Resugaring: Lifting Languages through Syntactic Sugar

Justin Pombrio

March 26, 2018

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.

ABSTRACT 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.

CONTENTS

ACKNOWLEDGEMENTS .

I would like to thank my thesis committee—Mitchell Wand, Eelco Visser, and Shriram Krishnamurthi—for overseeing this large and technical body of work.

I would like to thank Daniel J. Dougherty and Matthias Felleisen for their feedback on evaluation resugaring, and Joe Politz for providing a language that offered an excellent proving ground for it (chapter 4).

I would like to thank Paul Stansifer, Sebastian Erdweg, and Matthew Flatt for their feedback on scope inference (chapter 5).

I would like to thank Sorawee "Oak" Porncharoenwase for helping to generalize this work and integrate it with Pyret, as part of his Honors thesis.

Some of the papers that led to this thesis were rough around the edges when we initially submitted them. I would like to give many thanks to the many reviewers who gave very thorough feedback on these intial versions. This thesis owes much of its quality to these anonymous individuals.

Finally, I would like to thank the NSF, which partly funded all of this work.

Part I

SYNTACTIC SUGAR

1

SYNTACTIC SUGAR

1.1 WHAT IS IT?

The term *syntactic sugar* was introduced by Peter Landin in 1964 [37]. 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:

```
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:

```
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." [23, section 14.14.2] Ignoring some irrelevant details, it states that this *sugar*:

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

desugars into:

```
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:

Words can be thought of as pointing to clusters in concept-space. An extensional definition like this is an attempt to draw a neat box around such a cluster, which is always a little dubious because clusters typically have fuzzy boundaries and aren't box-shaped.

> 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 subset 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 [24, 48, 49].

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 [54] 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 shines.

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 be 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 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 focus on the importance of the abstraction provided by sugar not leaking [58]. 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 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.

[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.

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 (likewise implicitly defined) scope rules for a surface language.

type rules Finally, in typed languages with syntactic sugar, type error messages frequently reveal the desugared code (which the programmer didn't write), thus breaking the sugar's abstraction.

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:

While I use "we" throughout the rest of document to acknowledge all the others who helped with this work, my thesis statement is for me to defend. Hence "my".

> 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, allowing future implementers and researchers free to take full advantage of it 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:

Related work is dicussed individually in chapters chapters 4 to 6.

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

2

DESUGARING IN THE WILD

This chapter is under development.
TODO

- State version of each system discussed
- mention macro-defining-macros in expressiveness?
- More examples

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

We stretch this taxonomy to include 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?

DESUGARING 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)?

PHASE *When* does desugaring occur? Is it at compilation time, or at evaluation time (as in staged metaprogramming systems)?

EXPRESSIVENESS How expressive is it? Can a sugar take an arbitrary expression as a parameter, or only a primitive type (PARAMETER)? If it can take an expression, can it deconstruct it, or only use it parametrically (DECONSTRUCTION)? Do sugars exclusively produce expressions, or can they produce any kind of syntax (RESULT)?

SAFETY How *safe* is it? Can desugaring produce syntactically invalid code (SYNTAX SAFETY)? Can it produce an unbound variable (SCOPE SAFETY), or accidentally capture a variable (HYGIENE)? Can it produce code that contains a type error (TYPE SAFETY)?

There are two big clusters of desugaring systems that are worth calling out by name: macros and staged metaprogramming. Racket [CITE] is a prototypical language with macros: they are user-defined, the metalanguage is either a DSL or the Racket language itself, its macros are expanded (primarily) during compilation, and the desugaring order is outside-in.

Staged metaprogramming also involves user-defined sugars, but instead of desugaring as a separate phase, code is constructed, composed, and eventually `eval`'ed at runtime. MetaOCaml is a prototypical metaprogramming language: the metalanguage is MetaOCaml itself, the desugaring order is inside-out (reflecting the evaluation order of OCaml), and its sugars are user-defined and are expanded at runtime. This thesis is really about sugars that are expanded at compile time—so staged metaprogramming systems are technically out of scope—but they are closely enough related that we include them in the taxonomy.

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 visual and block based language/editors), 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 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 2.2.1.

There are variations among tree representations as well: desugaring rules may work over the concrete syntax of the language, or over its abstract syntax. We discuss this further in [REF].

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 [CITE] and implemented in the compiler(s); thus they are language-defined. Template Haskell sugars [CITE], 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: if done correctly, 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.

[TODO: Consider moving the rest of this prose into a different section.]

When sugars are user-defined, it raises difficulties with tools such as editors that need to support the new syntax. For example, if sugars are language-defined then an editor can just support the full language. If they are user-defined, however, how can an editor provide correct indentation, syntax highlighting, etc.?

Different languages work around this problem in different ways.

Lisps partly avoid this problem by having 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, `(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*, by analogy to legislative houses that are split into a lower and upper level.

The phrase bicameral syntax *was coined by Shriram Krishnamurthi <CITE plai>.*

This separation of concerns makes it easy for editors to support the *first* layer of syntax, which cannot be modified by syntactic sugar, while ignoring the second, which can. Doing so only *partly* avoids the problem. For example, the DrRacket editor for Racket [CITE] allows indentation schemes to be set on a per-macro basis (because different syntactic forms, while purely parenthetical, still have varying nesting patterns that should be indented differently), and it displays arrows showing where variables are bound by being macro-aware and expanding the program (see [REF]).

[FILL: SugarJ, other examples] [FILL: Spoofax, language workbenches: tie editor to language]

2.1.3 *Metalanguage*

The *metalanguage* is the language that the sugars are written in. It could be the same language that programs are written in, a different (but still general purpose) language, or a DSL especially for sugars (such as the pattern-based `define-syntax-rule` DSL we've been using for examples).

2.1.4 *Desugaring Order*

There are two major desugaring strategies used in desugaring systems. They loosely correspond to eager and lazy runtime evaluation, but differ in some important ways, so we will instead refer to them by their original names [CITE]: Outside-in (OI) and Inside-out (IO) desugaring:

You can also ask whether desugaring proceeds left-to-right or right-to-left. However, this is a comparatively minor detail, so we ignore it.

OI desugaring is superficially similar to lazy evaluation, in that desugaring proceeds from the outside in. However, there is one crucial difference: in lazy evaluation, if a function uses (e.g., does case analysis on) one of its (lazy) arguments, that forces the evaluation of that argument. In contrast, if an OI 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 oi is the only possible desugaring order.

io desugaring is similar to eager evaluation: the innermost sugars desugar first. The above paragraph suggested that io 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; (iii) sugars act on the ast rather than on concrete syntax; or (ii) sugars cannot deconstruct their parameters, so the desugaring order is largely irrelevant anyways; . In these settings, io 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 *Phase*

The *phase* of a desugaring system is *when* desugaring happens. This is typically at compile-time, but it doesn't have to be. For example, staged metaprogramming systems construct and evaluate programs at run-time [TODO: test], and Racket macros can run at an arbitrary number of different phases [REF].

2.1.6 *Expressiveness*

We will examine three 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?

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.

Of course, three 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:

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.

SYNTACTIC SAFETY Can a sugar expand to syntactically invalid code? For example, in the C Preprocessor (section 2.2.1), 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 analogous to type checking: a type checker will warn that a function definition contains a type error even, if if that function is never used.

HYGIENE Unlike the other safety properties discussed here, hygiene [CITE] 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):

Why not just write `(if a a b)`? That wouldn't work well if `a` had side effects.

```
(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*.

Some of the examples here were inspired by Clinger and Rees CITE.

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 (and in the example, see that the `temp` in `or false temp` should be bound by the surrounding `let`), and 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). There is more to say about hygiene, and we will discuss it further in [REF], but this is sufficient for our present taxometric purposes.

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 a type error? 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):

```
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.

2.2.1 *C Preprocessor*

Desugaring Order: IO
Authorship: User-defined
Representation: Token stream
Safety: [FILL]
Discussion: The C Preprocessor (hereafter cpp) [CITE] is a *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. Cpp is not Turing complete, 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 cpp operates on tokens, and is thus unaware of the higher-level syntax of C [CITE]. As an example, consider this innocent looking cpp desugaring rule that defines an alias for subtraction:

```
#define SUB(a, b) a - b
```

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

```
SUB(0, 2 - 1))
```

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

```
#define SUB(a, b) (a) - (b)
```

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

```
SUB(5, 3) * 2
```

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

```
#define SUB(a, b) ((a) - (b))
```

In general, both the inside boundary of a rule (the parameters `a` and `b`), and the outside boundary (the whole rhs) need to be protected to ensure that the expansion is parsed correctly. If the sugar is used in expression position, as in the `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 `do {...} while(0)`. Software developers should not need to know this.

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 essentially cannot be fixed from within the paradigm. *Overall, code transformations should never operate at the level of text.*

2.2.2 *C++ Templates*

Representation: Concrete syntax
Authorship: User-defined
Metalanguage: C++ templates (pattern based)
Parameters: Types and primitive values
Result: Function/method definition, struct/class definition, or type alias
Deconstruction: Yes (although the parameters are limited)
Desugaring Order: IO
Phases: One
Syntax safe: Yes
Scope safe: NA
Type safe: NA

C++ templates [CITE] are not general-purpose sugars, because they cannot take code as an 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. Let's use the following template declaration, taken from [CITE: pg344], as a running example. It declares a function to compute the area of a circle, that can be instantiated with different possible (presumably numeric) types `T`:

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

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:

```
  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:

If a template is invoked multiple times with the same parameters, only one copy of the code will be made, however.

```
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 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 `int`s. 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.

2.2.3 *Rust Macros*

Representation: Concrete Syntax
Authorship: User-defined
Desugaring Order: OI
Safety: [FILL]
Discussion:

2.2.4 *Haskell Templates*

Representation: Concrete or abstract syntax
Authorship: User-defined
Metalanguage: Haskell
Parameters: Any
Result: Any
Deconstruction: Yes
Desugaring Order: IO
Phases: One
Syntax Safe: Yes
Scope Safe: No
Type Safe: No

NOTES:

- Must be defined in separate file

2.2.5 *MetaOCaml*

Representation: Concrete syntax
Authorship: User-defined
Metalanguage: OCaml
Parameters: Expressions (+ OCaml values)
Result: Expressions (+ OCaml values)
Deconstruction: No
Desugaring Order: IO
Phases: Many
Syntax Safe: Yes

Scope Safe: Yes
Type Safe: Yes

3

NOTATION

In this chapter, we set the groundwork of notation and basic 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 (a la Scheme's `syntax-rules` [30].) 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, we require ASTs to explicitly distinguish between variable *declarations* x^{D} (i.e., binding sites), and variable *references* x^{R} (i.e., uses). This distinction 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 e. Terms can be inductively defined as:

While t would be a more obvious letter to use for terms, we use it to refer to types in chapter 6.

$$\begin{array}{lcll}
\text{constructor } C & ::= & \textit{name} & \text{syntactic construct name} \\
\text{term } e & ::= & \textit{value} & \text{primitive value} \\
 & | & (C\ e_1 \ldots e_n) & \text{AST node} \\
 & | & x^{\text{R}}_i & \text{variable reference at posn. } i \\
 & | & x^{\text{D}}_i & \text{variable declaration at posn. } i
\end{array}$$

3.2 Patterns

We require that desugaring be based on *patterns*: it proceeds by *matching* a LHS pattern against a term, and then *substituting* into the RHS pattern.

Patterns p are just terms that can contain pattern variables α:

$$\begin{array}{lcll}
\text{pattern } p & ::= & \alpha & \text{pattern variable} \\
 & | & \textit{value} & \text{primitive value} \\
 & | & (C\ p_1 \ldots p_n) & \text{AST node} \\
 & | & x^{\text{R}}_i & \text{variable reference at posn. } i \\
 & | & x^{\text{D}}_i & \text{variable declaration at posn. } i
\end{array}$$

Matching a term against a pattern produces an *environment*, or a mapping from pattern variable to term:

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

We will write desugaring rules with a double arrow:

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

3.3 DESUGARING

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

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

and we want to *expand* the term $(\texttt{Let}\ x^{\mathrm{D}}\ 1\ x^{\mathrm{R}})$. To do so, we match against the LHS:

$$\gamma = (\texttt{Let}\ x^{\mathrm{D}}\ 1\ x^{\mathrm{R}})/(\texttt{Let}\ \alpha\ \beta\ \gamma) = \{\alpha \mapsto x^{\mathrm{D}},\ \beta \mapsto 1,\ \gamma \mapsto x^{\mathrm{R}}\}$$

and substitute into the RHS:

$$\{\alpha \mapsto x^{\mathrm{D}},\ \beta \mapsto 1,\ \gamma \mapsto x^{\mathrm{R}}\} \bullet (\texttt{Apply}\ (\texttt{Lambda}\ \alpha\ \gamma)\ \beta) = (\texttt{Apply}\ (\texttt{Lambda}\ x^{\mathrm{D}}\ x^{\mathrm{R}})\ 1)$$

That was the *expansion* of a single rule, a.k.a. a single *rewrite*. *Desugaring* is the expansion of all sugars in a term. We write it as $\mathbb{D}(e)$, and assume that it is in OI order (see section 2.1.4).

3.3.1 *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.

WELL-FORMEDNESS CRITERIA FOR DESUGARING RULES

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 contained in its pattern variables during expansion.

RESTRICTIONS ON DESUGARING RULES

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. *Each desugaring rule's* LHS *must have the form* $(C\ e_1, ..., e_n)$. We will rely on this fact when showing that unexpansion is an inverse of expansion in section 4.6.2.

3. References and declarations in the RHS are given fresh names during expansion to ensure hygiene.

4. *The rules'* LHSs *must be disjoint.* If they are not, it presents issues both for resugaring evaluation sequences, and resugaring type systems.

Some of the resugaring chapters go beyond the assumptions shown here: 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 OI here). However, all chapters handle at least these kinds of sugars.

Part III

RESUGARING

4

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) [50, 51].

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 4.7 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 `Or`, as used in languages like Lisp and Scheme. We assume the surface language has `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 `Or` (that let-binds its first argument in case it has side-effects):

We present these examples using the actual syntax of Confection, *which uses* `x` *instead of* α *for a pattern variable, and represents variables as strings.*

```
Or(x, y) -> Let([Binding("t", x)],
                If(Id("t"), Id("t"), y)));
```

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

Consider the surface term `not(true) OR not(false)`. After desugaring, this would evaluate as follows in the core language (assuming a typical call-by-value evaluator):

```
    let t = not(true) in
      if t then t else not(false)
⟶  let t = false in
      if t then t else not(false)
⟶  if false then false else not(false)
⟶  not(false)
⟶  true
```

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

```
    not(true) OR not(false)
--→ false OR not(false)
--→ not(false)
--→ 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 *unexpand* core terms into surface terms by running rules "in reverse": matching against the RHS and substituting into the corresponding LHS. (To preserve Emulation, unexpansion must be an inverse of expansion: we prove that this is so in section 4.6.2.) We will now show how the last two steps may come about.

4.2.2 *Maintaining Abstraction*

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

```
let t = not(true) in
  if t then t else not(false)
```

it should not unexpand into `not(true) OR not(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 *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 RHS. When this happens, we conclude that there is no surface representation of the core term and skip the step.

Whereas the tags used in hygienic macro expansion [33] specify time steps, the tags in resugaring specify which sugar the code originated from.

To illustrate this, we revisit the same core evaluation sequence as before. "{Head$_{\texttt{Or}}$: }" is a tag on the desugared expression that indicates it originated from the `Or` sugar, and "`{Body: }`" is a tag that indicates it originated from sugar (without saying which, although in this case it came from `Or`):

```
    {Head_Or: let t = not(true) in
      {Body: if t then t else not(false)}}

⟶  {Head_Or: let t = false in
      {Body: if t then t else not(false)}}

⟶  {Body: if false then false else not(false)}

⟶  not(false)

⟶  true
```

The tags on the first two steps suggest that `Or`'s desugaring rule be applied in reverse. They can be unexpanded because they match the RHS of `Or`. The third step fails to resugar because the `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:

```
      not(true) OR not(false)
  --→ 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 4.6.3, 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 4.8) show that we do indeed obtain detailed and useful surface evaluation sequences.

A partial formalization of coverage can be found in the second paper this chapter is based on [51].

4.2.4 *Trading Abstraction for Coverage*

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

```
Or([x, y]) ->
  Let([Binding("t", x)],
      If(Id("t"), Id("t"), y)));
Or([x, y, ys ...] ->
  Let([Binding("t", x)],
      If(Id("t"), Id("t"), Or([y, ys ...])));
```

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 [32].

Consider the surface term (false OR false OR true). Given the revised definition of Or, unexpansion would yield the following lifted evaluation steps:

```
      false OR false OR true
  --→ true
```

In particular, it correctly suppresses any presentation of the recursive invocation of 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:

```
      false OR false OR true
  --→ false OR true
  --→ true
```

Thus, we let sugar authors make part of a rule's RHS visible by prefixing it with !. Here, writing the second Or rule as:

```
Let([Binding("t", x)],
    If(Id("t"), Id("t"), !Or([y, ys ...])))
```

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. The next chapter, section 5.8, shows how to fix this. Three changes must be made: (i) sugars must pass that chapter's *scope inference* algorithm; (ii) introduced variables must (unsurprisingly) be given fresh names, and resugaring must allow introduced variables to match against any name; and (iii) sugars must not drop pattern variables. Given these changes, section 5.8 proves that both desugaring and resugaring will be 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:

```
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):

```
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 λ_{JS} [24], 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 `<func>` denotes a resolved functional):

```
--→ <func>([1, 2])

--→ cases(List) [1, 2]:
```

```
          | empty() => 0
          | link(_, tail) => len(tail) + 1
        end
   --→ <func>([2]) + 1
   --→ (cases(List) [2]:
          | empty() => 0
          | link(_, tail) => len(tail) + 1
        end) + 1
   --→ <func>([]) + 1 + 1
   --→ (cases(List) []:
          | empty() => 0
          | link(_, tail) => len(tail) + 1
        end) + 1 + 1
   --→ 0 + 1 + 1
   --→ 1 + 1
   --→ 2
```

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 4.5.1), we will describe how to use tags to fully desugar and resugar terms, transforming not just a term but its subterms as well. Finally (section 4.5.2), 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.

pattern p	::=	α	pattern variable
	|	$(C\ p_1 ... p_n)$	AST node
	|	$[p_1 \dots p_n]$	list of length n
	|	$[p_1 \dots p_n\ p{*}]$	list of length $\geq n$ (ellipses)
	|	*value*	primitive value
	|	x	variable
	|	$(\texttt{Tag}\ o\ p)$	origin tag
term e	::=	p	pattern w/o pattern variables or ellipses
origin tag o	::=	$(\texttt{Head}\ i\ e)$	marks topmost rule production
	|	$(\texttt{Body}\ \mathit{bool})$	marks each rule production

Variables are denoted by a lowercase identifier (we omit the reference/declaration distinction, and the identity 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 $p{*}$ 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 4.5.1. We do *not* address hygiene in this chapter, but rather discuss it in section 5.8.

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 trx are the natural extension of regular expressions to handle trees: they add a primitive $(C\ trx_1 \dots trx_n)$ for matching a tree node labeled C with branches matching the regular tree expressions $trx_1 \dots trx_n$. 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 paper—our approach is not dependent on the exact definition. The precise requirements for the transformation language are given in section 4.6.3.

4.5 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* $\lceil b_1 \dots b_n \rceil$, or an *ellipsis binding* $\lceil b_1 \dots b_n\ b_e{*} \rceil$. A pattern variable within ellipses is bound to a list binding $\lceil b_1 \dots b_n \rceil$ instead of a list term $[b_1 \dots b_n]$; they behave slightly differently under substitution. Ellipsis bindings are similar, but needed

$$
\begin{array}{lcll}
b & := & e & \text{(term)} \\
  & \mid & \lceil b_1 \ldots b_n \rceil & \text{(list binding)} \\
  & \mid & \lceil b_1 \ldots b_n \, b_e{*} \rceil & \text{(ellipsis binding)} \\
  & & & \\
\gamma & := & \{\alpha \to b, \ldots\} &
\end{array}
$$

Figure 1: Bindings

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 γ 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 γ_1 and γ_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:

```
Or([x y ys ...] ->
  Let([Binding("t" x)]
      If(Id("t") Id("t") Or([y ys ...])));
```

Matching `Or([true Not(true) false true])` against `Or([x y ys ...])` produces the environment

$$\gamma = \{\alpha \to \texttt{true},\ \beta \to \texttt{Not(true)},\ \gamma \to \lceil \texttt{false}\ \texttt{true} \rceil\}$$

and substituting γ into the rule's rhs produces

```
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 $(e/p) \bullet e = e$ 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 the following well-formedness criteria for the lhs and rhs of each rule:

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

2. *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 pattern variables bound to atomic terms.

$$\begin{array}{lcl}
\mathit{value}/\mathit{value} & = & \{\} \\
x/x & = & \{\} \\
e/\alpha & = & \{\alpha \rightarrow e\} \\
[e_1 \ldots e_n]/[p_1 \ldots p_n] & = & \bigcup_{i=1..n}(e_i/p_i) \\
[e_1 \ldots e_n \ldots e_{n+k}]/[p_1 \ldots p_n p_e *] & = & \bigcup_{i=1..n}(e_i/p_i) \cup \mathit{merge}([e_{n+i}/p_e]_{i=1..k}) \\
(C\ e_1 \ldots e_n)/(C\ p_1 \ldots p_n) & = & \bigcup_{i=1..n}(e_i/p_i)
\end{array}$$

$$\begin{array}{lcl}
\gamma \bullet \mathit{value} & = & \mathit{value} \\
\gamma \bullet x & = & x \\
\gamma \bullet [p_1 \ldots p_n] & = & [\gamma \bullet p_1 \ldots \gamma \bullet p_n) \\
\gamma \bullet [p_1 \ldots p_n p_e *] & = & [\gamma \bullet p_1 \ldots \gamma \bullet p_n \mathbin{+\!\!+} \mathit{split}(\gamma, p_e) \\
 & & \text{(where } \mathbin{+\!\!+} \text{ is concatenation)} \\
\{\ldots, \alpha \rightarrow b, \ldots\} \bullet \alpha & = & \mathit{toTerm}(b) \\
\gamma \bullet (C\ p_1 \ldots p_n) & = & (C\ \gamma \bullet p_1 \ldots \gamma \bullet p_n)
\end{array}$$

$$\begin{array}{l}
\mathit{merge}([\{\alpha_1 \rightarrow b_1, \ldots\} \ldots \{\alpha_n \rightarrow b_n, \ldots\}]) \\
\quad = \quad \{\alpha_1 \rightarrow \lceil b_1 \ldots b_n \rceil, \ldots\}
\end{array}$$

$$\begin{array}{l}
\mathit{split}(\{\alpha_1 \rightarrow \lceil b_{11} \ldots b_{1k} \rceil \ldots \alpha_n \rightarrow \lceil b_{n1} \ldots b_{nk} \rceil\}, p) \\
\quad = \quad (\{\alpha_1 \rightarrow b_{11} \ldots \alpha_n \rightarrow b_{n1}\} \bullet p \ldots \{\alpha_1 \rightarrow b_{1k} \ldots \alpha_n \rightarrow b_{nk}\} \bullet p)
\end{array}$$

$$\begin{array}{lcl}
\mathit{toTerm}(p) & = & p \\
\mathit{toTerm}(\lceil b_1 \ldots b_n \rceil) & = & (\mathit{toTerm}(b_1) \ldots \mathit{toTerm}(b_n))
\end{array}$$

Figure 2: Matching and substitution

3. *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.)

