{resugar} function that follows the process outlined at the end of section 6.2, together with the extensions described in section 6.4. If \texttt{resugar} succeeds, it produces the resugared type rule, as well as the derivation which led to it. If it fails, it announces that no derivation was found (or, less likely, that more than one was found, in violation of assumption 4 (unique-rule)).

Assumption 5 (resugaring) is essentially a specification for \texttt{resugar}, and we believe our implementation obeys this property. We provide empirical evidence for this fact, and for the power of SweetT, in the next section.

6.6 Evaluation

There is no standard benchmark for work in this area. Therefore, we evaluate our approach in two ways. First, we try resugaring on a number of sugars we create atop existing type systems, to ensure that it can support that variety of type systems. Second, we show some case studies which validate that it can handle interesting sugars.

6.6.1 Type Systems

We evaluate SweetT by implementing a number of type systems from Types and Programming Languages (tapl [9]). We tested the type systems in Part II of talpl (except for references, pg. 167), as well as two later systems (subtyping and existentials). Altogether, this is:

• Booleans (pg. 93)
• Numbers (pg. 93)
• Simply Typed Lambda Calculus (pg. 103)
• Unit (pg. 119)
• Ascription (pg. 122)
• Let binding (pg. 124)
• Pairs (pg. 126)
• Tuples (pg. 128)
• Records (pg. 129)
• Sums (pg. 132)
• Variants (pg. 136)
• General recursion (pg. 144)
• Lists (pg. 147)
• Error handling (pg. 174)
• Algorithmic subtyping (pg. 212)
• Existential types (pg. 366)

We tested each type system by picking one or more sugars that made use of its features, resugaring them to obtain type rules, and validating the resulting type rules by hand. All of them resugared successfully.

Three type systems required extending SweetT’s unification algorithm: records and lists needed built-in support, as described in section 6.4.5 and subtyping required adding subtyping constraints, as well as a new \( t\text{-sub-premise} \) rule. References (pg. 167) would have required changing the form of judgments, from \( \Gamma \vdash e : t \) to \( \Gamma, \gamma \vdash e : t \) where \( \gamma \) is a store environment, which would be a more extensive change.

### 6.6.2 Case Studies

We describe six case studies below.

The first three are simpler than the rest. We describe them briefly, and show them in figs. 18 and 19. For each, the figure first shows the relevant core language type rules, then the sugar, then its core derivation, and finally the resugared type rule. To make them fit, we show all of the derivations after unification, eliminating equality constraints.

The last three case studies are more complex, so we discuss them more but do not show their type derivations (which do not fit on a page).

**letrec**  The letrec sugar (fig. 18) introduces recursive bindings using \( \lambda \) and fix (the fixpoint operator).

**\lambda ret**  The \( \lambda \) ret sugar (fig. 18) implements return in functions using tapl-style exceptions (using String as the fixed exception type). The variable return is marked as capturing in the sugar, and thus appears explicitly in the resulting type rule.

**upcast**  The upcast sugar (fig. 19) converts an expression to a supertype of its type via \( \eta \)-expansion. Notice that the core language type system contains subtyping judgments, as mentioned in section 6.6.1.

**foreach**  We consider a functional foreach loop, that performs a map on a list, and also provides break within the loop. If break is called, the loop halts and returns the elements processed so far. Its desugaring is:

\[
\text{foreach } x \text{ list body} \Rightarrow
\]
6.6 Evaluation

\[ t \text{-foreach} \quad \Gamma \vdash \text{list} : \text{List } D \quad \Gamma, x : D, \text{break} : (\text{Unit} \rightarrow B) \vdash \text{body} : F \]
\[ \Gamma \vdash \text{foreach } x \text{ list body} : \text{List } F \]

Figure 17: foreach type rule

```
letrec loop : ((List a) -> (List b) -> (List b)) =
    (λ (lst : (List a)) (acc : (List b))
      if (isnil lst)
      then acc
      else
        try
        let break = (λ (_ : Unit) raise "") in
        let x = head lst in
        loop (tail lst) (cons body acc)
        with (λ (_ : String) acc))
  in reverse (loop list nil)
```

where `reverse` is a global (section 6.4.4) with type \([i] \rightarrow [i]\). In addition, this sugar is declared to capture the variable `break` (see section 6.4.3).

