Resugaring: Restoring the Abstractions that Syntactic Sugar is Supposed to Provide

Justin Pombrio

February 5, 2018

Part I

SYNTACTIC SUGAR

1

SYNTACTIC SUGAR

1.1 WHAT IS IT?

The term *syntactic sugar* was introduced by Peter Landin in 1964[CITE]. 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, but is not discriminating enough to be a useful definition. I will define syntactic sugar as follows:

> 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.

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 I am introducing for the restoration of information that was lost during desugaring.

1.2 WHAT IS IT GOOD FOR?

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. 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 can just 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 thing 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[CITE] 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 force 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 sugars are not needed 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.

1.2.1 *A Thousand Grains of Sugar*

There are many axes on which desugaring mechanisms vary:

1. Desugaring is a syntax-to-syntax transformation, but what is the representation of syntax? There is a big difference between transformations on the *text* of the program vs. its *concrete surface syntax* vs. its *abstract surface syntax*.

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

3. 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?

1 It may sound like I am suggesting that everything should be manipulatable at run-time. I am 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.

4. In what order are constructs desugared? Most importantly, are nested sugars evaluated from the *innermost to outermost*, or from *outermost to innermost*?

5. How many *phases* of desugaring are there? Can there be more than one?

6. How *safe* is it? Can desugaring produce syntactically invalid code? Can it produce an unbound variable, or accidentally capture a variable? Can it produce code that contains a type error?

There is one big cluster of desugaring systems that should be called out by name: macro systems. They can be loosely defined as:

> *Macros* are user-defined sugars.

In practice, macro systems tend to share several features: (i) by definition, they are user-defined; (ii) the metalanguage is the programming language itself [CHECK]; and (iii) the evaluation order is usually outside-in [CHECK]. [CHECK: others?]

1.2.2 *Syntax 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: I discuss these in 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. I discuss this further in [REF].

1.2.3 *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 comprehen-

sions 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. The difference between the two is whether sugar is only a convenient method of simplifying language design, or whether users are given the power to extend the language themselves.

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 mostly avoid the problem by mostly not having syntax. They don't *entirely* avoid the problem, though. 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: resugaring in general]

[TODO: macro-defining-macros]

1.2.4 *Metalanguage*

1.2.5 *Evaluation Strategy*

There are two major evaluation strategies used in desugaring systems. They loosely correspond to eager and lazy runtime evaluation, but differ in some important ways described below, so I will instead refer to them by their original names [CITE]: Outside-in (01) and Inside-out (10) evaluation:

01 evaluation is similar to lazy evaluation. However, it has an unusual property: [FILL]

10 evaluation is similar to eager evaluation. It has [FILL]

1.2.6 *Number of Phases*

1.2.7 *Safety*

[TODO]: AST safety, scope-safety, type-safety

2

DESUGARING IN THE WILD

.[TODO: State version of each system discussed]

2.1 C PREPROCESSOR

Evaluation Strategy: 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 arguments `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 C++ TEMPLATES

Representation: Concrete Syntax
Authorship: User-defined
Evaluation Strategy: IO
Safety: [FILL]
Discussion: C++ templates [CITE] are not general-purpose sugars, because they cannot take code as an argument. Instead, they 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 arguments 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 arguments. 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 evaluation 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.

2.3 RUST MACROS

Representation: Concrete Syntax
Authorship: User-defined
Evaluation Strategy: OI
Safety: [FILL]
Discussion:

2.4 HASKELL TEMPLATES

Representation: Concrete or Abstract Syntax
Authorship: User-defined
Evaluation Strategy: IO
Safety: AST safe. Scope unsafe. Type unsafe.
Discussion: NOTES:

- Must be defined in separate file

3

RESUGARING FOR PYRET

3.1 EXAMPLE

3.1.1 *Define-struct*

CORE AST

```
Stmts:
| [{splicing-begin stmts:Stmts} @rest:Stmts]
   binding stmts in rest
   providing stmts, rest

| [{let x:Var v:Expr} @rest:Stmts]
   binding x in rest

| [{fun f:Var args:Args body:Expr} @rest:Stmts]
   binding args in body, rest in body
   providing f, rest
```

```
ALTERNATIVELY:

Stmts:
| {splicing-begin stmts:Stmts rest:Stmts}
   binding stmts in rest
   providing stmts, rest

| {let x:Var v:Expr rest:Stmts}
   binding x in rest

| {fun f:Var args:Args body:Expr rest:Stmts}
   binding args in body, rest in body
   providing f, rest

| {end}
```

```
Params:
| {param x:Var rest:Params}
| {end}
```