4. *Each transformation's* LHS *must have the form* $(C\ e_1 \ldots e_n)$. We will rely on this fact when showing that unexpansion is an inverse of expansion in section 4.6.2.

5. *The transformations'* LHSs *must be disjoint.* We explain the reason for this in section 4.5.

The first two restrictions are further justified by our formalization of expansion and unexpansion in Coq (section 4.6.4), 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 rhss. Formally,

$$\begin{aligned} \texttt{exp}_{rs}\ e &= (j, (e/p_j) \bullet p'_j) \\ & \quad \text{for } j = \min\{i | e \geq p_i\}_i \end{aligned}$$

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 original input term e—with which to bind pattern variables in p_i that do not appear in p'_i. Formally,

$$\texttt{unexp}_{rs}\ (j, e')\ e \quad = \quad ((e/p_j) \cup (e'/p'_j)) \bullet p_j$$

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 γ *stands in* for e, and overload $\texttt{unexp}_{rs}$ by writing:

$$\texttt{unexp}_{rs}\ (j, e')\ \gamma \quad = \quad (\gamma \cup (e'/p'_j)) \bullet p_j$$

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, 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:

```
Max([]) -> Raise("empty list");
Max(xs) -> MaxAcc(xs, -infinity);
```

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

```
    MaxAcc([-infinity], -infinity)
⟶ MaxAcc([], -infinity)
```

and the derived surface evaluation sequence is:

```
      Max([-infinity])
  --→ Max([])
```

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

Fortunately, the 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:

```
Max([]) -> Raise("Max: given empty list");
Max([x, xs ...]) -> MaxAcc([x, xs ...], -infinity);
```

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

CONFECTION implements a static check that admits the second definition but not the first. It checks that the LHSs 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 4.6.1.

4.5.1 *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: Head tags mark the outermost term constructed by a rule application, and Body tags mark each non-atomic term constructed by a rule application. Body tags serve to distinguish rule-generated code from user-written code, thereby maintaining Abstraction. They are automatically inserted into each rule's RHS during parsing. Crucially, these tags can be considered simply part of the 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 visible to the user. CONFECTION allows sugar authors to do so by prefixing a term with '!'. Each Body tag contains a boolean indicating whether it was made visible 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 γ 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 $\texttt{exp}_{rs}$ at each node:

Other desugaring orders, e.g., bottom-up instead of top-down, are possible; we follow the precedent set by Scheme macros.

$$
\begin{array}{lcl}
\texttt{desugar}_{rs}\ \mathit{value} & = & \mathit{value} \\
\texttt{desugar}_{rs}\ x & = & x \\
\texttt{desugar}_{rs}\ (C\ e_1,\ ...,\ e_n) & = & \texttt{desugar}_{rs}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ e') \\
 & & \text{where } \gamma \text{ stands in for } (C\ e_1,\ ...,\ e_n)/p_i \\
 & & \text{when } \texttt{exp}_{rs}\ (C\ e_1,\ ...,\ e_n) = (i, e') \\
\texttt{desugar}_{rs}\ (C\ e_1,\ ...,\ e_n) & = & (C\ \texttt{desugar}_{rs}\ e_1, ..., \texttt{desugar}_{rs}\ e_n) \\
 & & \text{otherwise} \\
\texttt{desugar}_{rs}\ (e_1 \ldots e_n) & = & (\texttt{desugar}_{rs}\ e_1 \ldots \texttt{desugar}_{rs}\ e_n) \\
\texttt{desugar}_{rs}\ (\texttt{Tag}\ o\ e) & = & (\texttt{Tag}\ o\ \texttt{desugar}_{rs}\ e)
\end{array}
$$

Resugaring can be performed by traversing a term, this time performing $\texttt{unexp}_{rs}\ (i, e)\ \gamma$ for any term e tagged with $(\texttt{Head}\ i\ \gamma)$. Thus $\texttt{resugar}_{rs}$ 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 $\texttt{unexp}_{rs}$.

If the unexpansion of any particular term fails, then resugaring as a whole fails, since the tagged term in question can neither be accurately represented as the result of an expansion nor shown as-is. Furthermore, resugaring should fail if any opaque `Body` tags remain. This ensures that code originating in sugar (and therefore wrapped in `Body` tags) is never exposed, guaranteeing Abstraction.

$$
\begin{array}{lcll}
\texttt{resugar}_{rs}\ e & = & \texttt{R}'_{rs}\ e & \text{when } \texttt{R}'_{rs}\ e \text{ has no opaque tags} \\
\texttt{resugar}_{rs}\ e & = & \bot & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{R}'_{rs}\ \mathit{value} & = & \mathit{value} \\
\texttt{R}'_{rs}\ x & = & x \\
\texttt{R}'_{rs}\ (\texttt{Tag}\ (\texttt{Body}\ b)\ e) & = & (\texttt{Tag}\ (\texttt{Body}\ b)\ \texttt{R}'_{rs}\ e) \\
\texttt{R}'_{rs}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ e') & = & \texttt{unexp}_{rs}\ (i, \texttt{R}'_{rs}\ e')\ \gamma \\
\texttt{R}'_{rs}\ (C\ e_1,\ ...,\ e_n) & = & (C\ \texttt{R}'_{rs}\ e_1,\ ...,\ \texttt{R}'_{rs}\ e_n) \\
\texttt{R}'_{rs}\ (e_1 \ldots e_1) & = & (\texttt{R}'_{rs}\ e_1 \ldots \texttt{R}'_{rs}\ e_n)
\end{array}
$$

4.5.2 *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->e` , `e->s` , `c->e` , and `e->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:

$$\begin{aligned}
\mathbb{D} &= \texttt{s->e ; desugar}_{rs}\texttt{ ; e->c} \\
\mathbb{R} &= \texttt{c->e ; resugar}_{rs}\texttt{ ; e->s}
\end{aligned}$$

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

```
def showSurfaceSequence(s):
  let c = desugar*(s)
  while c can take a reduction step:
    let s' = resugar*(c)
    if s': emit(s')
    c := step(c)
```

Implementing this requires a `step` relation; though most languages don't provide one natively, section 4.7 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 `desugar(s)`, and repeatedly dequeing 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 paper were run. It uses a user-written *grammar file* that specifies grammars 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 [5], the rules obey the semantics described in this paper. The rules are also checked against the well-formedness criteria of section 4.5, thus ensuring that our results hold. Confection is available at:

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

4.6 formal justification

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

4.6.1 *Transformations as Lenses*

We have found it helpful to view our transformation rules from the perspective of lenses [21]. In particular, the disjointness condition that prevents the `Max` problem of section 4.5 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 $get : C \dot{\rightarrow} A$ and $put : A \times C \dot{\rightarrow} C$ that obey the laws,

$$\begin{array}{lll}
put\ (get\ c, c) = \bot \text{ or } c & \forall c \in C & \textit{GetPut} \\
get\ (put\ (a, c)) = \bot \text{ or } a & \forall a \in A, c \in C & \textit{PutGet}
\end{array}$$

Taking $C = e$ and $A = (\mathbb{N}, e)$ gives $\texttt{exp}_{rs}$ and $\texttt{unexp}_{rs}$ 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,

$$\texttt{unexp}_{rs}\ (\texttt{exp}_{rs}\ e)\ e = e$$

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, $(e/p_i) \bullet p_i = e$. □

The PutGet *Law*

The *PutGet* law states that whenever it is well-defined,

$$\texttt{exp}_{rs}\ (\texttt{unexp}_{rs}\ (j, e')\ e) = (j, e')$$

Expanding the definitions gives that,

$$\begin{gathered}(i, (((e/p_j) \cup (e'/p'_j)) \bullet p_j/p_i) \bullet p'_i) = (j, e') \\ \text{for } i = \min\{i | ((e/p_j) \cup (e'/p'_j)) \bullet p_j \geq p_i\}_i\end{gathered}$$

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 4.5—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 $rs = p_1 \rightarrow p'_1, ..., p_n \rightarrow p'_n$ states that $p_i \vee p_j = \bot$ 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:

$$\begin{gathered}(i, (((e/p_j) \cup (e'/p'_j)) \bullet p_j/p_i) \bullet p'_i) = (j, e') \\ \text{for } i = \min\{i | ((e/p_j) \cup (e'/p'_j)) \bullet p_j \geq p_i\}_i\end{gathered}$$

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 $\gamma_1 \bullet p_j/p_i$ exists, so $p_i \vee p_j$ exists. On the other hand, if $p_i \vee p_j$ exists for some $i < j$, then e and e' can be chosen such that $(\gamma_1 \bullet p_j)/p_i$ 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 4.5 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.

4.6.2 *Desugar and Resugar are Inverses*

We show that `desugar` and `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* (`Tag` *o e*)*.*

Definition 3. *A* core term *is a term that contains no construct C that appears in the outermost position of any* LHS *of the rulelist.*

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

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

Proof. By induction over the term. □

Further, `desugar` and `resugar` are idempotent over core and surface terms, respectively.

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

Proof. By induction over the term. □

Theorem 2. *Assume that the lens laws hold for all transformations. Then for all surface terms* e*,* $\texttt{desugar}_{rs}\ e = e'$ *implies* $\texttt{resugar}_{rs}\ e' = e$*. And for all core terms* e'*,* $\texttt{resugar}_{rs}\ e' = e$ *implies* $\texttt{desugar}_{rs}\ e = e'$*.*

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

In the first case,

$$\begin{aligned}
& \texttt{Res}\ (\texttt{Des}\ (C\ e_1,\ ...,\ e_n)) & \\
= \ & \texttt{Res}\ (\texttt{Des}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ e')) & \\
& \quad \text{when } \texttt{E}\ (C\ e_1,\ ...,\ e_n) = (i, e') & \\
& \quad \text{and where } \gamma \text{ stands in for } (C\ e_1,\ ...,\ e_n) & \\
= \ & \texttt{Res}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ (\texttt{Des}\ e')) & \\
= \ & \texttt{U}\ (i, \texttt{Res Des}\ e')\ (C\ e_1,\ ...,\ e_n) & \\
= \ & \texttt{U}\ (i, e')\ (C\ e_1,\ ...,\ e_n) & \text{(by I.H.)} \\
= \ & (C\ e_1,\ ...,\ e_n) & \text{(by \textit{GetPut})}
\end{aligned}$$

In the second case,

$$\begin{aligned}
& \texttt{Des}\ (\texttt{Res}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ e')) & \\
= \ & \texttt{Des}\ (\texttt{U}\ (i, \texttt{Res}\ e')\ \gamma) & \\
= \ & \texttt{Des}\ (C\ e_1,\ ...,\ e_n) & \text{(using w.f.)} \\
& \quad \text{when } \texttt{U}\ (i, \texttt{Res}\ e')\ \gamma = (C\ e_1,\ ...,\ e_n) & \\
= \ & \texttt{Des}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ (\texttt{Res}\ e')) & \text{(by \textit{PutGet})} \\
= \ & (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ (\texttt{Des}\ (\texttt{Res}\ e'))) & \\
= \ & (\texttt{Tag}\ (\texttt{Head}\ i\ \gamma)\ e') & \text{(by I.H.)}
\end{aligned}$$

□

4.6.3 *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:*

- $\texttt{e->c}\ (\texttt{c->e}\ c) = c$ *for all c*
- $\texttt{s->e}\ (\texttt{e->s}\ e) = e$ *for all e*
- $(\texttt{c->e}\ c)$ *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 $\mathbb{D}(s') = c$. If s' occurs in the surface sequence, then resugaring must have succeeded with $\mathbb{R}(c) = s'$. Thus we simply need to show that $\mathbb{D}(\mathbb{R}(c)) = c$ for all terms c in the core language, i.e., that

$$\texttt{e->c}\ (\texttt{desugar}_{rs}\ (\texttt{s->e}\ (\texttt{e->s}\ (\texttt{resugar}_{rs}\ (\texttt{c->e}\ c))))) = c$$

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 purposefully 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 $\texttt{resugar}_{rs}$ 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.

4.6.4 *Machine-Checking Proofs*

We have made substantial progress formalizing our transformation system in the Coq proof assistant [61]. 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 4.5) that pass the first static check obey the lens laws.

This formalization helped us pin down the (sometimes subtle) well-formedness criteria of section 4.5.

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 becomes non-compositional. For instance, $\gamma \bullet [p_1, ..., p_n]$ is not a function of $\gamma \bullet p_1, ..., \gamma \bullet p_n$ when $p_1, ..., p_n$ contain ellipses.

4.7 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* [11] 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 [45]—based on A-normalization [18]—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 [10], or ways of emulating them (as discussed by Pettyjohn, et al. [45]), 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 (`racket-lang.org`), Pyret (`pyret.org`), and PLT Redex [16] (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 [11], it is much weaker than ours (e.g., 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.

4.8 EVALUATION

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.

4.8.1 *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 [35]. All of these behave exactly as one might expect other than `Letrec` and `Automaton`, discussed below.

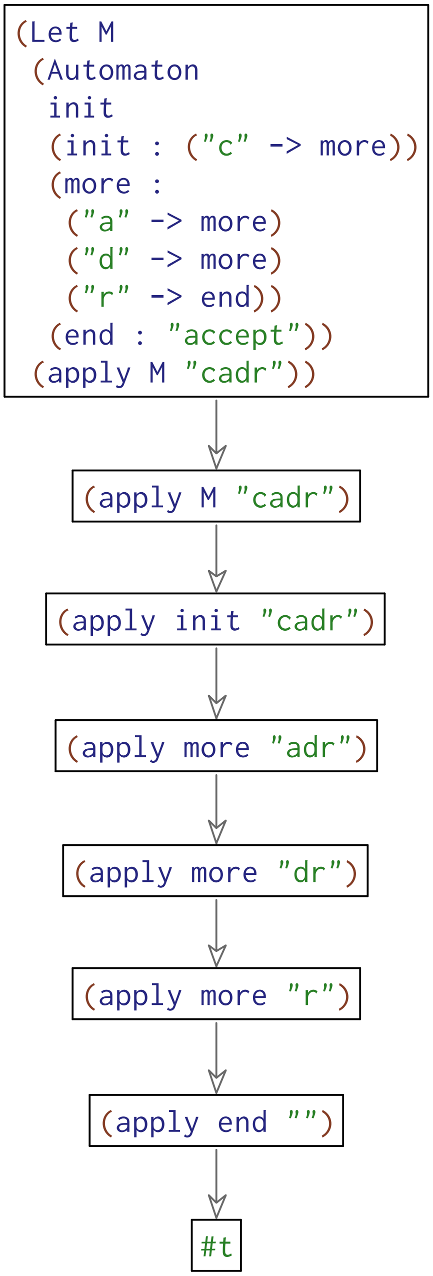
The `Letrec` sugar does not show any intermediate steps in which some but not all branches have been evaluated; thus the surface evaluation shows the branches all evaluating in one step. For instance, `(letrec ((x y) (y 2)) (+ x y))` steps directly to `(+ 2 2)`. Though this initially surprised us, it 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.

4.8.2 *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 current continuation. Using it, we can define a `return` sugar that returns early from a function:

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



```
(Let M
 (Automaton
  init
  (init : ("c" -> more))
  (more :
   ("a" -> more)
   ("d" -> more)
   ("r" -> end))
  (end : "accept"))
 (apply M "cadr"))
```

```
(apply M "cadr")
```

```
(apply init "cadr")
```

```
(apply more "adr")
```

```
(apply more "dr")
```

```
(apply more "r")
```

```
(apply end "")
```

```
#t
```

Figure 3: Automaton macro execution example

```
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.

4.8.3 *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 paper's authors, 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 builtin 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 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.

ast *Node*	*Description*	*Implemented?*
fun	function declaration	yes
when	one-arm conditional	yes
if	multi-arm conditional	yes
cases	multi-arm conditional	yes
cases-else	multi-arm conditional	yes
for	generalized looping construct	yes
op	binary operators	yes
not	negation	yes
paren	grouping construct	yes
left-app	infix notation	yes
list	list expressions	yes
dot	indirect field lookup	yes
colon	direct field lookup	yes
(currying syntax)	allowed in `fun` and `op`	yes
graph	create cyclic data	no
datatype	datatype declarations	no

Figure 4: Syntactic sugar in normal-mode Pyret

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 (the `s` terms are source locations, used for error-reporting):

```
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) --→ 6
```

The core evaluation sequence reveals why:

```
  1.["_plus"](2.["_plus"](3))
⟶ <func>(2.["_plus"](3))
⟶ <func>(<func>(3))
⟶ <func>(5)
⟶ 6
```

(`<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 `_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

```
Op(s, "+", x, y) ->
  Block(s,
  [ Let(s, Bind(s, "temp", ABlank),
           Obj(s, [Field(s, Str(s, "left"), x),
                   Field(s, Str(s, "right"), y)]))
  , App(s, Bracket(s, Bracket(s, Id(s, "temp"),
                                 Str(s, "left")),
                      Str(s, "_plus")),
           [Bracket(s, Id(s, "temp"),
                       Str(s, "right"))])]);
```

Figure 5: Alternate desugaring of addition

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 `{left: x, right: y}`, and then computes `temp.left._plus(temp.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) --→ 1 + 5 --→ 6
```

4.9 related work

There is a long history of trying 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 [26]. 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 [28]. 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. [12]. 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. [63] formalize the concept of tracking the origins of terms within term rewriting systems (which in their case represent the *evaluator*, not the *syntactic sugar* as in our case). They go on to show various applications, including visualizing program execution, implementing debugger breakpoints, and locating the sources of errors. Their work does not involve the use of syntactic sugar, however, while our work hinges on the interplay between syntactic sugar and evaluation. Nevertheless, we have adopted their notion of origin tracking for our transformations.

Krishnamurthi, et al. [36] 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 [14].

Clements [10, page 53] 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 paper. 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 [17] 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 similiar vein, Lorenzen and Erdweg [38] give a method for ensuring the type soundness of syntactic extensions by lowering author-provided typing rules for the surface language onto the core language's type system (and automatically verifying that soundness is entailed). Thus their work does for type checking what ours does for evaluation: it provides a surface type checker guaranteed to be sound with respect to the core language, while ours produces a surface evaluator guaranteed to emulate the core language.

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 [60]). 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 [44], or (ii) the surface information sought is more limited than the reduction sequence we provide (in the same ways as for debuggers for optimizing compilers, as described earlier). Applying our results to this area is future work.

5

RESUGARING SCOPE

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 α-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) [52].

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, refactorers 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 [6] for overlaying binding arrows on the source). However, this approach is far from ideal. It requires tools to be able to desugar programs and to resolve binding in the core language. This is an intimate level of knowledge of a language, though: syntactic sugar is supposed to be an abstraction, so external tools should ideally be unaware that a language even *has* syntactic sugar. Additionally, this approach fails completely if the source program cannot be desugared because it is incomplete or syntactically invalid (as programs are most of the time while 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 (whether type-checking rules or the evaluation rules themselves), though of course the mechanics of doing so will depend heavily on the semantics itself.