The resugared type rule for `foreach` is show in fig. 17. It demonstrates how different variables must be handled. In the desugaring, when `body` is used, several variables are in scope: `loop`, `lst`, `acc`, `break`, and `x`. However, in the resugared type rule, only `break` and `x` are in scope in the judgment for `body`: `x` because it is an argument to the sugar, and `break` because it is declared as capturing.

**Haskell List Comprehensions** List comprehensions [61 section 3.11] are given by the following transformation:

\[
\begin{align*}
[e | \text{True}] &= [e] \\
[e | q] &= [e | q, \text{True}] \\
[e | b, Q] &= \text{if } b \text{ then } [e | Q] \text{ else } [] \\
[e | p \leftarrow l, Q] &= \text{let } ok p = [e | Q] \\
&\quad \text{ok } _ = [] \\
&\quad \text{in concatMap } ok l \\
[e \mid \text{let decls, Q}] &= \text{let decls in } [e \mid Q]
\end{align*}
\]

A Haskell list comprehension has the form \([e | Q]\), where \(e\) is an expression and \(Q\) is a list of qualifiers. There are three kinds of qualifiers, which are visible in the rules above: (i) boolean guards \(b\) perform a filter; (ii) generators \(p \leftarrow l\) perform a map; and (iii) `let decls` declare local bindings.
We will ignore first two rules (which are uninteresting base cases), and focus on the last three, that introduce qualifiers. To resugar these three kinds of qualifiers, we declare `concatMap` as a global with type `(i -> [o]) -> [i] -> [o]`, as described in section 6.4.4 We also simplify the generator desugaring to consist of a single variable binding, because that is what is available in the TAPL core language we are desugaring to. Finally, we use `calc-type` (section 6.4.1) to determine the type of elements in generators. Thus, we are resugaring these slightly modified rules:

\[
\begin{align*}
[e \mid b, Q] &= \text{if } b \text{ then } [e \mid Q] \text{ else } [] \\
[e \mid x <- l, Q] &= \text{concatMap } \lambda(x :: t) \rightarrow [e \mid Q] \ l \\
[e \mid \text{let } x = e2, Q] &= (\text{let } x = e2 \text{ in } [e \mid Q])
\end{align*}
\]

SweetT resugars these rules, producing the following type rules (transcribed into Haskell syntax):

\[
\begin{align*}
t\text{-hlc-guard} & \quad \frac{\Gamma \vdash [e \mid b, Q] : C \quad \Gamma \vdash b : \text{Bool}}{\Gamma \vdash [e \mid b, Q] : C} \\
t\text{-hlc-gen} & \quad \frac{\Gamma, x : t \vdash [e \mid Q] : [o] \quad \Gamma \vdash l : [t]}{\Gamma \vdash [e \mid x <- l, Q] : [o]} \\
t\text{-hlc-let} & \quad \frac{\Gamma \vdash [e \mid \text{let } x = e2, Q] : B \quad \Gamma \vdash e2 : A}{\Gamma \vdash [e \mid \text{let } x = e2, Q] : B}
\end{align*}
\]

**NEWTYPE** Let us now look at a desugaring of `new-type` into existential types. The core language will have constructs for packing and unpacking existentials:

\[
\begin{align*}
t\text{-pack} & \quad \frac{\Gamma \vdash e : [X \mapsto U] t}{\Gamma \vdash \text{pack } (U e) \text{ as } (\exists X t) : (\exists X t)} \\
t\text{-unpack} & \quad \frac{\Gamma \vdash e_1 : (\exists X t_1) \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{unpack } e_1 \text{ as } (\exists X x) \text{ in } e_2 : t_2}
\end{align*}
\]

We define a new-type sugar that presents a concrete type `T` as an abstract type `X`, and provides wrapping and unwrapping functions (with user-chosen names) that convert from `T` to `X` and from `X` to `T` respectively. The desugaring is:

\[
\text{new-type (wrap unwrap) of } T \text{ as } X \text{ in body} \\
\Rightarrow \\
\text{unpack } (\text{pack } (T \text{ (pair id id) as } (\exists X (\text{Pair } (T \mapsto X)(X \mapsto T)))) \text{ as } (\exists X w) \text{ in let } \text{wrap} = \text{fst } w \text{ in} \\
\text{let } \text{unwrap} = \text{snd } w \text{ in} \\
\text{body}
\]

"hlc" stands for "Haskell list comprehension".
where \( \text{id} \) is a global (section 6.4.4) identity function.

