Resugaring: Lifting Evaluation Sequences through Syntactic Sugar

Justin Pombrio
Brown University
justinpombrio@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

Syntactic sugar is pervasive in language technology. It is used to shrink the size of a core language; to define domain-specific languages; and even to let programmers extend their language. Unfortunately, syntactic sugar is eliminated by transformation, so the resulting programs become unfamiliar to authors. Thus, it comes at a price: it obscures the relationship between the user's source program and the program being evaluated.

We address this problem by showing how to compute reduction steps in terms of the surface syntax. Each step in the surface language emulates one or more steps in the core language. The computed steps hide the transformation, thus maintaining the abstraction provided by the surface language. We make these statements about emulation and abstraction precise, prove that they hold in our formalism, and verify part of the system in Coq. We have implemented this work and applied it to three very different languages.

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

Keywords Programming Languages, Syntactic Sugar, Macros, Evaluation, Debugging, Resugaring

1. Introduction

Syntactic sugar is an essential component of programming languages and systems. It is central to the venerable linguistic tradition of defining programming languages in two parts: a (small) core language and a rich family of usable syntax atop that core. It is also used to provide the special syntaxes that characterize some domain-specific languages. Finally, some languages (notably members of the Lisp family) provide macros so even individual programmers can customize the language on a per-program or per-module basis.

In essence, desugaring is a compilation process. It begins with a rich language with its own abstract syntax, and compiles it by applying transformations to a smaller language with a correspondingly smaller abstract syntax. 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.

Unfortunately, syntactic sugar is a leaky abstraction. Many debugging and comprehension tools—such as an algebraic stepper [4] or reduction semantics explorer [9, 23]—present their output *using terms in the language*; this is also true of proof-theoretic tools such as theorem provers. Thus, when applied to core language terms resulting from desugaring, their output is also in terms of the core, thereby losing any correspondence with the surface language that the user employs.

There are several partial solutions to this, all unsatisfactory. One is to manually redefine the semantics in terms of the full surface language, which destroys the benefits provided by a small core. Another is to employ source tracking, but this is not actually a solution: the user will still see only expanded terms. Furthermore, any attempt to create a one-time solution for a given language does not apply to those languages where users can create additional syntactic sugar in the program itself (as in the Lisp tradition).

In this paper, we tackle the challenge of combining syntactic sugar with semantics. Given a set of transformation rules written in an expressive but restricted language, 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. 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 definitions 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.

2. 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 7 we show how to obtain such a stepper from

PLDI '14, June 9–11, 2014, Edinburgh, UK.

ACM 978-1-4503-2784-8/14/06.
http://dx.doi.org/10.1145/2594291.2594319

a generic, black-box evaluator with a strategy that can be implemented by pre-processing the program before evaluation.

The main contribution of this paper is a technique for *resugaring*: lifting the core evaluation sequence into one for the surface. 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. We have implemented a tool, CONFECTION, that performs this lifting.

Central to our approach are these three properties:

Emulation Each term in the generated surface evaluation sequence desugars into the core term which it is meant to represent.

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.

The rest of the paper presents this both informally and formally.

3. 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, and section 4 shows our work applied to such languages.

3.1 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):

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

This is the actual transformation syntax processed by CONFECTION, inspired by that of Stratego [3]. Nodes' names are written in title-case and their subnodes follow in parentheses, lists are surrounded by square brackets, and variables are written in lowercase. We call the whole production a *transformation rule*, or *rule* for short; the part to the left of the arrow is its LHS (left-hand-side), and the part after the arrow its RHS. When the LHS of a rule matches a term, this induces bindings for the variables in the LHS, which are then substituted in the RHS. The full definition of transformations is given in section 5. 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 6.2.) We will now show how the last two steps may come about.

3.2 Maintaining Abstraction

When unexpanding, we should only unexpand code that originated from a transformation. 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 the transformation.[1] Unexpansion attempts to process terms marked as originating from a transformation rule; this unexpansion will fail if the tagged term no longer has the form of the rule's RHS. When that happens, we conclude that there is no surface representation of the core term and skip the step.