AUXILIARY AST

```
IStructFields:
| [field:IStructField ...fields:IStructFields]
  providing field, fields

IStructField:
| {i-struct-field field:Str get:Var set:Var}
  providing get, set
```

SURFACE AST

```
SurfStmts:
  .....
| [(define-struct name:Var fields:StructFields) @rest:SurfStmts]
  binding name in rest, fields in rest
  providing name, fields, rest

StructFields:
| [field:StructField ...fields:StructFields]
  providing field, fields

StructField:
| (struct-field field:Str get:Var set:Var)
  providing get, set
```

DESUGARING RULES

```
   [(struct-field field:Str get:Var set:Var) @rest:IStructFields]
=> [{i-struct-field field get set} @rest]

   [(define-struct name:Var
      [(struct-field field:Str get:Var set:Var) ...]) @rest:SurfStmts]
=> [{fun name [x ...] {record [{record-field field x} ...]}}
    {splicing-begin [{fun get [rec] {record-get rec field}} ...]}
    {splicing-begin [{fun set [rec val] {record-set rec field val}} ...]}
    @rest]
```

3.1.2 *Pyret For Expressions*

To handle Pyret for-expressions, we need to do two things. First, when a for-expression binding (e.g. `n from 0`) desugars, it will simply return its binding (`n`) and its value (`0`) to the for-expression. It can do so with the desugaring rule:

```
    (s-for-bind l:Loc b:Bind v:Expr)
=> {for-bind b v}
```

where `ForBind` is a new type:

```
  ForBind ::= {for-bind Bind Expr}

with list scope:
  [{for-bind b v} ...]
  export b
  export ...
```

Then for-expressions can be implemented with the desugaring rule:

```
   (s-for l:Loc
          iter:Expr
          [{ForBind bind:Bind value:Expr} ...]@binds
          ann:Ann
          body:Expr
          blocky:Bool)
=> {Lambda l (CONCAT "for-body<" (FORMAT l false) ">")
     [] [bind ...] ann "" body None None blocky}
with scope:
  bind binds in body
```

Notice that `s-for` is pattern matching against the results of desugaring the `s-for-binds`. The `(CONCAT ...)` stuff is to compute at compile time a name for this lambda, which is what Pyret currently does.

3.2 EXPRESSIONS

core name C	::=	*name*	core syntactic construct name
surface name m	::=	*name*	surface syntactic construct name
expression e	::=	$\{C\, e_1 ... e_n\}$	core syntactic construct
	|	$(m\, e_1 ... e_n)$	surface syntactic construct
	|	$[e_1 ... e_n]$	list
	|	*string*	string literal
	|	$x_i^{\mathrm{R}} \mid x_i^{\mathrm{D}}$	variable
value v	::=	e	with no sugar invocations

3.3 EXPANSION

ellipsis label l	::=	*name*	ellipsis label
pattern p	::=	α	pattern variable
	|	$\{C\ p_1 ... p_n\}$	syntactic construct
	|	$(m\ p_1 ... p_n)$	sugar invocation
	|	$[ps]$	list
	|	*string*	string literal
	|	$x_i^{\mathrm{R}} \mid x_i^{\mathrm{D}}$	variable
seq. pattern ps	::=	ϵ	empty sequence
	|	p, ps	cons
	|	$p *\_l$	ellipsis with label l
fresh vars F	::=	$\{x, ...\}$	fresh variable set
type env. Γ	::=	$\begin{cases} \alpha : t, ... \\ i \mapsto [\Gamma], ... \end{cases}$	
substitution γ	::=	$\begin{cases} \alpha \mapsto e, ... \\ l \mapsto [\gamma ... \gamma], ... \\ x \mapsto x, ... \end{cases}$	

rewrite case c	::=	$(p_1, ..., p_k); \Gamma; F \Rightarrow p'$
desugaring rule r	::=	$\texttt{sugar}\ m = \{c_1, ..., c_n\}$
desugaring rules rs	::=	$\{r_1, ..., r_n\}$

3.3.1 *Matching and Substitution*

Lemma 1 (matching and substitution)**.** *Matching and substitution are inverses:*

Proof. Induct on p. [FILL] □

3.3.2 *Expansion*

See fig. 2. [TODO: Replace step with something that looks like desugaring.] [TODO: Replace v with something that looks like core terms.]