The intended application of this work is as follows:

1. Begin with a core language with known scoping rules, and a set of pattern-based desugaring rules. (We give a formal description of scope in section 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 [27, 59], type systems [38], and formal semantics [17]. 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 generalization, and sections 5.6.1 and 5.6.3 provide examples. (While the generalization 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:

$$\begin{array}{lll} e & ::= & (\texttt{Let}\ x^{\mathrm{D}}\ e_1\ e_2) \quad \text{“Let } x^{\mathrm{D}} \text{ equal } e_1 \text{ in } e_2\text{”} \\ & | & \ldots \end{array}$$

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

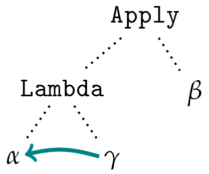
This Let may be desugared to Apply and Lambda by the following desugaring rule, which we will write using s-expressions:

$$(\texttt{Let}\ \alpha\ \beta\ \gamma) \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ \alpha\ \gamma)\ \beta)$$

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:

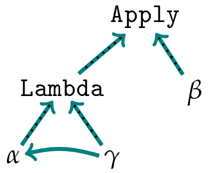
(RHS)



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 (γ). 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:

(RHS)

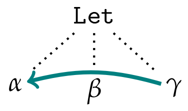


In general, the meaning of the arrows is that a variable declaration is in scope at every part of the program which has a (directed) path to it. (In the case of variable shadowing, the outer declaration is in scope at the inner declaration, which in turn is in scope at some region; references in this 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 γ to α in the rule's RHS, there must also be a path from γ to α in the LHS. This gives:

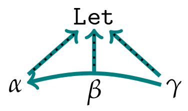
(LHS)



In English, this arrow says that the variable declared at α is in scope at the Let's body γ, as expected.

There are still some missing arrows, however: there should be down arrows to indicate that any declaration in scope at the Let should also be in scope at its children. These can be inferred in a similar way: whenever there is a path from the root to a 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:

(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:

$$\begin{array}{lcll}
e & ::= & (\texttt{Let*}\ b\ e) & \text{“Let-bind } b \text{ in } e\text{”} \\
 & | & \ldots & \\
b & ::= & (\texttt{Bind}\ x^{\mathrm{D}}\ e\ b) & \text{“Bind } x^{\mathrm{D}} \text{ to } e\text{, and bind } b\text{”} \\
 & | & \texttt{EndBinds} & \text{“No more bindings”}
\end{array}$$

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.

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

Let* can then be implemented with two desugaring rules:

$$(\texttt{Let*}\ (\texttt{Bind}\ \alpha\ \beta\ \gamma)\ \delta) \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ \alpha\ (\texttt{Let*}\ \gamma\ \delta))\ \beta)$$

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

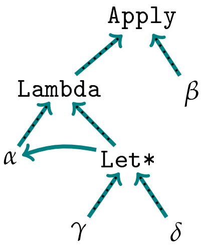
These rules would, for example, make the following transformation:

$$\begin{array}{l}
(\texttt{Let*}\ (\texttt{Bind}\ x^{\mathrm{D}}\ 1\ (\texttt{Bind}\ y^{\mathrm{D}}\ 2\ \texttt{EndBinds}))\ (\texttt{Plus}\ x^{\mathrm{R}}\ y^{\mathrm{R}})) \\
\quad \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ x^{\mathrm{D}}\ (\texttt{Apply}\ (\texttt{Lambda}\ y^{\mathrm{D}}\ (\texttt{Plus}\ x^{\mathrm{R}}\ y^{\mathrm{R}}))\ 2))\ 1)
\end{array}$$

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:

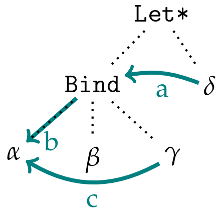
(RHS)



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 γ to α and from δ to α, the same must be true in the LHS (labeling the arrows for reference):

(LHS)



Notice that we drew the path from δ 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:

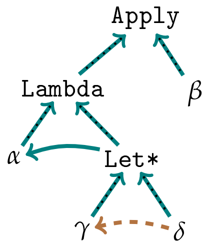
a. Declarations from a Let*'s binding list are visible in its body.

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.

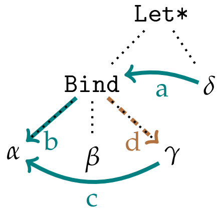
This information can now be applied to fill in the previously incomplete RHS picture. Arrow (a) represents a fact about the scoping of *every* Let*, so it must also apply in the RHS (highlighting it orange/dashed for exposition):

(RHS)



Adding this arrow introduces a path from δ to γ, however, that needs to be reflected back at the LHS!

(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.

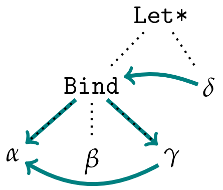
We make the meaning of arrows precise in section 5.3.2.

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.

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: $\delta \leq \texttt{Bind} \leq \gamma \leq \alpha$

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 3.3.1, 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.)

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

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:

constructor C	$::=$	*name*	syntactic construct name
term e	$::=$	*value*	primitive value
	$\vert$	$(C\ e_1 \ldots e_n)$	AST node
	$\vert$	x_i^{R}	variable reference
	$\vert$	x_i^{D}	variable declaration

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 $(C\ e_1 \ldots e_n)$ when there is no danger of ambiguity.

By the last assumption above, we do not support lists $[e_1 \ldots e_n]$ (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 perorder. 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. 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_i^{R} is* in scope of *a declaration x_j^{D} when $x_i^{\mathrm{R}} \leq x_j^{\mathrm{D}}$.*

Definition 7. *A declaration x_i^{D} is* more specific than *another x_j^{D} when $x_i^{\mathrm{D}} \leq x_j^{\mathrm{D}}$.*

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:*

$$x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \;\triangleq\; x^{\mathrm{D}} \in \min\{x_i^{\mathrm{D}} \mid x^{\mathrm{R}} \leq x_i^{\mathrm{D}}\}$$

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

$$\min S \;\triangleq\; \{a \in S \mid \nexists b \in S.\ b \leq a \text{ and } a \not\leq b\}$$

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

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

Definition 11 (Shadowing)**.** *A declaration* shadows *the most specific declarations that it has the same name as and is more specific than. Formally, x_i^{D} shadows x_j^{D} when:*

$$x_i^{\mathrm{D}} \mapsto x_j^{\mathrm{D}} \;\triangleq\; x_j^{\mathrm{D}} \in \min\{x_k^{\mathrm{D}} \mid i \neq k \text{ and } x_i^{\mathrm{D}} \leq x_k^{\mathrm{D}}\}$$

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" [19] obeys our scope-as-a-preorder definitions, for an appropriate choice of preorder ($\leq$). Finally, 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_j^{D} shadows x_i^{D} (x_j^{D} is the shadowing declaration and x_i^{D} is the shadowed declaration), and x_k^{R} is in scope of x_j^{D}. By definition, x_k^{R} will be bound by $\min\{x_l^{\text{D}} \mid x_k^{\text{R}} \leq x_l^{\text{D}}\}$. But $x_k^{\text{R}} \leq x_j^{\text{D}} \leq x_i^{\text{D}}$, so x_i^{D} cannot be in this set, and x_k^{R} cannot be bound by x_i^{D}. □

5.3.4 *Relationship to "Binding as Sets of Scopes"*

Scope-as-a-preorder aligns with the notion of scope expressed by [19]. 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" [19, pp. 3].

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

$$\begin{array}{l} x_i^{\text{R}} \leq x_i^{\text{R}} \\ x^{\text{R}} \leq y^{\text{D}} \text{ iff } \mathit{scope\text{-}set}(x^{\text{R}}) \supseteq \mathit{scope\text{-}set}(y^{\text{D}}) \\ x_i^{\text{D}} \leq y_j^{\text{D}} \text{ iff } \mathit{scope\text{-}set}(x_i^{\text{D}}) \supseteq \mathit{scope\text{-}set}(y_j^{\text{D}}) \end{array}$$

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 $\mathcal{S}(x^{\text{D}})$—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^{D} when $x^{\text{R}} \in \mathcal{S}(x^{\text{D}})$.
- One declaration x^{D} is *more specific than* another y^{D} when $\mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}})$.

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*.

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. To begin with, either form can be converted to the other.

***conv1*: FROM SCOPE-AS-A-PREORDER TO SCOPE-AS-SETS** Given a preorder $(\leq)$, define its conversion into a scope-as-sets function $\mathcal{S}$ by:

$$\mathcal{S}(y^{\text{D}}) \triangleq \{x^{\text{R}} \mid x^{\text{R}} \leq y^{\text{D}}\} \cup \{x^{\text{D}} \mid x^{\text{D}} \leq y^{\text{D}}\}$$

***conv2*: FROM SCOPE-AS-SETS TO SCOPE-AS-A-PREORDER** Given a scope-as-sets function $\mathcal{S}$, define its conversion into a preorder $(\leq)$ by letting $(\leq)$ be the least relation such that:

$$\begin{array}{lll} x_i^{\text{R}} \leq x_i^{\text{R}} & & \\ x^{\text{R}} \leq y^{\text{D}} & \text{when} & x^{\text{R}} \in \mathcal{S}(y^{\text{D}}) \\ x^{\text{D}} \leq y^{\text{D}} & \text{when} & \mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}}) \end{array}$$

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

Lemma 5 (Binding Preservation)**.** *Both Conv1 and Conv2 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:

$$\begin{array}{ll} & x^{\text{R}} \text{ in scope of } y^{\text{D}} \text{ (under SAS)} \\ \text{iff} & x^{\text{R}} \in \mathcal{S}(y^{\text{D}}) \\ \text{iff} & x^{\text{R}} \leq y^{\text{D}} \\ \text{iff} & x^{\text{R}} \text{ in scope of } y^{\text{D}} \text{ (under SAP)} \end{array}$$

And under both conversions, "more specific than" is preserved:

$$\begin{array}{ll} & x^{\text{D}} \text{ more specific than } y^{\text{D}} \text{ (under SAS)} \\ \text{iff} & \mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}}) \\ \text{iff} & x^{\text{D}} \leq y^{\text{D}} \\ \text{iff} & x^{\text{D}} \text{ more specific than } y^{\text{D}} \text{ (under SAP)} \end{array}$$

□

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 $Conv1(Conv2(\mathcal{S}))$, 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* $(\leq)$, $Conv2(Conv1(\leq)) = (\leq)$

Proof.

$$\begin{array}{lll} x_i^{\text{R}} \leq x_j^{\text{R}} & \text{when} & i = j \\ x^{\text{R}} \leq y^{\text{D}} & \text{when} & x^{\text{R}} \in \mathcal{S}(y^{\text{D}}) \\ & & \text{iff } x^{\text{R}} \in \{x^{\text{R}} \mid x^{\text{R}} \leq y^{\text{D}}\} \\ & & \text{iff } x^{\text{R}} \leq y^{\text{D}} \\ x^{\text{D}} \leq y^{\text{D}} & \text{when} & \mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}}) \\ & & \text{iff } x^{\text{D}} \leq y^{\text{D}} \end{array}$$

□

And they're inverses in the other direction when $\mathcal{S}$ is in a *normal form*. Specifically, we will say that a scope-as-sets function $\mathcal{S}$ is in *normal form* when:

$$\begin{array}{ll} \text{(i)} & \forall x^{\text{D}}.\ \mathcal{S}(x^{\text{D}}) \text{ contains only variables} \\ \text{(ii)} & \forall x^{\text{D}}.\ \forall y^{\text{D}}.\ x^{\text{D}} \in \mathcal{S}(y^{\text{D}}) \text{ iff } \mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}}) \end{array}$$

Lemma 7 (Inverses2)**.** *For any scope-as-sets function $\mathcal{S}$ in normal form, $Conv1(Conv2(\mathcal{S})) = \mathcal{S}$.*

Proof.

$$\begin{array}{lr} \mathcal{S}(y^{\text{D}}) & \\ = \{x^{\text{R}} \mid x^{\text{R}} \leq y^{\text{D}}\} \cup \{x^{\text{D}} \mid x^{\text{D}} \leq y^{\text{D}}\} & (\mathit{Conv1}) \\ = \{x^{\text{R}} \mid x^{\text{R}} \in \mathcal{S}(y^{\text{D}})\} \cup & \\ \quad \{x^{\text{D}} \mid \mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}})\} & (\mathit{Conv2}) \\ = \{x^{\text{R}} \mid x^{\text{R}} \in \mathcal{S}(y^{\text{D}})\} \cup \{x^{\text{D}} \mid x^{\text{D}} \in \mathcal{S}(y^{\text{D}})\} & \text{(by (ii))} \\ = \mathcal{S}(y^{\text{D}}) & \text{(by (i))} \end{array}$$

□

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

$$Norm(\mathcal{S}) = Conv1(Conv2(\mathcal{S}))$$

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

Proof. The first requirement—that the range of $\mathcal{S}$ only contains variables—is immediately fulfilled by *Conv1*. The second requirement follows from the definitions of *Conv1* and *Conv2*:
$x^{\text{D}} \in \mathcal{S}(y^{\text{D}})$ iff $x^{\text{D}} \leq y^{\text{D}}$ iff $\mathcal{S}(x^{\text{D}}) \subseteq \mathcal{S}(y^{\text{D}})$ □

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. *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 *scoping rules* that, given a term, determine a preorder over its variables.

5.4.1 *Scoping Rules: Simplified*

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

As an example, take the term $(\texttt{Lambda}\ x_1^{\mathrm{D}}\ (\texttt{Plus}\ x_2^{\mathrm{R}}\ 3))$. To find the bindings of this term, we must know the scoping rules for Lambda and Plus. A sensible rule for Plus is that a term $(\texttt{Plus}\ \alpha\ \beta)$ has preorder $\alpha \leq (\texttt{Plus}\ \alpha\ \beta)$ and $\beta \leq (\texttt{Plus}\ \alpha\ \beta)$, meaning that whatever a 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 Lambda is that a term $(\texttt{Lambda}\ \alpha\ \beta)$ has preorder $(\beta \leq \alpha \leq \texttt{Lambda})$, meaning that whatever a Lambda term is in scope of, its children are too, and that β (its body) is in scope of α (its declaration). Put together, and applied to the example term, these rules give that:

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

has preorder:

$$x_2^{\mathrm{R}}, 3 \leq \texttt{Plus} \leq x_1^{\mathrm{D}} \leq \texttt{Lambda}$$

Thus $x_2^{\mathrm{R}} \mapsto x_1^{\mathrm{D}}$ by definition 8, as it should be.

5.4.2 *A Problem*

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

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

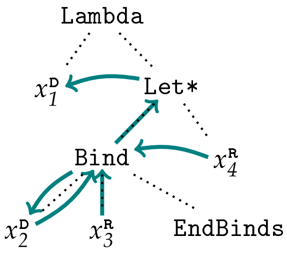
What will these scoping rules look like? Whatever they are, they should cause x_2^{D} to shadow x_1^{D}, x_3^{R} to be bound by x_1^{D}, and x_4^{R} to be bound by x_2^{D}. Formally, we should have:

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

which implies that, at a minimum:

$$x_2^{\mathrm{D}} \leq x_1^{\mathrm{D}} \text{ and } x_3^{\mathrm{R}} \leq x_1^{\mathrm{D}} \text{ and } x_4^{\mathrm{R}} \leq x_2^{\mathrm{D}}$$

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



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

However, this puts x^{R}_3 in scope of x^{D}_2, and as a result, x^{R}_3 will be bound by x^{D}_2! The problem is that Bind is trying to provide x^{D}_2, to make it available in the body of Let*, but in doing so it incidentally makes it available in the Bind's definition (to x^{R}_3). 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 *imports* (i.e., requires) from the bindings it *exports* (i.e., provides). In the running example, for instance, the Bind *imports* x^{D}_1, and *exports* x^{D}_1 and x^{D}_2 (with x^{D}_2 shadowing x^{D}_1). We will call imports and exports *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 *ports* of the subterms of e. With this in mind, we offer four kinds of bindings:[1]

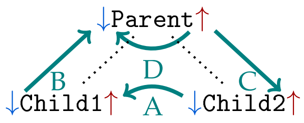
A. `bind i in j`: A term may make its i'th child's bindings available in its j'th child. If so, any declarations *exported* by child i will be *imported* by child j.

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

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

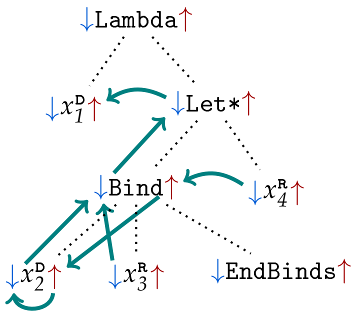
D. `re-export`: A term may take the declarations it *imported*, and *export* them. (This is not terribly useful in practice, but we offer it for completion.)

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



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

[1] There is a close analogy between ports and attributes in attribute grammars [31]: namely, imports are analogous to inherited attributes and exports are analogous to synthesized attributes. The paths between imports and exports that are allowed by our binding language (e.g., child export to parent export, but not child export to parent import) are precisely the relationships between inherited and synthesized attributes that are allowed in attribute grammars. Most algorithms for evaluating attribute grammars disallow cycles, however, while our preorders allow them.



$$\begin{aligned}
\downarrow\texttt{Let*} &\le \uparrow x^{\mathrm{D}}_{1} \\
\downarrow\texttt{Bind} &\le \downarrow\texttt{Let*} \\
\downarrow x^{\mathrm{R}}_{4} &\le \uparrow\texttt{Bind} \\
\downarrow x^{\mathrm{D}}_{2} &\le \downarrow\texttt{Bind} \\
\downarrow x^{\mathrm{R}}_{3} &\le \downarrow\texttt{Bind} \\
\uparrow x^{\mathrm{D}}_{2} &\le \downarrow x^{\mathrm{D}}_{2} \\
\uparrow\texttt{Bind} &\le \uparrow x^{\mathrm{D}}_{2}
\end{aligned}$$

Under this new preorder, $x^{\mathrm{R}}_{3} \mapsto x^{\mathrm{D}}_{1}$ and $x^{\mathrm{R}}_{4} \mapsto x^{\mathrm{D}}_{2}$ as desired.

5.4.4 *Scoping Rules: Unsimplified*

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

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

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

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

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

A set of *scope rules* Σ 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 Σ *is a partial map from term constructors C of arity n to binary relations over* $\{1, ..., n, \textsc{r}^{\uparrow}, \textsc{r}^{\downarrow}\}$, *such that:*

- *The relation is transitive.*
- $\textsc{r}^{\uparrow}$ *is a least element (* $\nexists a.\ (a, \textsc{r}^{\uparrow}) \in \Sigma[C]$*)*
- $\textsc{r}^{\downarrow}$ *is a greatest element (* $\nexists a.\ (\textsc{r}^{\downarrow}, a) \in \Sigma[C]$*)*

Here i represents the ith child term, $\textsc{r}^{\uparrow}$ represents the parent term's exports, and $\textsc{r}^{\downarrow}$ represents the parent term's imports. We will call pairs in the relation (e.g., $(1, \textsc{r}^{\downarrow})$) *facts*, and will equate them with their description in our binding language (so that $(1, \textsc{r}^{\downarrow}) = \texttt{import}\ 1$). 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 $\exists e'.\ (a = \downarrow e'$ or $a = \uparrow e')$ and $e' \sqsubseteq e$.