To illustrate this, we revisit the same core evaluation sequence as before. Compound terms that originated from the source are underlined, and "`Or:` " is a tag on the desugared expression that indicate it originated from the `Or` sugar:

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

The tags on the first three steps suggest that `Or`'s transformation rule be applied in reverse. The first two steps can be unexpanded because they match the RHS of `Or`, whereas the third does not match the RHS and 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.

3.3 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 6.3, we have not found a formalization of Coverage, so we can only strive to attain it in our systems and evaluate it in practice. Our examples (section 4 and section 8) show that we do indeed obtain detailed and useful surface evaluation sequences.

[1] Whereas the tags used in hygienic macro expansion [16] specify time steps, the tags in resugaring specify which sugar the code originated from.

3.4 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 [17].

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 the transformation—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. 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"])))
```

P	:=	`x`	(pattern variable)
	|	**a**	(constant)
	|	$l(P_1, ..., P_n)$	(node labeled l)
	|	$(P_1 \ldots P_n)$	(list of length n)
	|	$(P_1 \ldots P_n P_e{}^*)$	(list of length $\geq n$)
	|	(`Tag` O P)	(origin tag)
T	:=	P	(pattern w/o variables or ellipses)
O	:=	(`Head` i T)	(marks topmost rule production)
	|	(`Body` *bool*)	(marks each rule production)

Figure 1. Patterns

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} [13], 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.

5. The Transformation System

We will present our system in three parts. First (section 5.1), we will describe how our transformation system works, up to the level of performing a single transformation. Next (section 5.2), 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 5.3), we will show how to use the transformation system to lift core evaluation sequences to surface evaluation sequences.

5.1 Performing a Single Transformation

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

5.1.1 The Pattern Language

Since rules are applied both forward and in reverse, we represent their LHSs and RHSs uniformly as *patterns*. Patterns P are defined

$$
\begin{array}{rcll}
b & := & P & \text{(pattern)} \\
 & | & [|b_1 \ldots b_n|] & \text{(list binding)} \\
 & | & [|b_1 \ldots b_n {b_e}^*|] & \text{(ellipsis binding)} \\
 & & & \\
\sigma & := & \{\mathtt{x} \to b, \ldots\} &
\end{array}
$$

Figure 2. Bindings

inductively in figure 1. Variables are denoted by a lowercase identifier, labeled nodes are denoted by an uppercase identifier followed by a parenthesized list of subpatterns, and lists are denoted by a parenthesized 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 $\bullet^*$ to distinguish them from metasyntactic ellipses) in a list pattern denote zero or more repetitions of the pattern they follow. A *term* T is simply a pattern without variables or ellipses. Tags and origins (O) are described in section 5.2.1. We do *not* address hygiene in our system. We believe it is largely orthogonal to the problem at hand, and that our transformation system could be made hygienic without significant alterations.

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, our patterns are *regular tree expressions* [1]. Regular tree expressions trx are the natural extension of regular expressions to handle trees: they add a primitive ($\mathbf{l}\ trx_1 \ldots trx_n$) for matching a tree node labeled $\mathbf{l}$ with branches matching the regular tree expressions $trx_1 \ldots trx_n$. 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 6.3.