[FILL] One expansion rule. Note expansion contexts.

$\boxed{F \vdash e/p = \gamma}$

$$\text{m-pvar}\ \frac{}{F \vdash e/\alpha = \{\alpha \mapsto e\}} \qquad \text{m-str}\ \frac{}{F \vdash string/string = \{\}}$$

$$\text{m-capture}\ \frac{x \notin F}{F \vdash x/x = \{\}} \qquad \text{m-empty}\ \frac{}{F \vdash [\,]/\epsilon = \{\}}$$

$$\text{m-fresh}\ \frac{x \in F}{F \vdash y/x = \{x \mapsto y\}} \qquad \text{m-cons}\ \frac{\begin{array}{c}F \vdash e_1/p = \gamma_1 \\ F \vdash [e_2, ..., e_n]/ps = \gamma_s \\ \gamma_1 \dot\cup \gamma_2 = \gamma\end{array}}{F \vdash [e_1 ... e_n]/p, ps = \gamma}$$

$$\text{m-con}\ \frac{F \vdash e_1/p_1 = \gamma_1\ \cdots\ F \vdash e_n/p_n = \gamma_n \gamma_1 \dot\cup ... \dot\cup \gamma_n = \gamma}{F \vdash \{C\, e_1 ... e_n\}/\{C\, p_1 ... p_n\} = \gamma}$$

$$\text{m-star}\ \frac{F \vdash e_1/p = \gamma_1\ \cdots\ F \vdash e_n/p = \gamma_n}{F \vdash [e_1 ... e_n]/[p *\_ l] = \{l \mapsto [\gamma_1 ... \gamma_n]\}}$$

$\boxed{F \vdash \gamma \bullet p = e}$

$$\text{s-pvar}\ \frac{\alpha \mapsto e \in \gamma}{F \vdash \gamma \bullet \alpha = e} \qquad \text{s-fresh}\ \frac{x \in F \qquad x \mapsto y \in \gamma}{F \vdash \gamma \bullet x = y}$$

$$\text{s-str}\ \frac{}{F \vdash \gamma \bullet string = string} \qquad \text{s-empty}\ \frac{}{F \vdash \gamma \bullet [\epsilon] = [\,]}$$

$$\text{s-capture}\ \frac{x \notin F}{F \vdash \gamma \bullet x = x} \qquad \text{s-cons}\ \frac{\begin{array}{c}F \vdash \gamma \bullet p = e_1 \\ F \vdash \gamma \bullet [ps] = [e_2, ..., e_n]\end{array}}{F \vdash \gamma \bullet [p, ps] = [e_1 e_2 ... e_n]}$$

$$\text{s-star}\ \frac{\begin{array}{c}l \mapsto [\gamma_1 ... \gamma_n] \in \gamma \\ F \vdash \gamma_1 \bullet p = e_1\ \cdots\ F \vdash \gamma_n \bullet p = e_n\end{array}}{F \vdash \gamma \bullet [p *\_ l] = [e_1 ... e_n]}$$

$$\text{s-con}\ \frac{F \vdash \gamma \bullet p_1 = e_1\ \cdots\ F \vdash \gamma \bullet p_n = e_n}{F \vdash \gamma \bullet \{C\, p_1 ... p_n\} = \{C\, e_1 ... e_n\}}$$

$$\text{s-sugar}\ \frac{F \vdash \gamma \bullet p_1 = e_1\ \cdots\ F \vdash \gamma \bullet p_n = e_n}{F \vdash \gamma \bullet (m\, p_1 ... p_n) = (m\, e_1 ... e_n)}$$