This sugar is successfully resugared to give the type rule:

\[
\frac{\text{t-new-type}}{\Gamma, u : X \rightarrow T, w : T \rightarrow X \vdash \text{body} : A}
\]

\[
\Gamma \vdash \text{new-type (} w u \text{) of } T \text{ as } X \text{ in body} : A
\]

Notice that this type rule does not mention existentials in any way, thereby hiding the underlying implementation method and sparing the programmer from needing to understand anything but \text{new-type} itself.

6.7 Related Work

Work with the same goal We know of a few pieces of work with the same end goal as us: to take a language with syntactic sugar, and type check it without allowing for the possibility of a user seeing a type error in the core language.

In Lorenzen and Erdweg’s SoundExt [53], desugaring comes before type checking. Their formalism takes (i) a type system for the core language, (ii) a type system for the surface language, and (iii) desugaring rules. It then statically verifies that the surface type system is consistent with the core type system. More precisely, they ensure that, for any program, if that program type-checks in the surface language, then its desugaring must type-check in the core language. Our approach has a critical advantage over theirs: we do not require type rules to be written for the surface language, but rather infer them. This simplifies the process of extending the language, restoring the adage “oh, that’s just syntactic sugar”. We believe this is especially valuable to authors of, say, domain-specific languages, who are experts in a domain but may not be in the definition of type systems.

Lorenzen and Erdweg’s later SoundX [54] shows how to integrate desugaring and type rules, so that the same rule can serve both to extend desugaring and to extend the type system. Essentially, the LHS of a desugaring rule is given as a type rule, and the RHS is given as an expression (per usual). Again, the difference with our work is that we do not require type rules to be written for the surface language.

In a similar vein, both Granz et al.’s MacroML [30] and Mainland’s MetaHaskell [55] are staged programming languages. They provide the same guarantee as SoundExt and SoundX: in the words of Mainland, “Well-typed metaprograms should only generate well-typed object terms.” Therefore, as with our work, a user is guaranteed never to see a type error in desugared code. Unlike SweetT, these staged systems allow macro definitions to be interspersed with code. On the other hand, they do not allow macros to check the type of an ex-
expression (as in our calc-type, section 6.4.1), or to inspect code (they can only build code up from smaller fragments).

Omar et al. provide a syntactic extension mechanism for Wyvern called “type-specific languages” (TSLs) [58]. They note that syntactic extensions often conflict with each other, but can be resolved based on the type that the syntax is checked against. As a simple example, Python uses the same syntax {...} for both sets and dictionaries. The syntax {} is thus ambiguous, but this could be resolved by checking whether the type context expected a set or a dictionary. This is the purpose of TSLs. Like MacroML and MetaHaskell, Wyvern TSLs can only construct code, and cannot inspect or deconstruct it. (This is sufficient for their main intended use case, which is defining language literals.)

Finally, Heeren et al., and later Serrano and Hage, show how to augment a type system with new hand-written error messages [33,73]. They do so in the context of embedded DSLs that are implemented without syntactic sugar (which is why their work does not immediately apply to our situation). When coding in such an embedded DSL, programmers would normally be confronted with type errors arising from the implementation of the DSL. This line of work allows the DSL author to write custom error messages that instead frame the error in terms of the DSL.

work with similar goals  There are many systems that type check after desugaring; they potentially show a programmer a type error in code the programmer did not write. Some of these are type systems retrofitted onto languages with macros, such as Type Racket [81] and Typed Clojure [5]. They at best use sourcemaps, providing an accurate line number for a potentially confusing message. There are also metaprogramming systems added to languages with types, such as those of Haskell [76], Ocaml [15], and Scala [8]. They permit grammar extension, and allow desugaring to be defined as an arbitrary function from ast to ast. However, while their metasyntactic types capture the syntactic category of an expression (for instance Exp vs. Name in TemplateHaskell), they do not reflect the object types (e.g., expressions of type Int vs. expressions of type String). As a result, they need to type check after desugaring. (Contrast this to MetaHaskell and MacroML, described above.)