5.1.2 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 variables. This environment may be *substituted* into a pattern to produce another term. Formally, an environment is a mapping from pattern variables `x` to bindings b, where each *binding* is either a term T, a *list binding* $[|b_1 \ldots b_n|]$, or an *ellipsis binding* $[|b_1 \ldots b_n\ {b_e}^*|]$. A pattern variable within ellipses is bound to a list binding $[|b_1 \ldots b_n|]$ instead of a list term $(b_1 \ldots b_n)$; they behave slightly differently under substitution. Ellipsis bindings are similar, but needed only during unification when a variable within an ellipsis is itself bound to an ellipsis pattern.

We will write T/P to denote matching a term T against a pattern P, and write σP to denote substituting the bindings of an environment σ into a pattern P. We will write $T \geq P$ to mean that T/P is defined, and $\sigma_1 \cdot \sigma_2$ for the *right-biased* union of σ_1 and σ_2. The matching and substitution algorithms are given in figure 3, while bindings are defined in figure 2.

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

$$\sigma = \{\mathtt{x} \to \mathtt{true},\ \mathtt{y} \to \mathtt{Not(true)},\ \mathtt{ys} \to [|\mathtt{false}, \mathtt{true}|]\}$$

and substituting σ 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 variables (as seen in the next section).

5.1.3 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 $(T/P)P = T$ whenever T/P exists fails to hold in certain situations, such as when a pattern's ellipsis contains no variables (e.g., $(\mathbf{3}^*)$). For this reason and others, we require the following well-formedness criteria for the LHS and RHS of each rule:

1. *Each variable in the* RHS *also appears in the* LHS. Otherwise the variable would be unbound during expansion.
2. *Each variable appears at most once in the* LHS *and at most once in the* RHS. Allowing duplicate variables complicates matching, unification, and proofs of correctness. It also copies code and, in the worst case, can exponentially blow up programs. We therefore disallow duplication, with the sole exception of variables bound to atomic terms.
3. *An ellipsis of depth n must contain at least one 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* $l(T_1, \ldots, T_n)$. We will rely on this fact when showing that unexpansion is an inverse of expansion in section 6.2.

The first two restrictions are further justified by our formalization of expansion and unexpansion in Coq (section 6.4), where they occurred naturally as pre-conditions for proofs.

5.1.4 Applying Transformations

A *rulelist* rs is an ordered list of transformation rules $P_i \to P_i'$, where each rule is well-formed according to the criteria just described. A term T can then be *expanded* with respect to rs by matching T 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,

$$\begin{array}{rcl}
\mathtt{exp}_{\mathrm{rs}}\ T & = & (j, (T/P_j)P_j') \\
 & & \text{for } j = \min\{i | T \geq P_i\}_i
\end{array}$$

Unexpansion proceeds in reverse, matching against P_i' and then substituting into P_i. Recall, however, that our well-formedness criteria insisted that the 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 variables may be dropped in the RHS, the unexpansion of a term T' takes an additional

$$
\begin{array}{lcl}
\mathbf{a}/\mathbf{a} & = & \{\} \\
T/\mathsf{x} & = & \{\mathsf{x} \to T\} \\
(T_1 ... T_n)/(P_1 ... P_n) & = & \bigcup_{i=1..n}(T_i/P_i) \\
(T_1 ... T_n ... T_{n+k})/(P_1 ... P_n {P_e}^*) & = & \bigcup_{i=1..n}(T_i/P_i) \cup merge([T_{n+i}/P_e]_{i=1..k}) \\
l(T_1, \ ..., \ T_n)/l(P_1, \ ..., \ P_n) & = & \bigcup_{i=1..n}(T_i/P_i) \\
 & & \\
\sigma\,\mathbf{a} & = & \mathbf{a} \\
\sigma\,(P_1 ... P_n) & = & (\sigma P_1 ... \sigma P_n) \\
\sigma\,(P_1 ... P_n {P_e}^*) & = & (\sigma P_1 ... \sigma P_n \ +\!\!+ \ split(\sigma, P_e)) \ \text{(where } +\!\!+ \text{ is concatenation)} \\
\{..., \mathsf{x} \to b, ...\}\,\mathsf{x} & = & toTerm(b) \\
\sigma\, l(P_1, \ ..., \ P_n) & = & l(\sigma P_1, \ ..., \ \sigma P_n) \\
 & & \\
merge([\{\mathsf{x}_1 \to b_1, ...\}, \ ..., \{\mathsf{x}_n \to b_n, ...\}]) & = & \{\mathsf{x}_1 \to [|b_1, \ ..., b_n|], \ ...\} \\
 & & \\
split(\{\mathsf{x}_1 \to [|b_{11} ... b_{1k}|], \ ..., \ \mathsf{x}_n \to [|b_{n1} ... b_{nk}|]\}, P) & = & (\{\mathsf{x}_1 \to b_{11}, \ ..., \ \mathsf{x}_n \to b_{n1}\}P \ ... \ \{\mathsf{x}_1 \to b_{1k}, \ ..., \ \mathsf{x}_n \to b_{nk}\}P) \\
 & & \\
toTerm(P) & = & P \\