Figure 1: Matching and Substitution

$$\text{eval-ctx}\ \frac{L \vdash e \rightsquigarrow e'}{L \vdash E[e] \rightsquigarrow E[e']}$$

$$\text{eval-expand}\ \frac{\begin{array}{c} L = G, rs \qquad \texttt{sugar}\ m = \{c_1 ... c_n\} \in G \\ L \vdash (m\, e_1 ... e_n) \rightsquigarrow e'' \text{ by case } c_i \\ L \not\vdash (m\, e_1 ... e_n) \rightsquigarrow e' \text{ by case } c_j \text{ for any } j < i \end{array}}{L \vdash (m\, e_1 ... e_n) \rightsquigarrow e'}$$

$$\text{eval-case}\ \frac{\begin{array}{c} L \vdash e_i / p_i = \gamma_i \text{ for each } i \\ \gamma' \text{ gives fresh names to the variables in } F \\ \gamma_1 \dot\cup ... \dot\cup \gamma_n \dot\cup \gamma' = \gamma \\ F \vdash \gamma \bullet p' = e' \end{array}}{L \vdash (m\, e_1 ... e_n) \rightsquigarrow e' \text{ by case } (p_1, ..., p_n); \Gamma; F \Rightarrow p'}$$

Figure 2: Expansion

3.4 AST CHECKING

ast defn. G	::=	$A \mapsto \{t_1, ...t_n\}$	(with no production $A_1 \mapsto A_2$)
syntactic category A	::=	*name*	
syntax type t	::=	A	syntactic category
	|	$\{C\, t_1 ... t_n\}$	syntactic construct
	|	$[t]$	list
	|	String	string literal
	|	Decl	variable declaration
	|	Refn	variable reference
language L	::=	G, rs	

Lemma: If rules grammar check, then e obeys Surf implies ds(e) obeys Core.

Lemma: Normalizing a grammar does not change its language.

EXHAUSTION CHECKING We perform exhaustion checking to make sure that sugars cover all possible cases of their arguments, but do not give the algorithm here. It works by looking at *shapes*: a shape is a pattern that contains types in place of pattern variables. It is straightforward to check whether an expression matches a shape, and to convert a pattern into a shape. Exhaustion checking uses the fact that the expressions that do *not* match a shape can be expressed as a union of shapes. [TODO: prove]

$\boxed{L \vdash e : t}$

$$\text{e-con}\ \frac{A \mapsto \{C\, t_1 ... t_n\} \in L \quad L \vdash e_1 : t_1 \ \cdots\ L \vdash e_n : t_n}{L \vdash \{C\, e_1 ... e_n\} : A} \qquad \text{e-str}\ \frac{}{L \vdash \mathit{string} : \text{String}}$$

$$\text{e-list}\ \frac{L \vdash e_1 : t \ \cdots\ L \vdash e_n : t}{L \vdash [e_1 ... e_n] : [t]}$$

$$\text{e-refn}\ \frac{}{L \vdash x^{\text{R}} : \text{Refn}}$$

$$\text{e-decl}\ \frac{}{L \vdash x^{\text{D}} : \text{Decl}}$$

$$\text{e-sugar}\ \frac{L \vdash m : t_1, ..., t_n \to t \quad L \vdash e_1 : t_1 \ \cdots\ L \vdash e_n : t_n}{L \vdash (m\, e_1 ... e_n) : t}$$

$\boxed{L; \Gamma \vdash p : t}$

$$\text{p-pvar}\ \frac{\alpha : t \in \Gamma}{L; \Gamma \vdash \alpha : t} \qquad \text{p-sugar}\ \frac{L \vdash m : t_1, ..., t_n \to t \quad L; \Gamma \vdash p_1 : t_1 \ \cdots\ L; \Gamma \vdash p_n : t_n}{L; \Gamma \vdash (m\, p_1 ... p_n) : t}$$

$$\text{p-refn}\ \frac{}{L; \Gamma \vdash x : \text{Refn}} \qquad \text{p-empty}\ \frac{}{L; \Gamma \vdash [\epsilon] : [t]}$$

$$\text{p-decl}\ \frac{}{L; \Gamma \vdash x : \text{Decl}}$$

$$\text{p-str}\ \frac{}{L; \Gamma \vdash \mathit{string} : \text{String}} \qquad \text{p-cons}\ \frac{L; \Gamma \vdash p : t \quad L; \Gamma \vdash [ps] : [t]}{L; \Gamma \vdash [p, ps] : [t]}$$

$$\text{p-con}\ \frac{A \mapsto \{C\, t_1 ... t_n\} \in L \quad L; \Gamma \vdash p_1 : t_1 \ \cdots\ L; \Gamma \vdash p_n : t_n}{L; \Gamma \vdash \{C\, p_1 ... p_n\} : A} \qquad \text{p-star}\ \frac{l \mapsto [\Gamma'] \in \Gamma \quad L; \Gamma' \vdash p : t}{L; \Gamma \vdash [p\ *\_l] : [t]}$$

$$\texttt{sugar}\ m = \begin{cases} (p_{11}, ..., p_{1n}); \Gamma_1; F_1 \Rightarrow p'_1 \\ \quad ... \\ (p_{k1}, ..., p_{kn}); \Gamma_k; F_k \Rightarrow p'_k \end{cases} \in L$$