Lerner et al.’s SEMINAL [52] is a system for generating error messages in an unusual way: if a program fails to type check, then it searches for one or more small modification to the program such that it would type check, and presents these to the programmer as suggested changes. At a high-level, SEMINAL has a similar goal to type resugaring: it presents suggested modifications in terms of the program that the programmer wrote, rather than (as per type inference)
presenting type errors in terms of a mismatch between type annotations that the programmer did not write.

Wyk et al. introduce forwarding for attribute grammars [54]. Forwarding is a kind of inheritance. With it, instead of specifying all of its attributes directly, a node in the AST can instead opt to forward some of them to a new node of its own construction. If the original node is thought of as a surface node, and the new node is thought of as a core node, then this is a kind of desugaring, and (some of) the attributes of the core node are automatically propagated to the surface node. Thus, if these forwarded attributes include scope or type information, then the surface node’s scope or type has been defined to be the scope or type of its “desugaring”. Notice that this differs from resugaring, though: we lift scope rules (in the previous chapter), and type rules (in this chapter).

Fisher and Shivers’ Ziggurat [22] is a framework for defining a hierarchy of language levels, that makes it easy to attach static analysis of any sort to each level. However, it does not analyze the analysis, so it is possible for one level’s analysis to conflict with that of another.

Similar work has been done for scope rules. For an overview, see section 5.8.

Work with a different goal. Our work could be contrasted with Chang et al.’s Turnstile [10]. Turnstile is a macro-based framework for defining type systems. However, while it uses desugaring in the implementing language, it has no support for sugar in the implemented language. To this end, it is a competitor to other lightweight language modeling tools (like Redex), and we could have used Turnstile instead of Redex as the basis for our work (we settled on Redex for various practical reasons).

6.8 Discussion and Conclusion

In this chapter, we have presented an algorithm and system for type resugaring: given syntactic sugar over a typed language, it reconstructs type rules for that sugar. These rules can be added to a type-checker to check the sugar directly (and produce error messages at the level of the sugar, rather than its expanded code), and also be added to the documentation of the surface language. We show that the system can handle a variety of language constructs, and that it successfully suppresses the details of what the sugar expands to.

We discussed restrictions on the pattern language of sugars in section 6.3.1 and the underlying type system in sections 6.3.2 and 6.3.3. We also presented some limitations of the implementation in section 6.6.1. It is worth investigating to see if these restrictions can be lifted to make this idea even more broadly applicable.
In principle, not much in our work has specifically been about types. Therefore, this idea could just as well be applied to other syntax-driven deductive systems, such as a natural semantics [41] or structural operational semantics [64]. This would correspondingly enable the creation of semantic rules at the level of the surface language, which can not only enrich a language’s documentation but also facilitate its use in, say, a proof assistant.

6.9 Demos

This section shows SweetT in action. Figure 20 shows a sample usage of SweetT, and figs. 21 to 23 show many sugars and their resugared type rules. These sugars are mostly very simple; their purpose is not to show that SweetT can handle interesting sugars (this is what the case studies of section 6.6.2 are for), but rather to show that it can handle several different type systems. In particular, these sugars, together with those shown earlier in this chapter, make use of all of the TAPL type system features listed in section 6.6.1.
Sugars: Letrec and \( \lambda \text{ret} \)