toTerm([|b_1 \ ... \ b_n|]) & = & (toTerm(b_1) \ ... \ toTerm(b_n))
\end{array}
$$

Figure 3. Matching and substitution

argument—the original input term T—with which to bind variables in P_i that do not appear in P_i'. Formally,

$$\mathsf{unexp}_{\mathrm{rs}}\,(j, T')\,T \quad = \quad ((T/P_j) \cdot (T'/P_j'))P_j$$

Notice that T contains a good deal of redundant information. Since P_j and P_j' are statically known, it suffices to store only the environment $\sigma = T/P_j$ restricted to the variables not free in P_j'. We will say that σ *stands in* for T, and overload $\mathsf{unexp}_{\mathrm{rs}}$ by writing:

$$\mathsf{unexp}_{\mathrm{rs}}\,(j, T')\,\sigma \quad = \quad (\sigma \cdot (T'/P_j'))P_j$$

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

5.1.5 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.[2] 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 6.1.2.

5.2 Performing Transformations Recursively

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

5.2.1 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 3.4, 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.

[2] Bohannon, et al. [2] use the same disjointness precondition in their union operator for lenses.

`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 variables than the LHS, `Head` tags store the bindings σ for those 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.

5.2.2 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,[3] applying $\texttt{exp}_{\text{rs}}$ at each node:

$$
\begin{array}{lcl}
\texttt{desugar}_{\text{rs}}\ \mathbf{a} & = & \mathbf{a} \\
\texttt{desugar}_{\text{rs}}\ l(T_1,\ ...,\ T_n) & = & \texttt{desugar}_{\text{rs}}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ T') \\
 & & \text{where } \sigma \text{ stands in for } l(T_1,\ ...,\ T_n)/P_i \\
 & & \text{when } \texttt{exp}_{\text{rs}}\ l(T_1,\ ...,\ T_n) = (i, T') \\
\texttt{desugar}_{\text{rs}}\ l(T_1,\ ...,\ T_n) & = & l(\texttt{desugar}_{\text{rs}}\ T_1, ..., \texttt{desugar}_{\text{rs}}\ T_n) \\
 & & \text{otherwise} \\
\texttt{desugar}_{\text{rs}}\ (T_1 \ldots T_n) & = & (\texttt{desugar}_{\text{rs}}\ T_1 \ldots \texttt{desugar}_{\text{rs}}\ T_n) \\
\texttt{desugar}_{\text{rs}}\ (\texttt{Tag}\ O\ T) & = & (\texttt{Tag}\ O\ \texttt{desugar}_{\text{rs}}\ T)
\end{array}
$$

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

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.

$$
\begin{array}{lcll}
\texttt{resugar}_{\text{rs}}\ T & = & \texttt{R}'_{\text{rs}}\ T & \text{when } \texttt{R}'_{\text{rs}}\ T \text{ has no opaque tags} \\
\texttt{resugar}_{\text{rs}}\ T & = & \bot & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{R}'_{\text{rs}}\ \mathbf{a} & = & \mathbf{a} \\
\texttt{R}'_{\text{rs}}\ (\texttt{Tag}\ (\texttt{Body}\ b)\ T) & = & (\texttt{Tag}\ (\texttt{Body}\ b)\ \texttt{R}'_{\text{rs}}\ T) \\
\texttt{R}'_{\text{rs}}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ T') & = & \texttt{unexp}_{\text{rs}}\ (i, \texttt{R}'_{\text{rs}}\ T')\ \sigma \\
\texttt{R}'_{\text{rs}}\ l(T_1,\ ...,\ T_n) & = & l(\texttt{R}'_{\text{rs}}\ T_1,\ ...,\ \texttt{R}'_{\text{rs}}\ T_n) \\
\texttt{R}'_{\text{rs}}\ (T_1 \ldots T_1) & = & (\texttt{R}'_{\text{rs}}\ T_1 \ldots \texttt{R}'_{\text{rs}}\ T_n)
\end{array}
$$

5.3 Lifting Evaluation

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

We have defined desugaring and resugaring with respect to terms expressed in our pattern language. Real languages' source terms do not start in this form, so we will require functions for converting between syntax in the surface and core languages and terms in our pattern language. We will call these `s->t` , `t->s` , `c->t` , and `t->c` , using `s`, `c`, and `t` 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:

$$
\begin{array}{lcl}
\texttt{desugar}^*_{\text{rs}} & = & \texttt{s->t}\ ;\ \texttt{desugar}_{\text{rs}}\ ;\ \texttt{t->c} \\
\texttt{resugar}^*_{\text{rs}} & = & \texttt{c->t}\ ;\ \texttt{resugar}_{\text{rs}}\ ;\ \texttt{t->s}
\end{array}
$$

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 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 [3], the rules obey the semantics described in this paper. The rules are also checked against the well-formedness criteria of section 5.1.3, thus ensuring that our results hold.