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

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

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 Σ is the set of judgements 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 Σ". A judgment is *well formed* when $a, b \sqsubseteq e$. (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. SD-Decl allows declarations to extend the current scope. S-Lift says that facts learned about a subterm remain true in the whole term.

$$\boxed{\Sigma, e \vdash a \leq b}$$

Declarative Rules

$$\text{SD-Trans}\ \frac{\Sigma, e \vdash a \leq b \qquad \Sigma, e \vdash b \leq c}{\Sigma, e \vdash a \leq c}$$

$$\text{SD-Import}\ \frac{\texttt{import}\ i \in \Sigma[P]}{\Sigma, (C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \downarrow (C\ t_1 \ldots t_n)}$$

$$\text{SD-Export}\ \frac{\texttt{export}\ i \in \Sigma[P]}{\Sigma, (C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq \uparrow e_i}$$

$$\text{SD-Bind}\ \frac{\texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P]}{\Sigma, (C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \uparrow e_j}$$

Shared Rules (Declarative & Algorithmic)

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

$$\text{S-Lift}\ \frac{\Sigma, e_i \vdash a \leq b}{\Sigma, (C\ t_1 \ldots t_n) \vdash a \leq b}$$

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

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

Algorithmic Rules

$$\text{SA-Import}\ \frac{\Sigma, e_i \vdash a \leq \downarrow e_i \qquad \texttt{import}\ i \in \Sigma[P]}{\Sigma, (C\ t_1 \ldots t_n) \vdash a \leq \downarrow (C\ t_1 \ldots t_n)}$$

$$\text{SA-Export}\ \frac{\texttt{export}\ i \in \Sigma[P] \qquad \Sigma, e_i \vdash \uparrow e_i \leq a}{\Sigma, (C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq a}$$

$$\text{SA-Bind}\ \frac{\Sigma, e_i \vdash a \leq \downarrow e_i \qquad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \qquad \Sigma, e_j \vdash \uparrow e_j \leq b}{\Sigma, (C\ t_1 \ldots t_n) \vdash a \leq b}$$

Figure 6: Scope Checking

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 Σ 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):

left \ right	re-export	export	import	bind	lift
re-export	✗	✗	✗	✗	✗
export	✗	✗	✓	✓	✓
import	✗	✗	✗	✗	✗
bind	✗	✗	✓	✓	✓
lift	✗	✗	✓	✓	✓

□

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:

$$\boxed{\Sigma, e \vdash x \leq y} \qquad \text{S-Var}\,\frac{\Sigma, e \vdash \downarrow x_i \leq \downarrow y_j}{\Sigma, e \vdash x_i \leq y_j}$$

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

$$\boxed{\Sigma, e \vdash x \mapsto y} \qquad \text{S-Bound}\,\frac{x^{\mathrm{D}} \in \min\{x_i^{\mathrm{D}} \mid \Sigma, e \vdash x^{\mathrm{R}} \leq x_i^{\mathrm{D}}\}}{\Sigma, e \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}}}$$

$$\text{S-Shadow}\,\frac{x_j^{\mathrm{D}} \in \min\{x_k^{\mathrm{D}} \mid i \neq k \text{ and } \Sigma, e \vdash x_i^{\mathrm{D}} \leq x_k^{\mathrm{D}}\}}{\Sigma, e \vdash x_i^{\mathrm{D}} \mapsto x_j^{\mathrm{D}}}$$

These definitions form a scope preorder:

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

Proof sketch. The relation is a preorder by the derivation rules S-Refl1, S-Refl2, and SD-Trans. We must also show that references are least. Suppose instead that $x_i \leq x_j^{\mathrm{R}}$ for some $i \neq j$. Then $\downarrow x_i \leq \downarrow x_j^{\mathrm{R}}$ (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 α-equivalence. Two terms are α-equivalent if (i) each term is "well-bound"; (ii) they have the same "shape" (i.e., they are identical ignoring their variable names); and (iii) for every binding $x^{\mathrm{R}} \mapsto x^{\mathrm{D}}$ in one term, an analogous binding exists in the same location in the other term. To formalize what "same location" means, we will use a *join* operator ($e \bowtie e'$) that checks that e and e' have the same shape and finds a bijection between their variable occurrences as a witness to this fact:

$$\begin{array}{lll}
x_i^{\mathrm{D}} & \bowtie y_j^{\mathrm{D}} & = \{x_i^{\mathrm{D}} \leftrightarrow y_j^{\mathrm{D}}\} \\
x_i^{\mathrm{R}} & \bowtie y_j^{\mathrm{R}} & = \{x_i^{\mathrm{R}} \leftrightarrow y_j^{\mathrm{R}}\} \\
\mathit{const} & \bowtie \mathit{const} & = \emptyset \\
(C\ e_1 \ldots e_n) & \bowtie (C\ e'_1 \ldots e'_n) & = \bigcup_{i \in 1..n} e_i \bowtie e'_i \\
e & \bowtie e' & = \textsc{undefined} \quad \text{otherwise}
\end{array}$$

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

Definition 14 (Conflicting Declarations)**.** *Two variable declarations* x_i^{D} *and* x_j^{D} conflict *in a term e when:*

$$\textit{S-Conflict}\ \frac{\begin{array}{c}\Sigma, e \vdash a \le x_i^{\mathrm{D}} \quad \Sigma, e \vdash a \le x_j^{\mathrm{D}} \\ \min_{\Sigma,e}\{x_i^{\mathrm{D}}, x_j^{\mathrm{D}}\} = \{x_i^{\mathrm{D}}, x_j^{\mathrm{D}}\}\end{array}}{\Sigma, e \vdash x_i^{\mathrm{D}} \ \mathit{conflicts}\ x_j^{\mathrm{D}}}$$

(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 Σ when every reference is bound by exactly one declaration, and there are no conflicting declarations:

$$\text{S-WB}\ \frac{\begin{array}{c}\forall x^{\mathrm{R}} \in e.\ \exists! x^{\mathrm{D}} \in e.\ \Sigma, e \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ \nexists x_i^{\mathrm{D}}, x_j^{\mathrm{D}} \in e.\ \Sigma, e \vdash x_i^{\mathrm{D}}\ \mathit{conflicts}\ x_j^{\mathrm{D}}\end{array}}{\Sigma \vdash \textsc{wb}\ e}$$

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

Definition 15 (α-equivalence)**.**

$$\textit{S-}\alpha\textit{-Eqv}\ \frac{\begin{array}{c}\Sigma \vdash \textsc{wb}\ e \quad \Sigma \vdash \textsc{wb}\ e' \quad e \bowtie e' = \psi \\ \forall x^{\mathrm{R}}, x^{\mathrm{D}}.\ \Sigma, e \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}}\ \mathit{iff}\ \Sigma, e' \vdash \psi(x^{\mathrm{R}}) \mapsto \psi(x^{\mathrm{D}})\end{array}}{\Sigma \vdash e =_\alpha e'}$$

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

In section 5.6.5, 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 α-equivalence. More precisely, given a core language with scoping rules Σ_{core}, and a desugaring $\mathbb{D}$, our algorithm finds scoping rules Σ_{surf} that preserves α-equivalence (theorem 8), so that:

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

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

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:

$$\begin{array}{llll} \text{pattern } p & ::= & \mathit{value} & \text{primitive value} \\ & | & (C\ p_1 \ldots p_n) & \text{AST node} \\ & | & x_i^{\mathrm{R}} & \text{variable reference} \\ & | & x_i^{\mathrm{D}} & \text{variable declaration} \end{array}$$

Furthermore, desugaring must obey the requirements of section 3.3.1.

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 3.3, and as is done by Scheme-style `syntax-rules` macros [30].

In general, a rewrite will look like:

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

where p_0 and p are patterns, and $p[e_1, ..., e_n]$ denotes replacing the n pattern variables of pattern p with terms $e_1, ..., e_n$. (In section 5.2, p was called the LHS, and p' the 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 (`Lambda` x^{D} α), 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. Specif-

```
$inferScope(\Sigma,\ \{p_i \Rightarrow p'_i\}_{i \in 1..n})$  $\triangleq$  Let $\Sigma_{surf} = solve(\Sigma,\ \bigcup_{i \in 1..n} genConstrs(p_i \Rightarrow p'_i))$
    $checkScope(\Sigma_{surf},\ \{p_i \Rightarrow p'_i\}_{i \in 1..n})$
    Return $\Sigma_{surf}$

$genConstrs(p \Rightarrow p')$  $\triangleq$  $\{genConstr(a \le b,\ p \Rightarrow p')\}_{a,b \in H}$
    where $H = pattern\text{-}vars(p) \cup pattern\text{-}vars(p') \cup \{\textsc{r}\}$

$genConstr(a \le b,\ p \Rightarrow p')$  $\triangleq$  $(genConj(a \le b,\ p), genConj(a \le b,\ p'))$
    where $(p, q)$ means a constraint "$p$ iff $q$"

$genConj(a \le b,\ p)$  $\triangleq$  Smallest $\Sigma_{surf}$ such that $\Sigma, p \vdash a \le b$.
    (To compute this, take the premises of the (unique)
    derivation of $\Sigma, p \vdash a \le b$ using the
    Algorithmic Scope Checking rules (fig. 6).)

$solve(\Sigma_{core},\ constraints)$  $\triangleq$  Initialize $\Sigma_{surf}$ = $\Sigma_{core}$
    Until fixpoint:
        • If a fact $F$ in a constraint is in $\Sigma_{surf}$:
              Delete $F$ from the constraint
        • If one side of a constraint is empty:
              Delete the constraint
              Add the other side to $\Sigma_{surf}$
                  (maintaining transitive closure)
        • If any fact in $\Sigma_{surf}$ is in the complement of $\Sigma_{core}$:
              ERROR: Reject this sugar
    Return $\Sigma_{surf}$

$checkScope(\Sigma,\ \{p_i \Rightarrow p'_i\}_{i \in 1..n})$  $\triangleq$  For each rule $p \Rightarrow p'$:
    Assert that if $\Sigma, p \vdash a \le b$ and $a \in p'$ then $b \in p'$
        (otherwise ERROR)
    Assert that each reference $x^{\textsc{r}} \in p'$ is bound by a
        unique declaration $x^{\textsc{d}} \in p'$
```

Figure 7: Scope Inference Algorithm

ically, fix a rewrite rule $p \Rightarrow p'$. It is important that this rewrite does not change the binding of any variable *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 root (i.e., the whole term). For example, in $p_0[p[e_1, ..., e_n]]$, α_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)**.**
$\Sigma \vdash p \cong p'$ means that $\forall a, b \in \{\alpha_1, ..., \alpha_n, \text{R}\}$.

$$\Sigma, p \vdash a \leq b \textit{ iff } \Sigma, p' \vdash a \leq b$$

where R *("root") stands in for p or p', as appropriate, and omitted port signs are determined by what our binding language allows:*

$$\begin{array}{lcl}
\Sigma, p \vdash \alpha_i \leq \alpha_j & \triangleq & \Sigma, p \vdash \downarrow\alpha_i \leq \uparrow\alpha_j \\
\Sigma, p \vdash \alpha_i \leq \text{R} & \triangleq & \Sigma, p \vdash \downarrow\alpha_i \leq \downarrow p \\
\Sigma, p \vdash \text{R} \leq \alpha_j & \triangleq & \Sigma, p \vdash \uparrow p \leq \uparrow\alpha_j \\
\Sigma, p \vdash \text{R} \leq \text{R} & \triangleq & \Sigma, p \vdash \uparrow p \leq \downarrow p
\end{array}$$

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 Σ if $\forall\, a, b \sqsubseteq p_0, e_1, ..., e_n$ (i.e., each of a and b lies in one of $p_0, e_1, ..., e_n$):*

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

Lemma 11 (Scope-equivalent patterns preserve scope)**.** *If $\Sigma \vdash p \cong p'$, 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 $\downarrow e_1, \uparrow e_1, ..., \downarrow e_n, \uparrow e_n, \downarrow p, \uparrow p$. The fact that $\Sigma \vdash p \cong p'$ means that these subpaths can be converted to subpaths in p', bounded instead by $\downarrow e_1, \uparrow e_1, ..., \downarrow e_n, \uparrow e_n, \downarrow p', \uparrow p'$. Replace these subpaths. Now the whole path goes from a to b in $p_0[p'[e_1, ..., e_n]]$. □

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:

$$f_1 \wedge f_2 \wedge ... f_n \;\; \texttt{iff} \;\; f'_1 \wedge f'_2 \wedge ... f'_m$$

where each f_i is a fact (e.g. `bind 2 in 1` $\in \Sigma[Let]$). This constraint is found by stating that the premises of the derivation of $\Sigma, p \vdash a \leq b$ 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:

$$(\texttt{Let}\ \alpha\ \beta\ \gamma) \Rightarrow (\texttt{Apply}\ (\texttt{Lambda}\ \alpha\ \gamma)\ \beta)$$

One of the necessary constraints says that:

$$\begin{array}{rl} & \Sigma, (\texttt{Let}\ \alpha\ \beta\ \gamma) \vdash \alpha \leq \beta \\ \text{iff} & \Sigma, (\texttt{Apply}\ (\texttt{Lambda}\ \alpha\ \gamma)\ \beta) \vdash \alpha \leq \beta \end{array}$$

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:

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

Since the boundary has size four (α, β, γ, and R), continuing this way leads to a total of $4^2 = 16$ constraints:

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

We have just described how to generate constraints—covering the *gen* functions in fig. 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):

$$(\texttt{LambdaFF}\;\texttt{Flip}\,\alpha_1\,\alpha_2) \Rightarrow (\texttt{Lambda}\;\alpha_1\,\alpha_2)$$

$$(\texttt{LambdaFF}\;\texttt{Flop}\,\alpha_1\,\alpha_2) \Rightarrow (\texttt{Lambda}\;\alpha_2\,\alpha_1)$$

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 $\texttt{bind}\ 2\ \texttt{in}\ 3 \in \Sigma[\text{LambdaFF}]$, and from the second rule, we can learn that $\texttt{bind}\ 3\ \texttt{in}\ 2 \in \Sigma[\text{LambdaFF}]$. Applying either of these facts to the *other* rule gives that $\texttt{bind}\ 2\ \texttt{in}\ 1 \in \Sigma[\text{Lambda}]$: the *body* of the Lambda is in scope at its *parameter*! This contradicts the known signature for Lambda (we know that $\texttt{bind}\ 2\ \texttt{in}\ 1 \notin \Sigma[\text{Lambda}]$), 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 *checkScope* in fig. 7:

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

- A sugar cannot delete a pattern variable that might contain a bound declaration. For instance, it could not rewrite $(\texttt{lambda}\ \alpha\,\beta)$ to β, because β might contain a reference bound by a declaration in α. 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 *hygiene*. However, these problematic sugars would not be considered unhygienic in the traditional sense.

Traditionally, research on hygiene has focused on preventing sugars from accidentally capturing user-defined references and vice versa. For instance, if a user binds x^{D}_i and then uses x^{R} inside a sugar, and the sugar locally binds x^{D}_j, then x^{R} should not be bound by x^{D}_j. These hygiene violations are called "introduced-binder" and "introduced-reference" violations, respectively. There are also more subtle violations in which desugaring makes observations about declaration equality [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 α-equivalent, then they will still be α-equivalent after a desugaring $\mathbb{D}$:

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

(Recall from definition 15 that α-equivalence is parameterized by Σ. Therefore, in the above antecedent and consequent, α-equivalence is respectively defined by Σ_{surf} and Σ_{core}.)

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

Most hygiene papers don't mention this criterion for a simple reason: $=_\alpha$ is not defined on their surface language, so they cannot even state the requirement. Recent exceptions to this rule [27, 59] 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 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 $\mathbb{D}(e_1) =_\alpha \mathbb{D}(e_2)$ in the core language, where $\mathbb{D}$ is a naive, scope-unaware desugaring. Then desugaring will preserve α-equivalence, despite not being hygienic in any way!

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 variables:

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

Proof. .[FILL] □

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 $\Sigma_{surf} = inferScope(\Sigma_{core}, \{p_i \Rightarrow p'_i\}_{i\in 1..n})$. 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, Σ_{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 $\Sigma_{surf} \vdash p_i \cong p'_i$ 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 Σ_{core}.

solve works with scope signatures Σ_{core} and Σ_{surf}, and a set of constraints $\mathbb{C}$:

$$\begin{array}{ccc} F'_{11} \wedge F'_{12} \wedge ... & \text{iff} & F''_{11} \wedge F''_{12} \wedge ... \\ F'_{21} \wedge F'_{22} \wedge ... & \text{iff} & F''_{21} \wedge F''_{22} \wedge ... \\ & ... & \end{array}$$

At any point during the evaluation of *solve*, let the *meaning* ϕ of Σ_{core}, Σ_{surf}, and Cs be:

$$\bigwedge_{F\in\Sigma_{surf}} F \wedge \bigwedge_{F\in\Sigma_{core}} F \wedge \bigwedge_{F\in\overline{\Sigma_{core}}} \neg F \wedge \bigwedge_i \left(\bigwedge_j F'_{ij} \text{ iff } \bigwedge_k F''_{ik} \right)$$

That is,

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

Upon initialization during *solve*, ϕ follows from Σ_{core} and $\mathbb{C}$. Furthermore, every step of *solve* maintains ϕ. There are three kinds of steps to consider, and each follows from a logical equivalence:

- "If a fact F in a constraint is in Σ_{surf}: Delete F from the constraint."

$$F \wedge (F \wedge F_2 \wedge ... \text{ iff } F'_1 \wedge F'_2 \wedge ...) \equiv F \wedge (F_2 \wedge ... \text{ iff } F'_1 \wedge F'_2 \wedge ...)$$

- "If one side of a constraint is empty: Delete the constraint; Add the other side to Σ_{surf} (maintaining trans. closure)."

$$(\mathit{true} \text{ iff } F_1 \wedge ... \wedge F_n) \equiv F_1 \wedge ... \wedge F_n$$

- "If any fact F in Σ_{surf} is in the complement of Σ_{core}: ERROR"

$$F \wedge \neg F \equiv \mathit{false}$$

Finally, when *solve* halts, it is because the facts in $\mathbb{C}$ and Σ_{surf} are disjoint, and every constraint in $\mathbb{C}$ has at least one fact on each side. Therefore it is valid to obtain a minimal set of facts by setting every fact in $\overline{\Sigma_{surf}}$ to *false*; doing so will satisfy $\mathbb{C}$ by making both sides of every remaining constraint false. Thus we have shown that the final Σ_{surf}:

$$\bigwedge_{F \in \Sigma_{surf}} F \wedge \bigwedge_{F \in \overline{\Sigma_{surf}}} \neg F$$

is a minimal solution to the initial Σ_{core} and $\mathbb{C}$:

$$\bigwedge_{F \in \Sigma_{core}} F \wedge \bigwedge_{F \in \overline{\Sigma_{core}}} \neg F \wedge \bigwedge_{i} \left(\bigwedge_{j} F'_{ij} \text{ iff } \bigwedge_{k} F''_{ik} \right)$$

Thus the surface language scoping rules Σ_{surf} found by *solve* are valid and minimal, given the core language scoping rules Σ_{core} plus the constraints $\mathbb{C}$. □

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

Proof. Induct on the number of rewrites performed. □

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

Lemma 13. *$inferScope(\Sigma, \mathbb{C})$ runs in time $O(size(\mathbb{C}) + \Sigma_{C \in surf} arity(C)^3)$.*

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

The cubic parameter is concerning, but not a problem in practice for a number of reasons. First, $arity(C)$ 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 [30]: 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.[2] After that, our tool successfully inferred scope for all of the sugars except for Scheme's `do`. In the rest of this section, we will describe many of these sugars in more detail, ending with `do`.

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

5.6.1 *Case Study: Pyret `for` Expressions*

Consider the "for expressions" of the Pyret language:

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

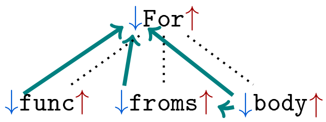
This example desugars into:

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

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

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

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



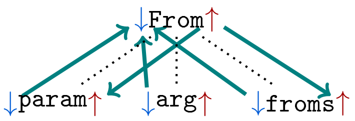
`import func` $\in \Sigma[$`For`$]$

`import froms` $\in \Sigma[$`For`$]$

`import body` $\in \Sigma[$`For`$]$

`bind froms in body` $\in \Sigma[$`For`$]$

[2] 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.



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

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 [46, section 3.11]: "List comprehensions satisfy these identities, which may be used as a translation into the kernel:"

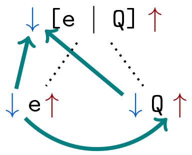
```
[ e | True ]            =  [e]                               (Base case)
[ e | q ]               =  [ e | q, True ]                   (Base case)
[ e | b, Q ]            =  if b then [ e | Q ] else []       Boolean guards
[ e | p <- l, Q ]       =  let ok p = [ e | Q ]
                          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

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

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

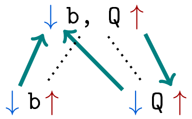
```
[n | n <- [1..], let d = divisors n, sum d == n]
```

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



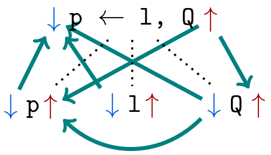
```
import e     ∈ Σ[ListComprehension]
import Q     ∈ Σ[ListComprehension]
bind Q in e  ∈ Σ[ListComprehension]
```

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



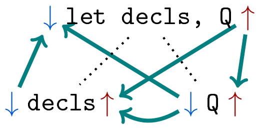
```
import b ∈ Σ[LC_Guard]
import Q ∈ Σ[LC_Guard]
export Q ∈ Σ[LC_Guard]
```

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



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

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



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

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 `(let ((x val) ...) body)`, and binds each declaration `x` to the corresponding `val` in `body`. The scope of this variant can be inferred similarly to how we inferred the scope of `let*` in section 5.2.

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

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

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

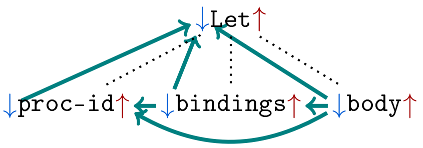
Named-let desugars by the rule:

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

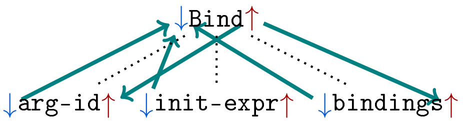
We will represent the AST for named-let expressions with the grammar:

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

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



```
import proc-id            ∈ Σ[Let]
import bindings           ∈ Σ[Let]
import body               ∈ Σ[Let]
bind proc-id in bindings  ∈ Σ[Let]
bind proc-id in body      ∈ Σ[Let]
bind bindings in body     ∈ Σ[Let]
```



```
import arg-id    ∈ Σ[Bind]
import init-expr ∈ Σ[Bind]
import bindings  ∈ Σ[Bind]
export arg-id    ∈ Σ[Bind]
export bindings  ∈ Σ[Bind]
```

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

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.

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

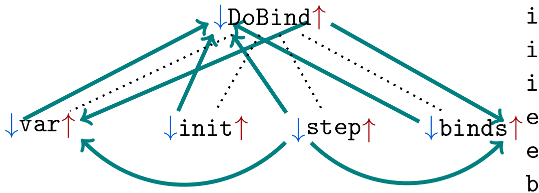
In general, `do` binds a list of variables [`sum` and `i`] to initial values [`0` and `0`], and then repeatedly evaluates the body of the loop [`(set! sum ...)`] and updates the variables according to optional step expressions [`(+ i 1)`] until a condition [`(= i 3)`] is met, at which point it evaluates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

The desugaring of `do` is given by [30, derived forms]:

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