**Core Type Rules:**

\[
\begin{align*}
\text{t-lambda} & \quad \frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x : T. e : (T \rightarrow U)} \\
\text{t-apply} & \quad \frac{\Gamma \vdash f : T \rightarrow U \quad \Gamma \vdash e : T}{\Gamma \vdash (f \ e) : U} \\
\text{t-fix} & \quad \frac{\Gamma \vdash e : T \rightarrow T}{\Gamma \vdash \text{fix} \ e : T} \\
\text{t-letrec} & \quad \frac{\Gamma \vdash e : \text{Str}}{\Gamma \vdash \text{letrec} \ x : C = a \ in \ b : \text{D}} \\
\text{t-try} & \quad \frac{\Gamma \vdash e : T \quad \Gamma \vdash \text{e\_catch} : \text{Str} \rightarrow T}{\Gamma \vdash \text{try} \ e \ with \ \text{e\_catch} : T}
\end{align*}
\]

**Desugaring Rules:**

\[
\begin{align*}
\text{letrec} \ x : C = a \ in \ b & \Rightarrow (\lambda x : C. b) \ (\text{fix} \ (\lambda x : C. a)) \\
\text{\lambda ret} \ x : T. b & \Rightarrow \lambda x : T. \text{try} \ (\text{let return} = (\lambda v : \text{Str}. \text{raise} v) \ in \ b) \ with \ (\lambda v : \text{Str}. v)
\end{align*}
\]

\(\text{\lambda ret} \) is a function with \text{return} automatically bound (i.e., marked as capturing) to escape from the function.

**Core Derivations:**

\[
\begin{align*}
\text{t-premise} & \quad \frac{\Gamma, x : C \vdash \text{letrec} \ x : C = a \ in \ b : \text{D}}{} \\
\text{t-lambda} & \quad \frac{\Gamma, x : C \vdash a : C}{\Gamma \vdash (\lambda x : C. a) : C \rightarrow C} \\
\text{t-apply} & \quad \frac{\Gamma \vdash (\lambda x : C. a) : C \rightarrow C \quad \Gamma, x : C \vdash b : C}{\Gamma \vdash (\lambda x : C. a) \ (b) : C} \\
\text{t-letrec} & \quad \frac{\Gamma, x : C \vdash \text{letrec} \ x : C = a \ in \ b : \text{D}}{\Gamma, x : C \vdash (\lambda x : C. a) \ (\text{fix} \ (\lambda x : C. a)) : \text{D}}
\end{align*}
\]

**Resugared Type Rules:**

\[
\begin{align*}
\text{t-letrec} & \quad \frac{\Gamma, x : C \vdash a : C \quad \Gamma, x : C \vdash b : \text{D}}{\Gamma \vdash \text{letrec} \ x : C = a \ in \ b : \text{D}} \\
\text{t-\lambda ret} & \quad \frac{\Gamma, x : T, \text{return} : (\text{Str} \rightarrow A) \vdash b : \text{Str}}{\Gamma \vdash (\text{\lambda ret} x : T. b) : T \rightarrow \text{Str}}
\end{align*}
\]

Figure 18: Derivation examples
**Sugar: Upcast**

**Core Type Rules:**

- **t-id**
  \[
  \frac{x : T \in \Gamma}{\Gamma \vdash x : T}
  \]

- **t-lambda**
  \[
  \frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x : T. e : (T \rightarrow U)}
  \]

- **t-apply**
  \[
  \frac{\Gamma \vdash f : T \rightarrow U \quad \Gamma \vdash e : T' \quad T' <: T}{\Gamma \vdash (f \ e) : U}
  \]

**Desugaring Rule:**

\[
\text{upcast } a \text{ as } C \Rightarrow (\lambda x : C. x) \ a
\]

**Core Derivation:**

- **t-id**
  \[
  \frac{}{\Gamma, x : C \vdash x : C}
  \]

- **t-lambda**
  \[
  \frac{\Gamma, x : C \vdash x : C}{\Gamma \vdash (\lambda x : C. x) : C \rightarrow C}
  \]

- **t-premise**
  \[
  \frac{}{\Gamma \vdash a : A}
  \]

- **t-sub-premise**
  \[
  \frac{}{A <: C}
  \]

- **t-upcast**
  \[
  \frac{\Gamma \vdash (\lambda x : C. x) \ a : C}{\Gamma \vdash \text{upcast } a \text{ as } C : C}
  \]