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.

6.1 Transformations as Lenses

We have found it helpful to view our transformation rules from the perspective of lenses [12]. In particular, the disjointness condition that prevents the `Max` problem of section 5.1.5 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 $get : C \rightharpoonup A$ and $put : A \times C \rightharpoonup C$ 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 $C = T$ and $A = (\mathbb{N}, T)$ gives $\texttt{exp}_{\text{rs}}$ and $\texttt{unexp}_{\text{rs}}$ 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.

6.1.1 The *GetPut* Law

The *GetPut* law applied to our transformations states that whenever it is well-defined,

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

Expanding the definitions produces:

$$((T/P_i) \cdot ((T/P_i)P'_i/P'_i))P_i = T$$

This law can be shown to hold without further preconditions.

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

Proof. Clearly $(T/P_i)P'_i/P'_i \subseteq T/P_i$. Thus $(T/P_i) \cdot ((T/P_i)P'_i/P'_i) = T/P_i$, and $((T/P_i) \cdot ((T/P_i)P'_i/P'_i))P_i = (T/P_i)P_i$. And since T is closed, $(T/P_i)P_i = T$. □

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

6.1.2 The *PutGet* Law

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

$$\texttt{exp}_{\text{rs}}\ (\texttt{unexp}_{\text{rs}}\ (j, T')\ T) = (j, T')$$

Expanding the definitions gives that,

$$(i, (((T/P_j) \cdot (T'/P'_j))P_j/P_i)P'_i) = (j, T')$$
$$\text{for } i = \min\{i | ((T/P_j) \cdot (T'/P'_j))P_j \geq P_i\}_i$$

This law, however, does not hold for all possible rulelists. In fact, we saw a situation in which it fails—the `Max` sugar in section 5.1.5—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 $= P_1 \to P'_1, \ldots, P_n \to P'_n$ *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:

$$(i, (((T/P_j) \cdot (T'/P'_j))P_j/P_i)P'_i) = (j, T')$$
$$\text{for } i = \min\{i | ((T/P_j) \cdot (T'/P'_j))P_j \geq P_i\}_i$$

First, note that the law always holds when $i = j$, so it is sufficient to consider $i < j$. Let $\sigma_1 = (T/P_j) \cdot (T'/P'_j)$. If the *PutGet* law does not hold, then $\sigma_1 P_j/P_i$ exists, so $P_i \vee P_j$ exists. On the other hand, if $P_i \vee P_j$ exists for some $i < j$, then T and T' can be chosen such that $\sigma_1 P_j/P_i$ 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 5.1.3 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.

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* ($\texttt{Tag}\ O\ T$).

Definition 3. *A* core term *is a term that contains no label* l *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* $\texttt{desugar}_{\text{rs}}\ T = T'$*, then* T' *is a core term. And if* $\texttt{resugar}_{\text{rs}}\ T' = T$*, then* T *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* T *is a surface term,* $\texttt{resugar}_{\text{rs}}\ T = T$*. And whenever* T' *is a core term,* $\texttt{desugar}_{\text{rs}}\ T' = T'$*.*

Proof. By induction over the term. □

Theorem 2. *Assume that the lens laws hold for all transformations. Then for all surface terms* T*,* $\texttt{desugar}_{\text{rs}}\ T = T'$ *implies* $\texttt{resugar}_{\text{rs}}\ T' = T$*. And for all core terms* T'*,* $\texttt{resugar}_{\text{rs}}\ T' = T$ *implies* $\texttt{desugar}_{\text{rs}}\ T = T'$*.*

Proof. For both cases, proceed by induction over the term. The two nontrivial cases are $\texttt{resugar}_{\text{rs}}\ (\texttt{desugar}_{\text{rs}}\ l(T_1, \ldots, T_n))$ and $\texttt{desugar}_{\text{rs}}\ (\texttt{resugar}_{\text{rs}}\ (\texttt{Tag}\ (\texttt{Head}\ i\ T)\ T'))$. For brevity, call $\texttt{desugar}_{\text{rs}}\ \texttt{Des}$, call $\texttt{exp}_{\text{rs}}\ \texttt{E}$, call $\texttt{resugar}_{\text{rs}}\ \texttt{Res}$, and call $\texttt{unexp}_{\text{rs}}\ \texttt{U}$.

In the first case,

$$\begin{aligned}
& \texttt{Res}\ (\texttt{Des}\ l(T_1, \ldots, T_n)) \\
= \ & \texttt{Res}\ (\texttt{Des}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ T')) \\
& \quad \text{when } \texttt{E}\ l(T_1, \ldots, T_n) = (i, T') \\