```
       (loop init ...)))
     ((do "step" x) x)
     ((do "step" x y) y)))
```

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



`import var`	$\in \Sigma[\texttt{DoBind}]$
`import init`	$\in \Sigma[\texttt{DoBind}]$
`import step`	$\in \Sigma[\texttt{DoBind}]$
`import binds`	$\in \Sigma[\texttt{DoBind}]$
`export var`	$\in \Sigma[\texttt{DoBind}]$
`export binds`	$\in \Sigma[\texttt{DoBind}]$
`bind var in step`	$\in \Sigma[\texttt{DoBind}]$
`bind binds in step`	$\in \Sigma[\texttt{DoBind}]$

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.[3]

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.

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

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 $\mathbb{D}$ is hygienic when it preserves α-equivalence:

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

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

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

$$\begin{array}{lcl} \mathit{Used} & \triangleq & \mathit{range}(\phi) \\ \mathit{Unused} & \triangleq & \mathit{vars}(e) - \mathit{range}(\phi) \\ \mathit{Copied} & \triangleq & \mathit{domain}(\phi) \\ \mathit{New} & \triangleq & \mathit{vars}(\mathbb{D}(e)) - \mathit{domain}(\phi) \end{array}$$

(where $\mathit{vars}(e)$ 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, $\mathbb{D}$ must avoid variable capture; thus $\mathbb{D}(e)$ cannot contain bindings between new variables (introduced by $\mathbb{D}$) and copied variables (taken from e):

Property 1.

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

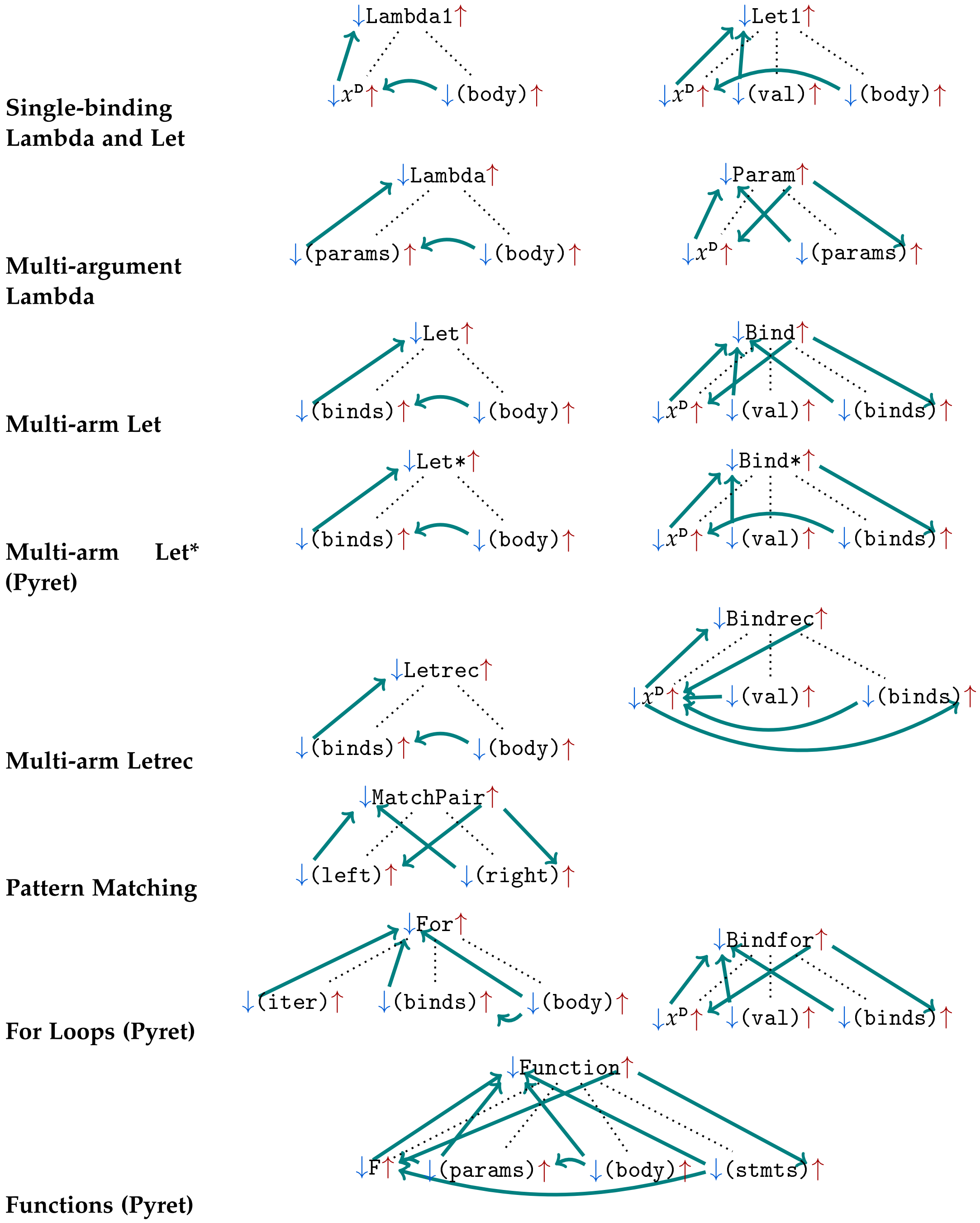


Figure 8: Catalog of Scoping Rules (Arrows that follow from transitivity omitted)

Second, $\mathbb{D}$ must preserve binding structure among the variables it copies:

Property 2 does not appear in prior work on hygiene because such work typically assumes that the binding structure of a surface term is defined by *the binding structure of its desugaring, causing this property to be true by definition.*

Property 2.

$$\begin{array}{l} \forall x^{\mathrm{R}}, x^{\mathrm{D}} \in \mathit{Copied}. \\ \Sigma, \mathbb{D}(e) \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \text{ \textit{iff} } \Sigma, e \vdash \phi(x^{\mathrm{R}}) \mapsto \phi(x^{\mathrm{D}}) \end{array}$$

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

Property 3.

$$\begin{array}{lll} \forall x^{\mathrm{R}} \in \mathit{Used}. & \forall x^{\mathrm{D}} \in \mathit{Unused}. & \Sigma, e \nvdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ \forall x^{\mathrm{R}} \in \mathit{New}. & \exists! x^{\mathrm{D}} \in \mathit{New}. & \Sigma, \mathbb{D}(e) \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \end{array}$$

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

Theorem 8 (Fundamental Hygiene Theorem)**.** *Let $\mathbb{D}$ be a desugaring function over terms with respect to scoping rules Σ that obeys the assumptions of section 3.3.1. Then $\mathbb{D}$ respects α-equivalence iff for every (well-bound) input term e (numbering by property number):*

$$\begin{array}{llll} 1. & \forall x^{\mathrm{R}} \in \mathit{Copied}. & \forall x^{\mathrm{D}} \in \mathit{New}. & \Sigma, \mathbb{D}(e) \nvdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ 1. & \forall x^{\mathrm{R}} \in \mathit{New}. & \forall x^{\mathrm{D}} \in \mathit{Copied}. & \Sigma, \mathbb{D}(e) \nvdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ 2. & \forall x^{\mathrm{R}} \in \mathit{Copied}. & \forall x^{\mathrm{D}} \in \mathit{Copied}. & \Sigma, \mathbb{D}(e) \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ & & & \text{\textit{iff} } \Sigma, e \vdash \phi(x^{\mathrm{R}}) \mapsto \phi(x^{\mathrm{D}}) \\ 3. & \forall x^{\mathrm{R}} \in \mathit{Used}. & \forall x^{\mathrm{D}} \in \mathit{Unused}. & \Sigma, e \nvdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \\ 3. & \forall x^{\mathrm{R}} \in \mathit{New}. & \exists! x^{\mathrm{D}} \in \mathit{New}. & \Sigma, \mathbb{D}(e) \vdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \end{array}$$

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

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

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

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

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

   $$\forall x^{\mathrm{R}} \in \mathit{Used}. \forall x^{\mathrm{D}} \in \mathit{Unused}. \Sigma, e \nvdash x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \tag{1}$$

2. ($x^{\mathrm{D}} \in \mathit{Used}$) α-varying x^{D} renames both x^{D} and x^{R} for every $x^{\mathrm{R}} \mapsto x^{\mathrm{D}}$ in e. Renaming x^{D} is problematic iff $\exists x_i^{\mathrm{R}}$. s.t. $\Sigma, \mathbb{D}(e) \vdash x_i^{\mathrm{R}} \mapsto x_i^{\mathrm{D}}$ where

$x_i^{\mathrm{D}} \in \mathit{image}(x^{\mathrm{D}})$ but $x_i^{\mathrm{R}} \notin \mathit{image}(x^{\mathrm{R}})$ for some $x^{\mathrm{R}} \mapsto x^{\mathrm{D}}$ (hence x_i^{R}'s binding will change when x^{D} is renamed). Thus:

$$\text{If } \Sigma, \mathbb{D}(e) \vdash x_i^{\mathrm{R}} \mapsto x_i^{\mathrm{D}} \text{ and } \phi(x_i^{\mathrm{D}}) \text{ exists, then } \phi(x_i^{\mathrm{R}}) \text{ exists and } \phi(x_i^{\mathrm{R}}) \mapsto \phi(x_i^{\mathrm{D}}) \tag{2}$$

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

a) ($x^{\mathrm{R}} \in \mathit{Unused}$) Renaming x^{R} in e is OK since it does not appear in $\mathbb{D}(e)$.

b) ($x^{\mathrm{R}} \in \mathit{Used}$) Consider some $x_i^{\mathrm{R}} \in \mathit{image}(x^{\mathrm{R}})$. Let $x_i^{\mathrm{R}} \mapsto x_i^{\mathrm{D}} \in \mathbb{D}(e)$ (it must be bound, since $\mathbb{D}(e)$ is well-bound). For the binding of x_i^{R} to not change when x^{D} (and x^{R}) is renamed, x_i^{D} must be in $\mathit{image}(x^{\mathrm{D}})$. Thus:

$$\text{If } \Sigma, \mathbb{D}(e) \vdash x_i^{\mathrm{R}} \mapsto x_i^{\mathrm{D}} \text{ and } \phi(x_i^{\mathrm{R}}) \text{ exists , then } \phi(x_i^{\mathrm{D}}) \text{ exists, and } \phi(x_i^{\mathrm{R}}) \mapsto \phi(x_i^{\mathrm{D}}) \tag{3}$$

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

$$\forall x^{\mathrm{R}} \in \mathit{New}.\ \exists! x^{\mathrm{D}} \in \mathit{New}.\ x^{\mathrm{R}} \mapsto x^{\mathrm{D}} \in \mathbb{D}(e) \tag{4}$$

This gives four equations that hold iff $\mathbb{D}$ 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 *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 *checkScope* checks.

5.8 HYGIENE FOR EVALUATION RESUGARING

We can also use this hygiene theorem to show how to make the resugaring system from the previous chapter (CONFECTION) hygienic, both during expansion and during unexpansion. Two things must be done. First, when CONFECTION desugars, it must (unsurprisingly) give newly introduced variables fresh names, and when resugaring it should account for these fresh names when matching. Second, this chapter's scope inference must be applied. This, unfortunately, disallows sugars that contain ellipses, although we are currently working to lift this restriction. Remember that scope inference serves a dual purpose: it not only infers scope rules, it also rejects bad sugars for which no valid scope rules exist.

While it may seem strange to use a scope inference process simply to make a system hygienic, keep in mind that we use a very strong definition of hygiene in this chapter. This definition refers to properties such as "desugaring may not cause a variable to become unbound" that are not even defined without the existence of surface scope rules. Presumably, a weaker version of hygiene could be achieved more easily (although it is not quite as simple as using an off-the-shelf hygienic expansion system, as it isn't immediately clear how that should interact with unexpansion).

With that said, we can state and prove the lemma:

Lemma 14. *Let $\mathcal{L}$ be a set of desugaring rules, and let Σ_{core} be scope rules for its core langauge. Suppose that:*

- *$\texttt{desugar}_{rs}$ gives fresh names to introduced variables (that is, variables on the* RHS*).*
- *$\texttt{resugar}_{rs}$ allows any variable name to match against an introduced variable.*
- *No rule in $\mathcal{L}$ drops a pattern variable.*
- *$\Sigma_{surf} = inferScope(\Sigma_{core}, \mathcal{L})$.*

Then both $\texttt{desugar}_{rs}$ and $\texttt{resugar}_{rs}$ preserve α-equivalence.

Proof. We will apply the hygiene theorem in both directions: both forward (during expansion) and in reverse (during unexpansion). There are two sets of requirements for the theorem: the assumptions about resugaring in section 5.5.1, and properties (1)–(3) in this section. We must show that both sets of requirements hold both forward and in reverse:

Assumptions about desugaring (section 5.5.1)

Assumption (1, forward) is an assumption of this lemma.
Assumption (1, reverse) is a basic well-formedness requirement (a pattern variable on the RHS but not the LHS does not make sense).
Assumption (2, forward) is assumed in section 4.5.
Assumption (2, reverse) is identical.
Assumption (3, forward) is also assumed in section 4.5.
Assumption (3, reverse) does not hold: rules' RHSs often *do* contain variables. Thus we must argue that unexpanding a rule whose RHS contains variables does not modify α-equivalence. Since the variables in the term being unexpanded were freshly named during desugaring, no other variables in the term could possibly be bound by or bound to them. Thus removing them preserves α-equivalence.
Assumption (4, forward) is an assumption of this lemma.
Assumption (4, reverse) is vacuously true, since the LHS is assumed to not contain variables (section 4.5).

Properties (1)–(3)

Property (1, forward) holds because we assume variables are given fresh names.
Property (1, reverse) is vacuously true because we assume that pattern variables are not dropped, so *Unused* is empty.
Property (2, forward) follows from our scope inference process: since desugaring preserves scope by theorem 7, bindings between variables must not change.
Property (2, reverse) also follows from our scope inference process, which shows that desugaring rules preserve scope. Preserving scope is a symmetric property, and will hold just as well if you run the rule in reverse.

> *Property (3, forward)* is checked by the *checkScope*(s)ubroutine of the *inferScope*(f)unction; if the check failed, then *inferScope*(w)ould have failed.
> *Property (3, reverse)*: consider the two equations of this property in turn. The first is equivalent to the first equation in Property (1, forward), which we already showed. The second is vacuously true (since *Unused* is empty), as well as following from the surface term being well-bound.

Thus the hygiene theorem applies in both directions, so both `desugar`$_{rs}$ and `resugar`$_{rs}$ are hygienic. □

5.9 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.

5.9.1 *Hygienic Expansion*

The real goal of hygienic expansion is to preserve α-equivalence: α-renaming a program should not change its meaning. Typically, however, α-equivalence is only *defined* for the core language. Thus, traditional approaches to hygiene have had to focus on avoiding specific issues like variable capture [33]. Recent work by [2] advances the theory by giving an algorithm-independent set of issues to avoid. However, even this work lacks the ground truth of α-equivalence preservation to base its claims on.

In contrast, [27] advocate that sugar specify the binding structure of the constructs they introduce, and build a system that does so. [59] follow with a more powerful system called Romeo based on the same approach. (We will discuss the binding languages used by these two tools in the next subsection.) Since we *infer* scope rules for the surface language, we can verify that desugaring preserves α-equivalence without requiring scope annotations on sugars.

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

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

5.9.2 *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 [19] in "Binding as Sets of Scopes". In fact, Flatt's notion of scope can be expressed as a preorder, as we show in section 5.3.4.

Scope-as-a-preorder and Flatt's Sets of Scopes were discovered independently: scope-as-a-preorder arose from some of the ideas from Romeo [59].

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

binding specification languages Our preorder-based binding specification language is novel, but similar in expressiveness to many others. It is perhaps most similar to [59]'s Romeo. The primary difference between the two is that Romeo has slightly more expressive power: given two declarations x_1^{D} and x_2^{D}, it is possible in Romeo for x_1^{D} to shadow x_2^{D} in one part of a term, but x_2^{D} to shadow x_1^{D} in a different part of a term.[4] 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, [56] present a semantics engineering workbench called Ott, which includes a comparable binding specification language. Like Romeo, Ott would allow two declarations to each shadow one another in different places. Furthermore, it gives additional power, 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".

[64] present a binding specification language called Unbound, which can be expressed using scope-as-a-preorder (and hence is no more expressive than it). They implement Unbound in Haskell, and give language-agnostic implementations of operations such as constructing and deconstructing terms, determining α-equivalence, and performing substitution. In Unbound, binding is specified via a set of *binding combinators*. These binding combinators can be expressed as a preorder.[5]

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

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

There are many more binding specification languages [1, 53, 34]. We have chosen what we believe to be a representative sample for comparison. We have shown that our binding specification language compares favorably in expressiveness, while simultaneously being simple enough to enable scope inference.

5.10 DISCUSSION AND CONCLUSION

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.

We see three clear directions in which to try to extend scope inference. First, support for ellipses in sugar definitions would make writing sugars easier. Second, allowing named imports and exports—à la Ott [56]—would make sugars like `do` inferable. Third, modules—à la Scope Graphs [42]—are necessary for inferring scope for modules and for classes. These last two changes are relatively straightforward extensions to our binding language, but research questions when applied to scope inference.

Bind P T	$\{\texttt{bind}\ 1\ \texttt{in}\ 2 \in \Sigma[\text{Bind}]\}$	Embed T	$\emptyset$
Rebind P P	$\{\texttt{bind}\ 1\ \texttt{in}\ 2 \in \Sigma[\text{Rebind}]\}$	Rec P	$\{\texttt{bind}\ 1\ \texttt{in}\ 1 \in \Sigma[\text{Rec}]\}$

SA → SD

$$\text{SA-Import}\,\dfrac{e_i \vdash a \leq {\downarrow} e_i \quad \texttt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\Longrightarrow \quad \text{SD-Trans}\,\dfrac{\text{S-Lift}\,\dfrac{e_i \vdash a \leq {\downarrow} e_i}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow} e_i} \quad \text{SD-Import}\,\dfrac{\texttt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash {\downarrow} e_i \leq {\downarrow}(C\ t_1 \ldots t_n)}}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\text{SA-Export}\,\dfrac{\texttt{export}\ i \in \Sigma[P] \quad e_i \vdash {\uparrow} e_i \leq a}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq a}$$

$$\Longrightarrow \quad \text{SD-Trans}\,\dfrac{\text{SD-Export}\,\dfrac{\texttt{export}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq {\uparrow} e_i} \quad \text{S-Lift}\,\dfrac{e_i \vdash {\uparrow} e_i \leq a}{(C\ t_1 \ldots t_n) \vdash {\uparrow} e_i \leq a}}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq a}$$

$$\text{SA-Bind}\,\dfrac{e_i \vdash a \leq {\downarrow} e_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad e_j \vdash {\uparrow} e_j \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b}$$

$$\Longrightarrow \quad \text{SD-Trans}^2\,\dfrac{\text{S-Lift}\,\dfrac{e_i \vdash a \leq {\downarrow} e_i}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow} e_i} \quad \text{SD-Bind}\,\dfrac{\texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash {\downarrow} e_i \leq {\uparrow} e_j} \quad \text{S-Lift}\,\dfrac{e_j \vdash {\uparrow} e_j \leq b}{(C\ t_1 \ldots t_n) \vdash {\uparrow} e_j \leq b}}{(C\ t_1 \ldots t_n) \vdash a \leq b}$$

Figure 9: Proof of theorem 6

SD → SA

$$\text{SD-Export}\,\frac{\texttt{export}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash \uparrow(C\ t_1 \ldots t_n) \leq \uparrow e_i} \Longrightarrow \text{SA-Export}\,\frac{\texttt{export}\ i \in \Sigma[P] \quad \text{S-Refl2}\,\frac{}{e_i \vdash \uparrow e_i \leq \uparrow e_i}}{(C\ t_1 \ldots t_n) \vdash \uparrow(C\ t_1 \ldots t_n) \leq \uparrow e_i}$$

$$\text{SD-Import}\,\frac{\texttt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \downarrow(C\ t_1 \ldots t_n)} \Longrightarrow \text{SA-Import}\,\frac{\texttt{import}\ i \in \Sigma[P] \quad \text{S-Refl1}\,\frac{}{e_i \vdash \downarrow e_i \leq \downarrow e_i}}{(C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \downarrow(C\ t_1 \ldots t_n)}$$

$$\text{SD-Trans}\,\frac{\text{S-Refl1}\,\frac{}{e \vdash \downarrow e \leq \downarrow e} \quad \frac{D}{e \vdash \downarrow e \leq a}}{e \vdash \downarrow e \leq a} \Longrightarrow \frac{D}{e \vdash \downarrow e \leq a} \qquad \text{(likewise for S-Refl2)}$$

$$\text{SD-Bind}\,\frac{\texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \uparrow e_j} \Longrightarrow \text{SA-Bind}\,\frac{\text{S-Refl1}\,\frac{}{e_i \vdash \downarrow e_i \leq \downarrow e_i} \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad \text{S-Refl2}\,\frac{}{e_j \vdash \uparrow e_j \leq \uparrow e_j}}{(C\ t_1 \ldots t_n) \vdash \downarrow e_i \leq \uparrow e_j}$$

$$\text{SD-Trans}\,\frac{\text{SA-Bind}\,\frac{e_i \vdash a \leq \downarrow e_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad e_j \vdash \uparrow e_j \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \quad \text{SA-Bind}\,\frac{e_j \vdash b \leq \downarrow e_j \quad \texttt{bind}\ k\ \texttt{in}\ j \in \Sigma[P] \quad e_k \vdash \uparrow e_k \leq c}{(C\ t_1 \ldots t_n) \vdash b \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\Longrightarrow \text{SA-Bind}\,\frac{e_i \vdash a \leq \downarrow e_i \quad \frac{\texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad \texttt{bind}\ k\ \texttt{in}\ j \in \Sigma[P]}{\texttt{bind}\ k\ \texttt{in}\ i \in \Sigma[P]} \quad e_k \vdash \uparrow e_k \leq c}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\text{SD-Trans}\,\frac{\text{SA-Bind}\,\frac{e_i \vdash a \leq \downarrow e_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad e_j \vdash \uparrow e_j \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \quad \text{S-Lift}\,\frac{e_j \vdash b \leq c}{(C\ t_1 \ldots t_n) \vdash b \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\Longrightarrow \text{SA-Bind}\,\frac{e_i \vdash a \leq \downarrow e_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad \text{SD-Trans}\,\frac{e_j \vdash \uparrow e_j \leq b \quad e_j \vdash b \leq c}{e_j \vdash \uparrow e_j \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

Figure 10: Proof of theorem 6 (continued)

SD → SA (cont.)

$$\text{SD-Trans}\ \dfrac{\text{S-Lift}\ \dfrac{e_i \vdash a \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \qquad \text{SA-Bind}\ \dfrac{e_i \vdash b \leq \downarrow e_i \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad e_j \vdash \uparrow e_j \leq c}{(C\ t_1 \ldots t_n) \vdash b \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\Longrightarrow \quad \text{SD-Bind}\ \dfrac{\text{SD-Trans}\ \dfrac{e_i \vdash a \leq b \quad e_i \vdash b \leq \downarrow e_i}{e_i \vdash a \leq \downarrow e_i} \quad \texttt{bind}\ j\ \texttt{in}\ i \in \Sigma[P] \quad e_j \vdash \uparrow e_j \leq c}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\text{SD-Trans}\ \dfrac{\text{S-Lift}\ \dfrac{e_i \vdash a \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \qquad \text{S-Lift}\ \dfrac{e_i \vdash b \leq c}{(C\ t_1 \ldots t_n) \vdash b \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\Longrightarrow \quad \text{S-Lift}\ \dfrac{\text{SD-Trans}\ \dfrac{e_i \vdash a \leq b \quad e_i \vdash b \leq c}{e_i \vdash a \leq c}}{(C\ t_1 \ldots t_n) \vdash a \leq c}$$

$$\text{SD-Trans}\ \dfrac{\text{SA-Export}\ \dfrac{\texttt{export}\ i \in \Sigma[P] \quad e_i \vdash \uparrow e_i \leq a}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq a} \qquad \text{SA-Import}\ \dfrac{e_i \vdash a \leq \downarrow e_i \quad \texttt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash a \leq \downarrow (C\ t_1 \ldots t_n)}}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq \downarrow (C\ t_1 \ldots t_n)}$$

$$\Longrightarrow \quad \text{S-ReExport}\ \dfrac{\dfrac{\texttt{export}\ i \in \Sigma[P] \quad \texttt{import}\ i \in \Sigma[P]}{\texttt{re-export} \in \Sigma[P]}}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq \downarrow (C\ t_1 \ldots t_n)}$$

$$\text{SD-Trans}\ \dfrac{\text{SA-Export}\ \dfrac{\texttt{export}\ i \in \Sigma[P] \quad e_i \vdash \uparrow e_i \leq a}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq a} \qquad \text{S-Lift}\ \dfrac{e_i \vdash a \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b}}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq b}$$

$$\Longrightarrow \quad \text{SA-Export}\ \dfrac{\texttt{export}\ i \in \Sigma[P] \quad \text{SD-Trans}\ \dfrac{e_i \vdash \uparrow e_i \leq a \quad e_i \vdash a \leq b}{e_i \vdash \uparrow e_i \leq b}}{(C\ t_1 \ldots t_n) \vdash \uparrow (C\ t_1 \ldots t_n) \leq b}$$

Figure 11: Proof of theorem 6 (continued)

SD → SA (cont.)

$$\text{SD-Trans}\;\dfrac{\text{S-Lift}\;\dfrac{e_i \vdash a \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \qquad \text{SA-Import}\;\dfrac{e_i \vdash b \leq {\downarrow} e_i \quad \mathtt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash b \leq {\downarrow}(C\ t_1 \ldots t_n)}}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\Longrightarrow \quad \text{SA-Import}\;\dfrac{\text{SD-Trans}\;\dfrac{e_i \vdash a \leq b \quad e_i \vdash b \leq {\downarrow} e_i}{e_i \vdash a \leq {\downarrow} e_i} \qquad \mathtt{import}\ i \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\text{SD-Trans}\;\dfrac{\text{SA-Bind}\;\dfrac{e_i \vdash a \leq {\downarrow} e_i \quad \mathtt{bind}\ j\ \mathtt{in}\ i \in \Sigma[P] \quad e_j \vdash {\uparrow} e_j \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b} \qquad \text{SA-Import}\;\dfrac{e_j \vdash b \leq {\downarrow} e_j \quad \mathtt{import}\ j \in \Sigma[P]}{(C\ t_1 \ldots t_n) \vdash b \leq {\downarrow}(C\ t_1 \ldots t_n)}}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\Longrightarrow \quad \text{SA-Import}\;\dfrac{e_i \vdash a \leq {\downarrow} e_i \qquad \dfrac{\mathtt{bind}\ j\ \mathtt{in}\ i \in \Sigma[P] \quad \mathtt{import}\ j \in \Sigma[P]}{\mathtt{import}\ i \in \Sigma[P]}}{(C\ t_1 \ldots t_n) \vdash a \leq {\downarrow}(C\ t_1 \ldots t_n)}$$

$$\text{SD-Trans}\;\dfrac{\text{SA-Export}\;\dfrac{\mathtt{export}\ i \in \Sigma[P] \quad e_i \vdash {\uparrow} e_i \leq a}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq a} \qquad \text{SA-Bind}\;\dfrac{e_i \vdash a \leq {\downarrow} e_i \quad \mathtt{bind}\ j\ \mathtt{in}\ i \in \Sigma[P] \quad e_j \vdash {\uparrow} e_j \leq b}{(C\ t_1 \ldots t_n) \vdash a \leq b}}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq b}$$

$$\Longrightarrow \quad \text{SA-Export}\;\dfrac{\dfrac{\mathtt{export}\ i \in \Sigma[P] \quad \mathtt{bind}\ j\ \mathtt{in}\ i \in \Sigma[P]}{\mathtt{export}\ j \in \Sigma[P]} \qquad e_j \vdash {\uparrow} e_j \leq b}{(C\ t_1 \ldots t_n) \vdash {\uparrow}(C\ t_1 \ldots t_n) \leq b}$$

Figure 12: Proof of theorem 6 (continued)

6

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)[CITE: to appear].

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 documentation 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 language'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 language 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 $\overline{I_{core}}$ are the core language type rules and $\overline{I_{surf}}$ are the resugared surface type rules, then

Goal 1.

$$\overline{I_{surf}} \Vdash \Gamma \vdash e : t \quad \textit{iff} \quad \overline{I_{core}} \Vdash \Gamma \vdash \mathbb{D}(e) : t$$

where $\overline{I} \Vdash J$ means that judgment J is provable by inference rules $\overline{I}$, and $\mathbb{D}(e)$ means the desugaring of expression e.

Notice the assumption implicit in this equation: the right-hand-side says t, rather than $\mathbb{D}(t)$. 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 to future work.

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

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

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

$$\begin{array}{rl} & \overline{I_{surf}} \Vdash \Gamma \vdash \texttt{(}\alpha \texttt{ and } \beta\texttt{)} : \texttt{t} \\ \text{iff} & \overline{I_{core}} \Vdash \Gamma \vdash \mathbb{D}(\alpha \texttt{ and } \beta) : \texttt{t} \end{array}$$

Expanding out the sugar:

$$\begin{array}{rl} & \overline{I_{surf}} \Vdash \Gamma \vdash \texttt{(}\alpha \texttt{ and } \beta\texttt{)} : \texttt{t} \\ \text{iff} & \overline{I_{core}} \Vdash \Gamma \vdash \texttt{(if } \mathbb{D}(\alpha) \texttt{ then } \mathbb{D}(\beta) \texttt{ else false)} : \texttt{t} \end{array}$$

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

$$\texttt{t-and}^{\rightarrow}\ \frac{\Gamma \vdash \texttt{(if } \mathbb{D}(\alpha) \texttt{ then } \mathbb{D}(\beta) \texttt{ else false)} : \texttt{t}}{\Gamma \vdash \texttt{(}\alpha \texttt{ and } \beta\texttt{)} : \texttt{t}}$$

and perhaps also its converse:

$$\texttt{t-and}^{\leftarrow}\ \frac{\Gamma \vdash \texttt{(}\alpha \texttt{ and } \beta\texttt{)} : \texttt{t}}{\Gamma \vdash \texttt{(if } \mathbb{D}(\alpha) \texttt{ then } \mathbb{D}(\beta) \texttt{ else false)} : \texttt{t}}$$

The rule t-and$^{\rightarrow}$can be read as "to prove that (α and β) has type *t* under type environment Γ in the surface language, prove that its desugaring has type *t* under Γ 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 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-and$^{\leftarrow}$is actually redundant: since t-and$^{\rightarrow}$is the only type rule mentioning and, t-and$^{\leftarrow}$is admissible. Therefore, we only need t-and$^{\rightarrow}$.

In this particular case, we have added only the rule t-and$^{\rightarrow}$, but in general we would add one such rule for each sugar. This could be called the *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 true and false has the type derivation:

$$\texttt{t-and}^{\rightarrow}\ \dfrac{\text{t-if}\ \dfrac{\dfrac{}{\vdash \texttt{true} : \texttt{Bool}} \quad \dfrac{}{\vdash \texttt{false} : \texttt{Bool}} \quad \dfrac{}{\vdash \texttt{false} : \texttt{Bool}}}{\vdash \texttt{(if true then false else false)} : \texttt{Bool}}}{\vdash \texttt{(true and false)} : \texttt{Bool}}$$

Since the extension type rules (like t-and$^{\rightarrow}$) always succeed, any type errors will be found in the core language. For example, if the first argument to and was not a boolean, this will be discovered by the t-if rule, not by the t-and$^{\rightarrow}$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 and will share a common form. It will always follow the template:

$$\texttt{t-and}^{\rightarrow}\,\dfrac{\text{t-if}\,\dfrac{\dfrac{D_\alpha}{\Gamma \vdash \alpha : \texttt{Bool}} \qquad \dfrac{D_\beta}{\Gamma \vdash \beta : \texttt{Bool}} \qquad \text{t-false}\,\dfrac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}}{\Gamma \vdash (\texttt{if } \alpha \texttt{ then } \beta \texttt{ else false}) : \texttt{Bool}}}{\Gamma \vdash (\alpha \texttt{ and } \beta) : \texttt{Bool}}$$

where the sub-derivations D_α and D_β depend on α and β. Notice that the rest of the derivation is constant: *every* type-derivation of α `and` β 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 `and`:

$$\text{t-and}\,\dfrac{\Gamma \vdash \alpha : \texttt{Bool} \qquad \Gamma \vdash \beta : \texttt{Bool}}{\Gamma \vdash (\alpha \texttt{ and } \beta) : \texttt{Bool}}$$

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 `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 `or` sugar:

The `let` in the desugaring of `or` prevents the duplicate evaluation of α.

$$\alpha \texttt{ or } \beta \;\Rightarrow\; \texttt{let x = } \alpha \texttt{ in if x then x else } \beta$$

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:

$$\Gamma \vdash (\texttt{let x = } \alpha \texttt{ in if x then x else } \beta) : \texttt{t}$$

We can begin by applying the core type rules, obtaining a partial derivation, shown at the top of section 6.2. However, the core type rules (unsurprisingly) cannot prove the judgments about pattern variables (marked with $\boxed{?}$). 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`:

$$\text{t-premise}\,\dfrac{\text{fresh } x}{\Gamma \vdash \alpha : x}$$

(This rule will be generalized in section 6.4.2 and section 6.4.3.) 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 section 6.2.

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 section 6.2), then we solve these constraints (via an ordinary unification algorithm). Solving the constraints of section 6.2 gives the substitution $\{A = \texttt{Bool}, B = \texttt{Bool}\}$. Finally, gathering the premises and conclusion of the derivation and applying the substitution to them produces the type rule for `or`:

$$\text{t-or}\,\dfrac{\Gamma \vdash \alpha : \texttt{Bool} \qquad \Gamma \vdash \beta : \texttt{Bool}}{\Gamma \vdash \alpha \texttt{ or } \beta : \texttt{Bool}}$$

$$\text{t-let}\,\dfrac{\dfrac{\boxed{?}}{\Gamma \vdash \alpha : A} \qquad \text{t-if}\,\dfrac{\text{t-id}\,\dfrac{}{\Gamma, x : A \vdash \texttt{x} : A} \quad A = \texttt{Bool} \quad \text{t-id}\,\dfrac{}{\Gamma, x : A \vdash \texttt{x} : A} \quad \dfrac{\boxed{?}}{\Gamma, x : A \vdash \beta : B} \quad A = B}{\Gamma, x : A \vdash \texttt{if x then x else } \beta : B}}{\Gamma \vdash \texttt{let x = } \alpha \texttt{ in if x then x else } \beta : B}$$

$$\text{t-let}\,\dfrac{\text{t-prem.}\,\dfrac{}{\Gamma \vdash \alpha : A} \qquad \text{t-if}\,\dfrac{\text{t-id}\,\dfrac{}{\Gamma, x : A \vdash \texttt{x} : A} \quad A = \texttt{Bool} \quad \text{t-id}\,\dfrac{}{\Gamma, x : A \vdash \texttt{x} : A} \quad \text{t-prem.}\,\dfrac{}{\Gamma, x : A \vdash \beta : B} \quad A = B}{\Gamma, x : A \vdash \texttt{if x then x else } \beta : B}}{\Gamma \vdash \texttt{let x = } \alpha \texttt{ in if x then x else } \beta : B}$$

Figure 13: Derivation of `or`. Top: an incomplete derivation. Bottom: a complete derivation, using `t-premise`.

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. $\Gamma \vdash$ `(let x =` α `in if x then x else` β`)` $: t$
2. Search for a derivation of this judgment using the core language type rules plus the `t-premise` rule from above. Fail if no derivation, or if multiple derivations, are found. For example, this will find the derivation shown in section 6.2.
3. Gather the equality constraints from the derivation. Additionally, if multiple premises (i.e., judgments proved by the `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 `or`, there are two equality constraints: $A = \texttt{Bool}$ and $A = B$. The `t-premise` rule is used only once for α and once for β, so no additional constraints are needed. The solution is $\{A = \texttt{Bool}, B = \texttt{Bool}\}$.
4. Form a type rule whose premises are the judgments proved by `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 [20] (`racket-lang.org`), and makes use of the semantics engineering tool Redex [16]. *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 1 and goal 2 given these assumptions. Roughly speaking, these assumptions are:

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

NOTATION EXPLANATION

$$\begin{array}{rcll}
e & ::= & \mathit{value} & \text{(primitive value)} \\
 & \mid & x & \text{(variable)} \\
 & \mid & (C\ e_1 \ldots e_n) & \text{(AST node)} \\
p & ::= & \mathit{value} & \\
 & \mid & x & \\
 & \mid & (C\ p_1 \ldots p_n) & \\
 & \mid & \alpha & \text{(pattern variable)} \\
t & ::= & \mathit{type} & \text{(type)} \\
\Gamma & ::= & \cdot \mid \Gamma, x : t & \text{(type environment)} \\
J & ::= & \Gamma \vdash p : t & \text{(type judgment)} \\
I & ::= & J_1 \ldots J_n / J & \text{(inference rule)} \\
\overline{I} & ::= & I_1 \ldots I_n & \text{(set of inference rules)} \\
\gamma & ::= & \{\alpha \to e, \ldots\} & \text{(substitution)} \\
\mathcal{L} & ::= & \{p \Rightarrow p', \ldots\} & \text{(desugaring rules)}
\end{array}$$

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 α inside is a pattern variable; (ii) from the other perspective, p is an expression inside a type rule, in which α 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:

$\overline{I} \Vdash J$ to mean that judgment J is provable under inference rules $\overline{I}$ (i.e., there is a derivation that proves J).

$\overline{I} \Vdash J_1 \ldots J_n \to J$ to mean that there is a derivation that proves J with unproven leaves J_i.

$(\gamma \bullet p)$ to denote applying substitution γ to expression p. To substitute into a type judgment, substitute into its parts (likewise for type environments):

$$\gamma \bullet (\Gamma \vdash p : t) \quad = \quad (\gamma \bullet \Gamma) \vdash (\gamma \bullet p) : (\gamma \bullet t)$$

Figure 14: Notation explanation

- 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.

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:

$$\begin{array}{lcl}
\mathbb{D}(\mathit{value}) & = & \mathit{value} \\
\mathbb{D}(x) & = & x \\
\mathbb{D}(\gamma \bullet p) & = & (\mathbb{D}(\gamma)) \bullet (\mathcal{L}[p]) \\
 & & \text{if } p \text{ is in the surface language} \\
\mathbb{D}(C\, e_1 \dots e_n) & = & (C\, \mathbb{D}(e_1) \dots \mathbb{D}(e_n)) \\
 & & \text{if } C \text{ is in the core language}
\end{array}$$

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

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

$$\begin{array}{lcl}
\mathbb{D}(\Gamma \vdash p : t) & = & \Gamma \vdash \mathbb{D}(p) : t \\
\mathbb{D}(\{x \to p, \dots\}) & = & \{x \to \mathbb{D}(p), \dots\}
\end{array}$$

Unsurprisingly, substitution distributes over pattern-based desugaring:

Lemma 15 (Distributivity of Substitution and Desugaring)**.**

$$\mathbb{D}(\gamma \bullet J) = \mathbb{D}(\gamma) \bullet \mathbb{D}(J)$$

Proof. Let $J = \Gamma \vdash p : t$. By definition, $\mathbb{D}(\Gamma \vdash p : t) = \Gamma \vdash \mathbb{D}(p) : t$. Induct on p.

Base case: e is a primitive value:

$\mathbb{D}(\gamma \bullet \mathit{value}) = \mathit{value} = \mathbb{D}(\gamma) \bullet \mathit{value}$.

Base case: e is a variable: likewise.

Base case: e is a pattern variable α:

$\mathbb{D}(\gamma \bullet \alpha) = \mathbb{D}(\gamma[\alpha]) = (\mathbb{D}(\gamma))[\alpha] = \mathbb{D}(\gamma) \bullet \alpha = \mathbb{D}(\gamma) \bullet \mathbb{D}(\alpha)$.

Inductive case: e is a compound term $\{\alpha_1 \mapsto e_1, ...\} \bullet p$:

$$\begin{array}{cll}
 & \mathbb{D}(\gamma \bullet (\{\alpha_1 \mapsto e_1, ...\} \bullet p)) & \\
= & \mathbb{D}(\{\alpha_1 \mapsto (\gamma \bullet e_1), ...\} \bullet p) & \text{(substitution)} \\
= & \{\alpha_1 \mapsto \mathbb{D}(\gamma \bullet e_1), ...\} \bullet p' & \text{where } \mathcal{L}[p] = p' \\
= & \{\alpha_1 \mapsto (\mathbb{D}(\gamma) \bullet \mathbb{D}(e_1)), ...\} \bullet p' & \text{(I.H.)} \\
= & \mathbb{D}(\gamma) \bullet (\{\alpha_1 \mapsto \mathbb{D}(e_1), ...\} \bullet p') & \text{(substitution)} \\
= & \mathbb{D}(\gamma) \bullet \mathbb{D}(\{\alpha_1 \mapsto e_1, ...\} \bullet p) & \text{(desugar)}
\end{array}$$

□

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. That is:

Assumption 1 (Unique Desugaring)**.** *For every pair p_1 and p_2 of sugar* LHS*s, there are no substitutions γ_1 and γ_2 such that $\gamma_1 \bullet p_1 = \gamma_2 \bullet p_2$.*

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 $(\alpha$ `and` $\beta)$ must share a common template, and we implicitly assumed that this template could not depend on α or on β. 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 section 6.2) 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:*

$$\overline{I} \Vdash J_1 \ldots J_n \rightarrow J \quad \textit{iff} \quad \forall \gamma.\ \overline{I} \Vdash \gamma \bullet J_1 \ldots \gamma \bullet J_n \rightarrow \gamma \bullet J$$

Likewise for rules:

$$\overline{I} \Vdash J_1 \ldots J_n \,/\, J \quad \textit{iff} \quad \forall \gamma.\ \overline{I} \Vdash \gamma \bullet J_1 \ldots \gamma \bullet J_n \,/\, \gamma \bullet J$$

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:*

$$\textit{If } \overline{I} \Vdash J_1 \ldots J_n \rightarrow J \textit{ and } \forall i.\ \overline{I} \Vdash J_i\textit{, then } \overline{I} \Vdash J.$$

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 $\gamma \bullet J$ that can be proven, then the proof of $\gamma \bullet J$ *must* use $J_1 \ldots J_n \rightarrow J$, and thus prove as intermediate steps $\gamma \bullet J_i$ for each $i \in 1..n$. Formally, we define determinism as:

Definition 18 (Determinism)**.** *A set of inference rules* $\overline{I}$ *is* deterministic *when, for any concrete judgment* $\gamma \bullet J$*:*
If $\overline{I} \Vdash \gamma \bullet J$ *and* $\overline{I} \Vdash J_1 \ldots J_n \rightarrow J$*, then* $\overline{I} \Vdash \gamma \bullet J_i$ *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 $\overline{I_{core}}$ 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 16 (Determinism). *Suppose that at most one type rule in $\overline{I}$ ever applies to a concrete judgment J. Then $\overline{I}$ is deterministic.*

Proof. Suppose that $\overline{I} \Vdash \gamma \bullet J$ and $\overline{I} \Vdash J_1 \ldots J_n \rightarrow J$. We aim to show that $\overline{I} \Vdash \gamma \bullet J_i$ for each $i \in 1..n$.

Induct on the derivation $\overline{I} \Vdash J_1 \ldots J_n \rightarrow J$. Let the bottommost step in the derivation be $\overline{I} \Vdash J'_1 \ldots J'_m \;/\; J$, and call this rule R. By assumption 2 (substitution), $\overline{I} \Vdash \gamma \bullet J'_1 \ldots \gamma \bullet J'_m \;/\; \gamma \bullet J$. Since, by assumption 4 (unique-rule), only one rule can apply to the judgment $\gamma \bullet J$, no rule other than R may apply. Hence the derivation of $\gamma \bullet J$ *must* have $\overline{I} \Vdash \gamma \bullet J'_1 \ldots \gamma \bullet J'_m \;/\; \gamma \bullet J$ as the bottommost step. Thus for each $i \in 1..m$:

- $\overline{I} \Vdash \gamma \bullet J'_i$, and
- There is a subset $J_{i_1} \ldots J_{i_l}$ of $J_1 \ldots J_n$ such that $\overline{I} \Vdash J_{i_1} \ldots J_{i_l} \rightarrow J'_i$. Since each judgment $J_1 \ldots J_n$ must be used in the derivation $\overline{I} \Vdash J$, 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,
$\overline{I} \Vdash \gamma \bullet J_{i_1} \ldots \gamma \bullet J_{i_l}$. Since the union of these sets is $\gamma \bullet J_1 \ldots \gamma \bullet J_n$, 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 $\overline{I_{core}}$ 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 Resugaring*

Our final set of requirements is on the behavior of the type resugaring 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 $p_{\text{LHS}} \Rightarrow p_{\text{RHS}} \in \mathcal{L}$. Resugaring will search for a partial derivation of the sugar's RHS:

$$\overline{I_{core}} \Vdash J_1 \ldots J_n \rightarrow J_{\text{RHS}}$$

where $J_1 \ldots J_n$ are provable using the `t-premise` rule and J_{RHS} has the form $J_{\text{RHS}} = \Gamma \vdash p_{\text{RHS}} : t$. If such a derivation is found, and is unique, then we will write:

$$resugar(\overline{I_{core}}, p_{\text{LHS}} \Rightarrow p_{\text{RHS}}) = J_1 \ldots J_n / J_{\text{LHS}}$$

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 $\overline{I_{surf}}$. Therefore:

Assumption 5 (Resugaring). *Suppose that*
$resugar(\overline{I_{core}}, p_{\text{LHS}} \Rightarrow p_{\text{RHS}}) = J_1 \ldots J_n / (\Gamma \vdash p_{\text{LHS}} : t)$. Then:

$$\overline{I_{core}} \Vdash J_1 \ldots J_n \rightarrow \mathbb{D}(\Gamma \vdash p_{\text{LHS}} : t)$$

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 17 (Surface Determinism)**.** *If resugaring succeeds, then I_{surf} is* deterministic. *Repeating the definition of determinism, this means that:*

If $I_{surf} \Vdash \gamma \bullet J$ and $I_{surf} \Vdash J_1 \dots J_n \to J$, then $I_{surf} \Vdash \gamma \bullet J_i$ 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 γ_1 and γ_2 such that $\gamma_1 \bullet J_1 = \gamma_2 \bullet J_2 = J$. Thus $\gamma_1 \bullet e_1 = \gamma_2 \bullet e_2 = e$. However, this contradicts assumption 1 (unique-sugar). Thus at most one type rule in I_{surf} may apply to a concrete judgment.

Then, by lemma 16, I_{surf} is deterministic. □

6.3.4 *Main Theorem*

Given the requirements of this section, type resugaring obeys goal 1:

Theorem 9. *Grant assumptions 1–5 from this section, let $\mathcal{L} = p_{\text{LHS}} \Rightarrow_{\text{RHS}} \dots$, and suppose that*
$resugar(\overline{I_{core}}, p_{\text{LHS}} \Rightarrow_{\text{RHS}}) \cdots = \overline{I_{surf}}$. Then:

$$\overline{I_{surf}} \Vdash J_{surf} \textit{ iff } \overline{I_{core}} \Vdash \mathbb{D}(J_{surf})$$

Proof. Given in fig. 15. □

Furthermore, resugaring obeys goal 2, essentially by construction:

Lemma 18. *Resugaring obeys goal 2: type rules for surface constructs never mention core constructs.*

Proof. Let $resugar(I_{core}, p_{\text{LHS}} \Rightarrow p_{\text{RHS}}) = J_1 \dots J_n / J_{\text{LHS}}$ be any surface rule. We aim to show that $J_1 \dots J_n$ and J_{LHS} do not mention core constructs C. By assumption 5 (resugaring), $I_{core} \Vdash J_1 \dots J_n \to \mathbb{D}(J_{\text{LHS}})$, where $J_1 \dots J_n$ are all provable using `t-premise`. We gave the `t-premise` rule in section 6.2, and generalize it in section 6.4.3 and section 6.4.4. However, in *all* of its versions, the judgment must be over a surface term. Thus $J_1 \dots J_n$ do not mention core constructs.

Finally, the expression in J_{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. □

6.4 desugaring features

There are several important features of desugaring that make the above story more interesting. We describe them in this section.

Proof. Split on the "iff".

Forward implication ("soundness"). Induct on the derivation proving that $\overline{I_{surf}} \Vdash J_{surf}$. Let $J_1 \ldots J_n / J_0 = \mathit{resugar}(\overline{I_{core}}, \_)$ be the rule in I_{surf} used to prove J_{surf}, and let γ be the substitution such that $J_{surf} = \gamma \bullet J_0$. Then:

$$\begin{array}{lll}
 & \overline{I_{surf}} \Vdash J_{surf} & \text{assumption} \\
\text{iff} & \overline{I_{surf}} \Vdash \gamma \bullet J_0 & \text{equality} \\
\text{implies} & \overline{I_{surf}} \Vdash \gamma \bullet J_i \text{ for } i \in 1..n & \text{by lemma 17 (surface determinism)} \\
\text{implies} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma \bullet J_i) \text{ for } i \in 1..n & \text{inductive hypothesis}
\end{array}$$

Also:

$$\begin{array}{lll}
 & \overline{I_{core}} \Vdash J_1 \ldots J_n \;/\; \mathbb{D}(J_0) & \text{by assumption 5 (resugaring)} \\
\text{implies} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma) \bullet J_1 \ldots \mathbb{D}(\gamma) \bullet J_n \;/\; \mathbb{D}(\gamma) \bullet \mathbb{D}(J_0) & \text{by assumption 2 (substitution)} \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma) \bullet \mathbb{D}(J_1) \ldots \mathbb{D}(\gamma) \bullet \mathbb{D}(J_n) \;/\; \mathbb{D}(\gamma) \bullet \mathbb{D}(J_0) & \text{since } \mathbb{D}(J_i) = J_i \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma \bullet J_1) \ldots \mathbb{D}(\gamma \bullet J_n) \;/\; \mathbb{D}(\gamma \bullet J_0) & \text{by lemma 15 (distributivity)} \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma \bullet J_1) \ldots \mathbb{D}(\gamma \bullet J_n) \;/\; \mathbb{D}(J_{surf}) & \text{equality}
\end{array}$$

Thus $\overline{I_{core}} \Vdash \mathbb{D}(J_{surf})$ by assumption 3 (composition).

Reverse implication ("completeness"). Induct on the derivation proving that $\overline{I_{core}} \Vdash \mathbb{D}(J_{surf})$. Let $J_1 \ldots J_n / J_0 = \mathit{resugar}(\overline{I_{core}}, \_)$ be the rule in $\overline{I_{surf}}$ for the (outermost) sugar in J_{surf}'s expression, and let γ be the substitution such that $J_{surf} = \gamma \bullet J_0$. Then:

$$\begin{array}{lll}
 & \overline{I_{core}} \Vdash \mathbb{D}(J_{surf}) & \text{assumption} \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma \bullet J_0) & \text{equality} \\
\text{implies} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma) \bullet \mathbb{D}(J_0) & \text{by lemma 15 (distributivity)} \\
\text{Also:} & \overline{I_{core}} \Vdash J_1 \ldots J_n \to \mathbb{D}(J_0) & \text{by assumption 5 (resugaring)} \\
\text{implies} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma) \bullet J_1 \ldots \mathbb{D}(\gamma) \bullet J_n & \text{by corollary 2 (core determinism)} \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma) \bullet \mathbb{D}(J_1) \mathbb{D}(\gamma) \bullet \mathbb{D}(J_n) & \text{since } \mathbb{D}(J_i) = J_i \\
\text{iff} & \overline{I_{core}} \Vdash \mathbb{D}(\gamma \bullet J_1) \ldots \mathbb{D}(\gamma \bullet J_1) & \text{by lemma 15 (distributivity)} \\
\text{implies} & \overline{I_{surf}} \Vdash \gamma \bullet J_1 \ldots \gamma \bullet J_n & \text{inductive hypothesis}
\end{array}$$

Also:

$$\begin{array}{lll}
 & \overline{I_{surf}} \Vdash J_1 \ldots J_n \;/\; J_0 & \text{by assumption 5 (resugaring)} \\
\text{implies} & \overline{I_{surf}} \Vdash \gamma \bullet J_1 \ldots \gamma \bullet J_n \;/\; \gamma \bullet J_0 & \text{by assumption 2 (substitution)} \\
\text{iff} & \overline{I_{surf}} \Vdash \gamma \bullet J_1 \ldots \gamma \bullet J_n \;/\; \gamma \bullet J_{surf} & \text{equality}
\end{array}$$

Thus $\overline{I_{surf}} \Vdash J_{surf}$ by assumption 3 (composition). □

Figure 15: Proof of theorem 9.

6.4.1 *Calculating Types*

Consider the desugaring of `let` into the application of a `lambda`:

$$\mathtt{let}\ x\ \mathtt{=}\ \alpha\ \mathtt{in}\ \beta\ \Rightarrow\ \mathtt{(}\lambda\ x : \boxed{?}\mathtt{.}\ \beta\mathtt{)(}\alpha\mathtt{)}$$

What is the missing type? It needs to match the type of α, 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:

$$\mathtt{let}\ x\ \mathtt{=}\ \alpha\ \mathtt{in}\ \beta\ \Rightarrow\ \mathtt{calc\text{-}type}\ \alpha\ \mathtt{as}\ X\ \mathtt{in}\ \mathtt{(}\lambda\ x : X\mathtt{.}\ \beta\mathtt{)(}\alpha\mathtt{)}$$

This binds the type variable X to the type of α 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:

$$\mathtt{calc\text{-}type}\ p_1\ \mathtt{as}\ t\ \mathtt{in}\ p_2$$

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.[1] Notice that this requires 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:

$$\text{t-calc-type}\ \dfrac{\Gamma \vdash p_1 : t_1 \qquad t_1 = t \qquad \Gamma \vdash p_2 : t_2}{\Gamma \vdash \mathtt{(calc\text{-}type}\ p_1\ \mathtt{as}\ t\ \mathtt{in}\ p_2\mathtt{)} : t_2}$$

With this type rule, we can find a type derivation for `let`, shown in fig. 16. It leads to the type rule:

$$\text{t-let}\ \dfrac{\text{t-calc-type}\ \dfrac{\text{t-premise}\ \dfrac{}{\Gamma \vdash \alpha : \mathtt{A}} \qquad \mathtt{A\ =\ t} \qquad \text{t-app}\ \dfrac{\text{t-lambda}\ \dfrac{\text{t-premise}\ \dfrac{}{\Gamma, \mathtt{x} : \mathtt{t} \vdash \beta : \mathtt{D}}}{\Gamma \vdash \mathtt{(}\lambda\ \mathtt{x{:}t.}\ \ \beta\mathtt{)} : \mathtt{t} \to \mathtt{D}} \qquad \text{t-premise}\ \dfrac{}{\Gamma \vdash \alpha : \mathtt{t}}}{\Gamma \vdash \mathtt{(}\lambda\ \mathtt{x{:}t.}\ \ \beta\mathtt{)(}\alpha\mathtt{)} : \mathtt{D}}}{\Gamma \vdash\ \mathtt{(calc\text{-}type}\ \alpha\ \mathtt{as\ t\ in\ (}\lambda\ \mathtt{x{:}t.}\ \ \beta\mathtt{)}\ \alpha\mathtt{)} : \mathtt{D}}}{\Gamma \vdash \mathtt{let\ x\ =}\ \alpha\ \mathtt{in}\ \beta : \mathtt{D}}$$

Figure 16: Type derivation of `let`

$$\text{t-let}\ \dfrac{\Gamma \vdash \alpha : \mathtt{A} \qquad \Gamma, \mathtt{x} : \mathtt{A} \vdash \beta : \mathtt{B}}{\Gamma \vdash \mathtt{let\ x\ =}\ \alpha\ \mathtt{in}\ \beta : \mathtt{B}}$$

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.

[1] `calc-type` can also be used to force a more specific surface type rule than would be inferred. For example, `(calc-type` α `as List<`X`> in ...)` will lead to a surface type rule that enforces that α 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.

6.4.2 *Recursive Sugars*

Consider boolean guards in Haskell list comprehensions, which are defined by the desugaring rule (in Haskell's syntax):

$$\texttt{[}\alpha\ \texttt{|}\ \beta\texttt{,}\ \gamma\texttt{]} \quad \Rightarrow \quad \texttt{if}\ \beta\ \texttt{then}\ \texttt{[}\alpha\ \texttt{|}\ \gamma\texttt{]}\ \texttt{else}\ \texttt{[]}$$

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 $\Gamma \vdash \texttt{[}\alpha\ \texttt{|}\ \gamma\texttt{]} : \_$, 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 $\texttt{[}\alpha\ \texttt{|}\ \gamma\texttt{]}$ is a surface term: when desugaring, pattern variables such as α and γ 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:

$$\text{t-premise}\ \frac{\text{fresh } x \qquad p \text{ is a surface term}}{\Gamma \vdash p : x}$$

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:

$$\texttt{const}\ \alpha \quad \Rightarrow \quad \lambda\ x : \texttt{Unit.}\ \alpha$$

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 α. 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.)

Picking fresh names for sugar-introduced variables suffices for hygiene because our sugars are declared outside the language.

This feature must also be reflected in the surface type system. First, let $\mathcal{F}$ be the set of introduced variables that are not marked as captured. We then add the type rule:

$$\text{t-fresh}\ \frac{\Gamma \vdash p : t \qquad p \text{ is a surface term} \qquad x_1 \ldots x_n \in \mathcal{F}}{\Gamma, x_1 : t_1 \ldots x_n : t_n \vdash p : t}$$

to remove unnecessary fresh variables from the type environment, and by modifying `t-premise` to only work on judgments so limited:

Our implementation combines `t-fresh` and `t-premise` into one rule for convenience, but the effect is the same.

$$\text{t-premise}\ \frac{p \text{ is a surface term} \qquad \forall x \in \Gamma.\ x \notin \mathcal{F} \qquad \text{fresh } x'}{\Gamma \vdash p : x'}$$

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 Γ. 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:*

$$\frac{x \notin \Gamma \qquad x \notin e \qquad \Gamma, x : t' \vdash e : t}{\Gamma \vdash e : t}$$

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 ability to reference "globals" is but a poor approximation to a macro system that allows macros and code to be interspersed, in which a macro may reference any identifier it is in scope of. However, type resugaring in this setting is a much harder problem which we leave to future work.

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`):

$$\frac{\texttt{plus} : \texttt{N},\texttt{N}\rightarrow\texttt{N},\ \Gamma \vdash \alpha : \texttt{N} \qquad \texttt{plus} : \texttt{N},\texttt{N}\rightarrow\texttt{N},\ \Gamma \vdash \beta : \texttt{N}}{\texttt{plus} : \texttt{N},\texttt{N}\rightarrow\texttt{N},\ \Gamma \vdash \alpha\ \texttt{+}\ \beta : \texttt{N}}$$

However, this is both verbose and unusual, so we opt to leave the `N,N→N` implicit. We do so by adding the type rule:

$$\text{t-global}\ \frac{globals[x] = t}{\Gamma \vdash x : t}$$

which allows `plus` to be left out of Γ.

6.4.5 *Variable Arities*

We support syntactic constructs with variable arity by having a sort called e^* that represents a sequence of expressions:

$$\begin{array}{lcll} e^* & ::= & \epsilon & \text{empty sequence} \\ & | & (\texttt{cons}\ e\ e^*) & \text{nonempty sequence} \\ & | & \alpha & \text{pattern variable} \end{array}$$

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:

$\texttt{(or (cons}\ \alpha\ \epsilon\texttt{))} \ \Rightarrow\ \alpha$

$\texttt{(or (cons}\ \alpha\ \texttt{(cons}\ \beta\ \gamma\texttt{)))} \ \Rightarrow\ \texttt{if}\ \alpha\ \texttt{then true else (or (cons}\ \beta\ \gamma\texttt{))}$

Type resugaring produces one type rule for each desugaring rule:

$$\text{sugar-or-1}\ \frac{\Gamma \vdash \alpha : \texttt{A}}{\Gamma \vdash (\texttt{or}\ (\texttt{cons}\ \alpha\ \epsilon)) : \texttt{A}}$$

$$\text{sugar-or-2}\ \frac{\Gamma \vdash (\texttt{or}\ (\texttt{cons}\ \beta\ \gamma)) : \texttt{Bool} \qquad \Gamma \vdash \alpha : \texttt{Bool}}{\Gamma \vdash (\texttt{or}\ (\texttt{cons}\ \alpha\ (\texttt{cons}\ \beta\ \gamma))) : \texttt{Bool}}$$

The first rule may appear to be too general, but it accurately reflects the sugar as written: (`or` (`cons` 3 ϵ)) is a synonym for 3 and has type Number. However, we can statically restrict the singleton `or` to accept only booleans using `calc-type`:

$\texttt{(or (cons}\ \alpha\ \epsilon\texttt{))} \ \Rightarrow\ \texttt{calc-type}\ \alpha\ \texttt{as Bool in}\ \alpha$

at which point the resugared type rule becomes:

$$\text{sugar-or-1}\ \frac{\Gamma \vdash \alpha : \mathit{Bool}}{\Gamma \vdash (\texttt{or}\ (\texttt{cons}\ \alpha\ \epsilon)) : \mathit{Bool}}$$

as probably desired.

6.5 IMPLEMENTATION

We have implemented a prototype of our tool in PLT Redex [16], a semantics engineering tool. It can be found at `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.
- Desugaring rules, given by a LHS and 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 `calc-type`, as described in section 6.4.1.
- Type definitions of globals, as described in section 6.4.4.

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

SweetT then provides a `resugar` function that follows the process outlined at the end of section 6.2, together with the extensions described in section 6.4. If `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 `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 [8]). We tested the type systems in Part II of TAPL (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 builtin support, as described in section 6.4.5, and subtyping required adding subtyping constraints, as well as a new `t-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 γ is a store environment, which would be a more extensive change.

The `t-sub-premise` rule is like `t-premise`, but for subtyping judgements instead of type judgements.

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 λ and `fix` (the fixpoint operator).

λRET The λ`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 η-expansion. Notice that the core language type system contains subtyping judgements, 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:

```
foreach x list body
=>
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] -> [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.

$$\text{t-foreach}\ \frac{\Gamma \vdash \mathit{list} : \texttt{List}\ D \qquad \Gamma, x : D, \mathit{break} : (\texttt{Unit} \to B) \vdash \mathit{body} : F}{\Gamma \vdash \texttt{foreach}\ x\ \mathit{list}\ \mathit{body} : \texttt{List}\ F}$$

Figure 17: `foreach` type rule

HASKELL LIST COMPREHENSIONS List comprehensions [46, 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 <- l, Q]     = let ok p = [e | Q]
                          ok _ = []
                      in concatMap ok l
[e | let decls, Q] = let decls in [e | Q]
```

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 <- 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:

```
[e | b, Q] = if b then [e | Q] else []
[e | x <- l, Q] = calc-type l as [t] in
    concatMap (\(x :: t) -> [e | Q]) l
[e | let x = e2, Q] = (let x = e2 in [e | Q])
```

"hlc" stands for "Haskell list comprehension".

SweetT resugars these rules, producing the following type rules (transcribed into Haskell syntax):

$$\text{t-hlc-guard}\ \frac{\Gamma \vdash \texttt{[}e\ \texttt{|}\ Q\texttt{]} : \mathrm{C} \qquad \Gamma \vdash b : \mathit{Bool}}{\Gamma \vdash \texttt{[}e\ \texttt{|}\ b\texttt{,}\ Q\texttt{]} : \mathrm{C}}$$

$$\text{t-hlc-gen}\ \frac{\Gamma, x : t \vdash \texttt{[}e\ \texttt{|}\ Q\texttt{]} : [\mathrm{o}] \qquad \Gamma \vdash l : [\mathrm{t}]}{\Gamma \vdash \texttt{[}e\ \texttt{|}\ x\ \texttt{<-}\ l\texttt{,}\ Q\texttt{]} : [\mathrm{o}]}$$

$$\text{t-hlc-let}\ \frac{\Gamma, x : A \vdash \texttt{[}e\ \texttt{|}\ Q\texttt{]} : \mathrm{B} \qquad \Gamma \vdash e_2 : \mathrm{A}}{\Gamma \vdash \texttt{[}e\ \texttt{|}\ \texttt{let}\ x\ \texttt{=}\ e_2\texttt{,}\ Q\texttt{]} : \mathrm{B}}$$

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:

$$\text{t-pack}\ \frac{\Gamma \vdash e : [\mathrm{X} \mapsto \mathrm{U}]\mathrm{t}}{\Gamma \vdash \texttt{pack}\ \texttt{(}U\ e\texttt{)}\ \texttt{as}\ \texttt{(}\exists\ X\ t\texttt{)} : \texttt{(}\exists\ X\ t\texttt{)}}$$

$$\text{t-unpack}\ \frac{\Gamma \vdash e_1 : (\exists\ X\ t_1) \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{unpack}\ e_1\ \texttt{as}\ (\exists\ X\ x)\ \texttt{in}\ e_2 : t_2}$$

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:

```
new-type (wrap unwrap) of T as X in body
⇒
unpack (pack (T (pair id id)
          as (∃ X (Pair (T → X) (X → T)))))
as (∃ X w)
in let wrap = fst w in
   let unwrap = snd w in
   body
```

where `id` is a global (section 6.4.4) identity function.

This sugar is successfully resugared to give the type rule:

$$\text{t-new-type}\ \frac{\Gamma, u : X \to T, w : T \to X \vdash body : A}{\Gamma \vdash \texttt{new-type}\ (w\ u)\ \texttt{of}\ T\ \texttt{as}\ X\ \texttt{in}\ body : A}$$

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 `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* [39], 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* [40] 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 [22] and Mainland's MetaHaskell [41] 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 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) [43]. 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 [25, 55]. 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 [62] and Typed Clojure [4]. 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 [57], Ocaml [13], and Scala [7]. 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.)

Fish and Shivers' Ziggurat [17] 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 5.9.

WORK WITH A DIFFERENT GOAL Our work could be contrasted with Chang et al.'s *Turnstile* [9]. 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 [29] or structural operational semantics [47]. 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 λret

CORE TYPE RULES:

$$\text{t-lambda}\ \dfrac{\Gamma, \texttt{x} : \texttt{T} \vdash \texttt{e} : \texttt{U}}{\Gamma \vdash \lambda\ \texttt{x:T. e} : (\texttt{T} \rightarrow \texttt{U})} \qquad \text{t-apply}\ \dfrac{\Gamma \vdash \texttt{f} : \texttt{T} \rightarrow \texttt{U} \qquad \Gamma \vdash \texttt{e} : \texttt{T}}{\Gamma \vdash (\texttt{f e}) : \texttt{U}}$$

$$\text{t-fix}\ \dfrac{\Gamma \vdash \texttt{e} : \texttt{T} \rightarrow \texttt{T}}{\Gamma \vdash (\texttt{fix e}) : \texttt{T}}$$

$$\text{t-raise}\ \dfrac{\Gamma \vdash \texttt{e} : \texttt{Str}}{\Gamma \vdash (\texttt{raise e}) : \texttt{T}} \qquad \text{t-try}\ \dfrac{\Gamma \vdash \texttt{e} : \texttt{T} \qquad \Gamma \vdash \texttt{e}_{\texttt{catch}} : \texttt{Str} \rightarrow \texttt{T}}{\Gamma \vdash \texttt{try e with } \texttt{e}_{\texttt{catch}} : \texttt{T}}$$

DESUGARING RULES:

$$\texttt{letrec x : C = a in b} \Rightarrow (\lambda\ \texttt{x:C. b) (fix (}\lambda\ \texttt{x:C. a))}$$

$$\begin{aligned}&\lambda\texttt{ret x:T. b}\\ &\Rightarrow \lambda\ \texttt{x:T. try (let return = (}\lambda\ \texttt{v:Str. raise v) in b) with (}\lambda\ \texttt{v:Str. v)}\end{aligned}$$

λ`ret` is a function with `return` automatically bound (i.e., marked as capturing) to escape from the function.

CORE DERIVATIONS:

$$\text{t-letrec}^{\rightarrow}\ \dfrac{\text{t-apply}\ \dfrac{\text{t-lambda}\ \dfrac{\text{t-premise}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{C} \vdash \texttt{b} : \texttt{D}}}{\Gamma \vdash (\lambda\ \texttt{x:C. b}) : \texttt{C} \rightarrow \texttt{D}} \qquad \text{t-fix}\ \dfrac{\text{t-lambda}\ \dfrac{\text{t-premise}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{C} \vdash \texttt{a} : \texttt{C}}}{\Gamma \vdash (\lambda\ \texttt{x:C. a}) : \texttt{C} \rightarrow \texttt{C}}}{\Gamma \vdash (\texttt{fix (}\lambda\ \texttt{x:C. a)}) : \texttt{C}}}{\Gamma \vdash ((\lambda\ \texttt{x:C. b) (fix (}\lambda\ \texttt{x:C. a))}) : \texttt{D}}}{\Gamma \vdash \texttt{letrec x : C = a in b} : \texttt{D}}$$

$$\text{t-}\lambda\text{ret}^{\rightarrow}\ \dfrac{\text{t-}\lambda\ \dfrac{\text{t-try}\ \dfrac{\text{t-let}\ \dfrac{\text{t-}\lambda\ \dfrac{\text{t-raise}\ \dfrac{\text{t-id}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{T}, \texttt{v} : \texttt{Str} \vdash \texttt{v} : \texttt{Str}}}{\Gamma, \texttt{x} : \texttt{T}, \texttt{v} : \texttt{Str} \vdash \texttt{raise v} : \texttt{A}}}{\Gamma, \texttt{x} : \texttt{T} \vdash (\lambda\ \texttt{v:Str. raise v}) : \texttt{Str} \rightarrow \texttt{A}} \qquad \text{t-prem.}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{T}, \texttt{return} : \texttt{Str} \rightarrow \texttt{A} \vdash \texttt{b} : \texttt{Str}}}{\Gamma, \texttt{x} : \texttt{T} \vdash (\texttt{let return = (}\lambda\ \texttt{v:Str. raise v) in b}) : \texttt{Str}} \qquad \text{t-}\lambda\ \dfrac{\text{t-id}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{T}, \texttt{v} : \texttt{Str} \vdash \texttt{v} : \texttt{Str}}}{\Gamma, \texttt{x} : \texttt{T} \vdash (\lambda\ \texttt{v:Str. v}) : \texttt{Str} \rightarrow \texttt{Str}}}{\Gamma, \texttt{x} : \texttt{T} \vdash (\texttt{try (let return = (}\lambda\ \texttt{v:Str. raise v) in b) with (}\lambda\ \texttt{v:Str. v)}) : \texttt{Str}}}{\Gamma \vdash \lambda\ \texttt{x:T. (try (let return = (}\lambda\ \texttt{v:Str. raise v) in b) with (}\lambda\ \texttt{v:Str. v))} : \texttt{T} \rightarrow \texttt{Str}}}{\Gamma \vdash (\lambda\texttt{ret x:T. b}) : \texttt{T} \rightarrow \texttt{Str}}$$

RESUGARED TYPE RULES:

$$\text{t-letrec}\ \dfrac{\Gamma, \texttt{x} : \texttt{C} \vdash \texttt{a} : \texttt{C} \qquad \Gamma, \texttt{x} : \texttt{C} \vdash \texttt{b} : \texttt{D}}{\Gamma \vdash \texttt{letrec x : C = a in b} : \texttt{D}}$$

$$\text{t-}\lambda\text{ret}\ \dfrac{\Gamma,\ \texttt{x} : \texttt{T},\ \texttt{return} : (\texttt{Str} \rightarrow \texttt{A}) \vdash \texttt{b} : \texttt{Str}}{\Gamma \vdash (\lambda\texttt{ret x:T. b}) : \texttt{T} \rightarrow \texttt{Str}}$$

Figure 18: Derivation examples

Sugar: Upcast

Core Type Rules:

$$\text{t-id}\ \frac{\texttt{x:T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \qquad \text{t-lambda}\ \frac{\Gamma, \texttt{x} : \texttt{T} \vdash \texttt{e} : \texttt{U}}{\Gamma \vdash \lambda\ \texttt{x:T.}\ \texttt{e} : (\texttt{T} \to \texttt{U})}$$

$$\text{t-apply}\ \frac{\Gamma \vdash \texttt{f} : \texttt{T} \to \texttt{U} \qquad \Gamma \vdash \texttt{e} : \texttt{T}' \qquad \texttt{T}' <: \texttt{T}}{\Gamma \vdash (\texttt{f}\ \texttt{e}) : \texttt{U}}$$

Desugaring Rule:

$$\texttt{upcast a as C} \Rightarrow (\lambda\ \texttt{x:C.}\ \texttt{x})\ \texttt{a}$$

Core Derivation:

$$\text{t-upcast}^{\to}\ \dfrac{\text{t-apply}\ \dfrac{\text{t-lambda}\ \dfrac{\text{t-id}\ \dfrac{}{\Gamma, \texttt{x} : \texttt{C} \vdash \texttt{x} : \texttt{C}}}{\Gamma \vdash (\lambda\ \texttt{x:C.}\ \texttt{x}) : \texttt{C} \to \texttt{C}} \qquad \text{t-premise}\ \dfrac{}{\Gamma \vdash \texttt{a} : \texttt{A}} \qquad \text{t-sub-premise}\ \dfrac{}{\texttt{A} <: \texttt{C}}}{\Gamma \vdash ((\lambda\ \texttt{x:C.}\ \texttt{x})\ \texttt{a}) : \texttt{C}}}{\Gamma \vdash \texttt{upcast a as C} : \texttt{C}}$$

Resugared Type Rule:

$$\text{t-upcast}\ \frac{\Gamma \vdash \texttt{a} : \texttt{A} \qquad \texttt{A} <: \texttt{C}}{\Gamma \vdash \texttt{upcast a as C} : \texttt{C}}$$

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
  [(⊢ Γ e_1 t_1)
   (⊢ Γ e_2 t_2)
   (⊢ Γ e_3 t_3)
   (con (t_1 = Bool))
   (con (t_2 = t_3))
   ------ t-if
   (⊢ Γ (if e_1 e_2 e_3) t_3)]

  [------ t-true
   (⊢ Γ true Bool)]

  [------ t-false
   (⊢ Γ false Bool)])

(define rule_not
  (ds-rule "not" #:capture()
           (not ~a)
           (if ~a false true)))

(view-sugar-type-rules demo ⊢ (list rule_not))
```

INFERS THE FOLLOWING RESUGARED TYPE RULE:

$$\text{not}\,\frac{(\Gamma \vdash\ a : \text{Bool})}{(\Gamma \vdash (\text{not}\ \ a) : \text{Bool})}$$

Figure 20: Sample SweetT usage

Booleans

```
(ds-rule "unless" #:capture()
  (unless ~a ~b)
  (if ~a unit ~b))
```

$$\text{unless}\ \frac{(\Gamma \vdash\ a : \text{Bool}) \qquad (\Gamma \vdash\ b : \text{Unit})}{(\Gamma \vdash (\text{unless}\ \ a\ \ b) : \text{Unit})}$$

Nats

```
(ds-rule "ifzero" #:capture()
  (ifzero ~a ~b ~c)
  (if (iszero ~a) ~b ~c))
```

$$\text{ifzero}\ \frac{(\Gamma \vdash\ c : C) \qquad (\Gamma \vdash\ a : \text{Nat}) \qquad (\Gamma \vdash\ b : C)}{(\Gamma \vdash (\text{ifzero}\ \ a\ \ b\ \ c) : C)}$$

STLC

```
(ds-rule "let" #:capture()
  (let x = ~a in ~b)
    (calctype ~a as t in
      ((λ (x : t) ~b) ~a)))
```

$$\text{let}\ \frac{(\Gamma \vdash\ a : A) \qquad ((\text{bind}\ x\ A\ \Gamma) \vdash\ b : D)}{(\Gamma \vdash (\text{let}\ x =\ a\ \text{in}\ \ b) : D)}$$

Unit

```
(ds-rule "thunk" #:capture()
  (thunk ~a)
  (λ (x : Unit) ~a))
```

$$\text{thunk}\ \frac{(\Gamma \vdash\ a : A)}{(\Gamma \vdash (\text{thunk}\ \ a) : (\text{Unit -> } A))}$$

Ascription

```
(ds-rule "sametype" #:capture()
  (sametype ~a ~b)
  (calctype ~b as x_t in
    (~a as x_t)))
```

$$\text{sametype}\ \frac{(\Gamma \vdash\ a : B) \qquad (\Gamma \vdash\ b : B)}{(\Gamma \vdash (\text{sametype}\ \ a\ \ b) : B)}$$

Let

```
(ds-rule "or" #:capture()
  (or ~a ~b)
  (let x = ~a in (if x x ~b)))
```

$$\text{or}\ \frac{(\Gamma \vdash\ a : \text{Bool}) \qquad (\Gamma \vdash\ b : \text{Bool})}{(\Gamma \vdash (\text{or}\ \ a\ \ b) : \text{Bool})}$$

Figure 21: Sample sugars, pg.1

Pair

```
(ds-rule "let-pair" #:capture()
  (let-pair x y = ~a in ~b)
  (calctype ~a as (Pair t_1 t_2) in
    (let p = ~a in
      (let x = (fst p) in
        (let y = (snd p) in
          ~b)))))
```

$$\text{let-pair}\ \frac{(\Gamma \vdash\ \text{a} : (\text{Pair E F})) \qquad ((\text{bind y F (bind x E } \Gamma)) \vdash\ \text{b} : \text{G})}{(\Gamma \vdash (\text{let-pair x y = } \ \text{a in } \ \text{b}) : \text{G})}$$

Tuple

```
(ds-rule "tuple2" #:capture()
  (tuple2 ~a ~b)
  (tuple (cons ~a (cons ~b ϵ))))
```

$$\text{tuple2}\ \frac{(\Gamma \vdash\ \text{a} : \text{A}) \qquad (\Gamma \vdash\ \text{b} : \text{B})}{(\Gamma \vdash (\text{tuple2 } \ \text{a } \ \text{b}) : (\text{Tuple (cons A (cons B } \epsilon))))}$$

Record

```
(ds-rule "rec-point" #:capture()
  (rec-point ~a ~b)
  (record (field x ~a (field y ~b ϵ))))
```

$$\text{rec-point}\ \frac{(\Gamma \vdash\ \text{a} : \text{A}) \qquad (\Gamma \vdash\ \text{b} : \text{B})}{(\Gamma \vdash (\text{rec-point } \ \text{a } \ \text{b}) : (\text{Record (field x A (field y B } \epsilon))))}$$

Sum

```
(ds-rule "sum-map" #:capture()
  (sum-map ~a ok ~b)
  (case ~a of
    (inl err => (inl err))
    (inr ok => (inr ~b))))
```

$$\text{sum-map}\ \frac{(\Gamma \vdash\ \text{a} : (\text{Sum F B})) \qquad ((\text{bind ok B } \Gamma) \vdash\ \text{b} : \text{E})}{(\Gamma \vdash (\text{sum-map } \ \text{a ok } \ \text{b}) : (\text{Sum F E}))}$$

Figure 22: Sample sugars, pg.2

Variant

```
;Show that variants make sums irrelevant
(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))))))
```

$$\text{inl}\ \frac{(\Gamma \vdash\ \text{l} : \text{A})}{(\Gamma \vdash (\text{inl* }\ \text{l}) : (\text{Variant (field l A (field r t\_r } \epsilon))))}$$

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))))))
```

$$\text{inr}\ \frac{(\Gamma \vdash\ \text{r} : \text{A})}{(\Gamma \vdash (\text{inr* }\ \text{r}) : (\text{Variant (field l t\_l (field r A } \epsilon))))}$$

Variant

```
(ds-rule "case" #:capture()
  (case* ~e of (x => ~l) (y => ~r))
  (case ~e of (cons (l = x => ~l)
              (cons (r = y => ~r) ϵ))))
```

$$\text{case}\ \frac{(\Gamma \vdash\ \text{e} : (\text{Variant (field l C (field r H } \rho 1))))\quad ((\text{bind x C } \Gamma) \vdash\ \text{l} : \text{K})\quad ((\text{bind y H } \Gamma) \vdash\ \text{r} : \text{K})}{(\Gamma \vdash (\text{case* }\ \text{e of (x => }\ \text{l) (y => }\ \text{r)}) : \text{K})}$$

Fixpoint

```
(ds-rule "letrec" #:capture()
  (letrec x : ~t = ~a in ~b)
    ((λ (x : ~t) ~b)
      (fix (λ (x : ~t) ~a)))))
```

$$\text{letrec}\ \frac{((\text{bind x C } \Gamma) \vdash\ \text{a} : \text{C})\qquad ((\text{bind x C } \Gamma) \vdash\ \text{b} : \text{D})}{(\Gamma \vdash (\text{letrec x} : \text{C} =\ \text{a in }\ \text{b}) : \text{D})}$$

Figure 23: Sample sugars, pg.3

BIBLIOGRAPHY

[1] Peter Aczel. A general church-rosser theorem. Technical report, University of Manchester, 1978.

[2] Michael D. Adams. Towards the essence of hygiene. In *Principles of Programming Languages*, pages 457–469, New York, NY, USA, 2015. ACM.

[3] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1991.

[4] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure. In *European Symposium on Programming Languages and Systems*, Berlin, Heidelberg, 2016. Springer-Verlag.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2), 2008.

[6] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Krishnamurthi Shriram, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12:159–182, 2002.

[7] Eugene Burmako. Scala macros: let our powers combine! In *Scala Workshop*, New York, NY, USA, 2013. ACM.

[8] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[9] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Principles of Programming Languages*, New York, NY, USA, 2017. ACM.

[10] John Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2006.

[11] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming Languages and Systems*, London, UK, 2001. Springer-Verlag.

[12] Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *International Conference on Functional Programming*, pages 235–246, New York, NY, USA, 2010. ACM.

[13] Daniel de Rauglaudre. Camlp5 - reference manua, 2007. http://pauillac.inria.fr/ ddr/camlp5/doc/pdf/camlp5-5.06.pdf.

[14] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. In *Lisp and Symbolic Computation*, 1988.

[15] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-avoiding and hygienic program transformations. In *European Conference on Object-Oriented Programming*, pages 489–514, Berlin, Heidelberg, 2014. Springer-Verlag.

[16] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

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

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Languages Design and Implementation*, New York, NY, USA, 1993. ACM.

[19] Matthew Flatt. Binding as sets of scopes. In *Principles of Programming Languages*. ACM, 2016.

[20] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `racket-lang.org/tr1/`.

[21] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Principles of Programming Languages*, 2005.

[22] Steve Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *International Conference on Functional Programming*, New York, NY, USA, 2001. ACM.

[23] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification: Java SE 8 Edition, 2015. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html`.

[24] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-oriented Programming*, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *International Conference on Functional Programming*, New York, NY, USA, 2003. ACM.

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

[27] David Herman and Mitchell Wand. A theory of hygienic macros. In *European Symposium on Programming Languages and Systems*, pages 48–62, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Languages Design and Implementation*, New York, NY, USA, 1992. ACM.

[29] Gilles Kahn. Natural semantics. In *International Symposium on Theoretical Aspects of Computer Software*, 1987.

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

[31] Donald E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*. Springer-Verlag, 1968.

[32] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages*, New York, NY, USA, 1987. ACM.

[33] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming*, New York, NY, USA, 1986. ACM.

[34] Gabrie Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, pages 311–331, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] Shriram Krishnamurthi. Automata via macros. *Journal of Functional Programming*, 16(3), 2006.

[36] Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *Generative and Component-Based Software Engineering*, 1999.

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

[38] Florian Lorenzen and Sebastian Erdweg. Modular and automated type-soundness for language extensions. In *International Conference on Functional Programming*, New York, NY, USA, 2013. ACM.

[39] Florian Lorenzen and Sebastian Erdweg. Modular and automated type-soundness for language extensions. In *International Conference on Functional Programming*, New York, NY, USA, 2013. ACM.

[40] Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *Principles of Programming Languages*, New York, NY, USA, 2016. ACM.

[41] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with metahaskell. In *International Conference on Functional Programming*, New York, NY, USA, 2012. ACM.

[42] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *European Symposium on Programming Languages and Systems*, pages 205–231, Berlin, Heidelberg, 2015. Springer-Verlag.

[43] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *European Conference on Object-Oriented Programming*, New York, NY, USA, 2014. Springer-Verlag.

[44] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *International Conference on Functional Programming*, New York, NY, USA, 2012. ACM.

[45] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, 2005.

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

[47] Gordon D. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[48] Joe Gibbs Politz, Matt Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Dynamic Languages Symposium*, 2012.

[49] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty: A tested semantics for the python programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 217–232, New York, NY, USA, 2013. ACM.

[50] Justin Pombrio and Shriram Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Programming Languages Design and Implementation*, pages 361–371, New York, NY, USA, 2014. ACM.

[51] Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *International Conference on Functional Programming*, pages 75–87, New York, NY, USA, 2015. ACM.

[52] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. Inferring scope through syntactic sugar. In *International Conference on Functional Programming*, New York, NY, USA, 2017. ACM.

[53] François Pottier. An overview of Cαml. In *Electronic Notes in Theoretical Computer Science*, 2005.

[54] R Core Team. R Language Definition, 2018.
`https://cran.r-project.org/doc/manuals/R-lang.html`.

[55] 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.

[56] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 2010.

[57] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Haskell Workshop*, New York, NY, USA, 2002. ACM.

[58] Joel Spolsky. The law of leaky abstractions. Blog post: Joel on Software, 2002. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

[59] Paul Stansifer and Mitchell Wand. Romeo: a system for more flexible binding-safe programming. In *International Conference on Functional Programming*, pages 53–65, New York, NY, USA, 2014. ACM.

[60] Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*. Springer-Verlag, 2008.

[61] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, version 8.4 edition, 2012.

[62] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Programming Languages Design and Implementation*, New York, NY, USA, 2011. ACM.

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

[64] Stephanie Weirich, Brent Yorgey, and Tim Sheard. Binders unbound. In *International Conference on Functional Programming*, pages 333–345, New York, NY, USA, 2011. ACM.