**Resugared Type Rule:**

- **t-upcast**
  \[
  \frac{\Gamma \vdash a : A \quad A <: C}{\Gamma \vdash \text{upcast } a \text{ as } C : C}
  \]

Figure 19: Derivation examples (cont.)
#lang racket

(require redex)
(require "../resugar.rkt")

(define-resugarable-language demo
  #:keywords(if true false Bool)
  (e ::= ....
    (if e e e))
  (v ::= ....
    true
    false)
  (t ::= ....
    Bool)
  (s ::= ....
    (not s)))

(define-core-type-system demo
  [(\(\Gamma\) e_1 t_1)
     (\(\Gamma\) e_2 t_2)
     (\(\Gamma\) e_3 t_3)
     (con (t_1 = Bool))
     (con (t_2 = t_3))
     ------- t-if
     (\(\Gamma\) (if e_1 e_2 e_3) t_3)]

  [------- t-true
   (\(\Gamma\) true Bool)]

  [------- t-false
   (\(\Gamma\) false Bool)])

(define rule_not
  (ds-rule "not" #:capture()
    (not ~a)
    (if ~a false true)))

(view-sugar-type-rules demo \(\Gamma\) \(\Gamma\) (list rule_not))

\[\begin{array}{c}
\text{INFERS THE FOLLOWING RESUGARED TYPE RULE:}\\
\frac{\text{not}}{(\Gamma \vdash \text{a : Bool})}
\frac{\cdot}{(\Gamma \vdash \text{not a} : \text{Bool})}
\end{array}\]

Figure 20: Sample SweetT usage
Figure 21: Sample sugars, p. 1

Let

$$\frac{a : \text{Bool}}{(\text{let } x = a \text{ in } \text{unit}) : \text{unit}}$$

Ascription

$$\frac{(\text{let } x = a \text{ in } \text{unit}) : \text{unit}}{a : \text{Nat}}$$

Unit

$$\frac{a : \text{Nat}}{(\text{unit } a) : \text{Nat}}$$

STLC

$$\frac{a : \text{Nat} \land (\text{let } x = a \text{ in } \text{unit}) : \text{unit}}{a : \text{Nat} \land (\text{unit } a) : \text{Nat}}$$

Nats

$$\frac{a : \text{Nat} \land (\text{let } x = a \text{ in } \text{unit}) : \text{unit}}{a : \text{Nat} \land (\text{unit } a) : \text{Nat}}$$

Booleans

$$\frac{(\text{unit } a) : \text{unit}}{(\text{unit } a) : \text{unit}}$$
Variant

(ds-rule "inl" #:capture()
(inl* ~l)(calctype ~l as t_l in
(variant l = ~l as (Variant (field l t_l (field r t_r ϵ)))))

Γ ⊢ l : A
inl (Γ ⊢ (inl* l) : (Variant (field l t_l (field r t_r ϵ))))

Variant

(ds-rule "inr" #:capture()
(inr* ~r)(calctype ~r as t_r in
(variant r = ~r as (Variant (field l t_l (field r t_r ϵ)))))

Γ ⊢ r : A
inr (Γ ⊢ (inr* r) : (Variant (field l t_l (field r A ϵ))))

Variant

(ds-rule "case" #:capture()
(case* ~e of (x => ~l) (y => ~r))
(case ~e of (cons (l = x => ~l) (cons (r = y => ~r) ϵ)))

Γ ⊢ e : (Variant (field l C (field r H ρ1)))

((bind x C Γ ⊢ l : K) ((bind y H Γ ⊢ r : K))
case (Γ ⊢ (case* e of (x => l) (y => r)) : K))

Fixpoint

(ds-rule "letrec" #:capture()
(letrec x : ~t = ~a in ~b)
((λ (x : ~t) ~b) (fix (λ (x : ~t) ~a))))

((bind x C Γ ⊢ a : C) ((bind x C Γ ⊢ b : D))
letrec (Γ ⊢ (letrec x : C = a in b) : D))

Show that variants make sums irrelevant.


[73] Alejandro Serrano and Jurriaan Hage. Type error diagnosis for embedded dsls by two-stage specialized type rules. In European Symposium on Programming Languages and Systems, New York, NY, USA, 2016. ACM.