& \quad \text{and where } \sigma \text{ stands in for } l(T_1, \ldots, T_n) \\
= \ & \texttt{Res}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ (\texttt{Des}\ T')) \\
= \ & \texttt{U}\ (i, \texttt{Res}\ \texttt{Des}\ T')\ l(T_1, \ldots, T_n) \\
= \ & \texttt{U}\ (i, T')\ l(T_1, \ldots, T_n) && \text{(by I.H.)} \\
= \ & l(T_1, \ldots, T_n) && \text{(by } \textit{GetPut}\text{)}
\end{aligned}$$

In the second case,

$$\begin{aligned}
& \texttt{Des}\ (\texttt{Res}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ T')) \\
= \ & \texttt{Des}\ (\texttt{U}\ (i, \texttt{Res}\ T')\ \sigma) \\
= \ & \texttt{Des}\ l(T_1, \ldots, T_n) && \text{(using w.f.)} \\
& \quad \text{when } \texttt{U}\ (i, \texttt{Res}\ T')\ \sigma = l(T_1, \ldots, T_n) \\
= \ & \texttt{Des}\ (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ (\texttt{Res}\ T')) && \text{(by } \textit{PutGet}\text{)} \\
= \ & (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ (\texttt{Des}\ (\texttt{Res}\ T'))) \\
= \ & (\texttt{Tag}\ (\texttt{Head}\ i\ \sigma)\ T') && \text{(by I.H.)}
\end{aligned}$$

□

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:*

- $\texttt{t->c}\ (\texttt{c->t}\ c) = c$ *for all* c
- $\texttt{s->t}\ (\texttt{t->s}\ t) = t$ *for all* t
- $(\texttt{c->t}\ c)$ *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 $\texttt{desugar}^*_{\text{rs}} s' = c$. If s' occurs in the surface sequence, then resugaring must have succeeded with $\texttt{resugar}^*_{\text{rs}} c = s'$. Thus we simply need to show that $\texttt{desugar}^*_{\text{rs}}(\texttt{resugar}^*_{\text{rs}} c) = c$ for all terms c in the core language, i.e., that

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

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 t 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 $\texttt{resugar}_{\text{rs}}$ 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.

6.4 Machine-Checking Proofs

We have made substantial progress formalizing our transformation system in the Coq proof assistant [25]. 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 5.1.3) that pass the first static check obey the lens laws.

This formalization helped us pin down the (sometimes subtle) well-formedness criteria of section 5.1.3.

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, $\sigma[P_1, ..., P_n]$ is not a function of $\sigma P_1, ..., \sigma P_n$ when $P_1, ..., P_n$ contain ellipses.

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* [5] 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 [22]—based on A-normalization [11]—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 [4], or ways of emulating them (as discussed by Pettyjohn, et al. [22]), 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 [9] (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 [5], 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.

8. Evaluation

In this section we describe sugars we implemented to test the expressiveness of our system. (Section 4 shows a non-trivial outcome.) In what follows, we manually verified that each of the implemented sugars showed the expected surface steps.

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 non-deterministically 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 [18]. All of these behave exactly as one might expect other than `Letrec` and `Automaton`, discussed below.

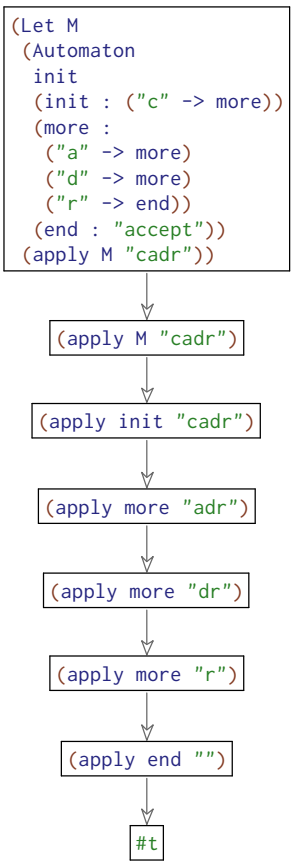

Figure 4. Automaton macro execution example

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 4 shows a run in Redex's evaluation visualizer; the underlying core evaluation took 264 steps.

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")])]);

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

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 5. Syntactic sugar in normal-mode Pyret

(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.

8.3 Pyret: A Case Study

Pyret, shown in section 4, 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 figure 5 shows, we were able to handle almost all of Pyret's sugar. An example of the result in action was shown in section 4.

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.

```
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 6. Alternate desugaring of addition

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 forces the left and right subexpressions to be evaluated fully before resolving the operation, as shown in figure 6.

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
```

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 [14]. 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 [15]. 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. [6]. 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. [7] 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. [19] 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 [8].

Clements [4, 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 [10] 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 [20] 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 sys-

tem execution (see the survey by Stevens [24]). 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 [21], 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.

Artifact Evaluation

The artifact for this paper was not submitted for evaluation because the second author was a co-chair of the evaluation process. The artifact is available from

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

Acknowledgments

We thank Daniel J. Dougherty and Matthias Felleisen for their feedback, Joe Politz for a language that offered an excellent proving ground for this work, and the anonymous reviewers who provided helpful feedback on the paper. This work was partially supported by support from the US National Science Foundation and Google.

References

[1] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Conference on Functional Programming Languages and Computer Architecture*, 1991.

[2] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, 2008.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2), 2008.

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

[5] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming Languages and Systems*, 2001.

[6] R. Culpepper and M. Felleisen. Fortifying macros. In *International Conference on Functional Programming*, 2010.

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

[8] R. K. Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. In *Lisp and Symbolic Computation*, 1988.

[9] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[10] D. Fisher and O. Shivers. Static analysis for syntax objects. In *International Conference on Functional Programming*, 2006.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Programming Languages Design and Implementation*, 1993.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Principles of Programming Languages*, 2005.

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-oriented Programming*, 2010.

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

[15] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Languages Design and Implementation*, 1992.

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

[17] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages*, 1987.

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

[19] S. Krishnamurthi, M. Felleisen, and B. F. Duba. From macros to reusable generative programming. In *Generative and Component-Based Software Engineering*, 1999.

[20] F. Lorenzen and S. Erdweg. Modular and automated type-soundness for language extensions. In *International Conference on Functional Programming*, 2013.

[21] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *International Conference on Functional Programming*, 2012.

[22] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, 2005.

[23] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6), 2010.

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

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