The cases are exhaustive over $t_1, ..., t_n$ in G

$L; \Gamma_i \vdash p_{ij} : t_j$ for each $i \in 1..k, j \in 1..n$

$L; \Gamma_i \vdash p'_i : t$ for each $i \in 1..k$

$$\boxed{L \vdash m : t, ..., t \to t}\ \text{g-sugar}\ \frac{}{L \vdash m : t_1, ..., t_n \to t}$$

$$\boxed{L \vdash \gamma : \Gamma}\ \gamma\text{-env}\ \frac{\begin{array}{c} L \vdash e_1 : t_1 \ \cdots\ e_n \vdash t_n : \\ L \vdash \gamma_{11} : \Gamma_1 \ \cdots\ L \vdash \gamma_{1j} : \Gamma_1 \\ ... \\ L \vdash \gamma_{m1} : \Gamma_m \ \cdots\ L \vdash \gamma_{mk} : \Gamma_m \end{array}}{L \vdash \begin{cases} \alpha_1 \mapsto e_1, ..., \alpha_n \mapsto e_n \\ i_1 \mapsto [\gamma_{11}, ..., \gamma_{1j}] \\ ... \\ i_m \mapsto [\gamma_{m1}, ..., \gamma_{mk}] \end{cases} : \Gamma', \begin{cases} \alpha_1 : t_1, ..., \alpha_n : t_n, \\ i_1 \mapsto [\Gamma_1], ..., i_m \mapsto [\Gamma_m], \end{cases}}$$

Figure 3: Grammar Checking

3.4.1 *Type Soundness*

We prove soundness by way of progress + preservation:

Theorem 1 (Soundness). *If $L \vdash e : t$, then $L \vdash e \rightsquigarrow^* v$ where $L \vdash v : t$, or e runs forever.*

Proof. Lemma 2 (progress) and lemma 3 (preservation) together imply that either: (i) e is a value, or (ii) $L \vdash e \rightsquigarrow e'$ and $L \vdash e' : t$. Apply this repeatedly. Either e eventually steps to a value v, and has remained the same type t throughout the evaluation, or e never halts. □

Lemma 2 (Progress). *If $L \vdash e : t$, then $L \vdash e \rightsquigarrow e'$, or e is a value.*

Proof. First, verify that our evaluation contexts include every case that isn't a value. Thus either e is a value and we are done, or e contains a redex: $e = E[(m\, e_1 ... e_n)]$. In the latter case, we will show that e can take a step because the eval-expand rule applies. There are two premises that need to be satisfied:

- First, we must show that m is bound in L. Since e type-checked, it must be: the only rule which can type-check a sugar invocation is p-sugar; this in turn must use rule g-sugar; finally g-sugar requires that $m \in L$.
- Second, we must show that the pattern match of $e_1, ..., e_n$ succeeds on any case $(p_1, ..., p_n); \Gamma; F$ of the desugaring rule. By assumption 1, it does.

□

Assumption 1 (Exhaustion). *If the set of cases in a desugaring rule are exhaustive over $t_1, ..., t_n$ according to our exhaustion checking algorithm, then for every possible argument list $e_1, ..., e_n$ that matches the given types (i.e., $L \vdash e_1 : t_1, ..., L \vdash e_n : t_n$), there is a case c_i such that $e_1, ..., e_n$ successfully matches against c_i. [TODO: prove]*

Proof. Not given. We have not stated our exhaustion checking algorithm here, and so cannot prove it correct. We believe it is straightforward (if tedious). □

Lemma 3 (Preservation). *If $L \vdash e : t$ and $L \vdash e \rightsquigarrow e'$, then $L \vdash e' : t$.*

Proof. Since e can take an expansion step, it must have a redex (via eval-ctx): $e = E[(m\, e_1 ... e_n)]$. And furthermore (by eval-expand) m must be bound in L, and there must be a first case of m that matches e. Call it $c_i = (p_1, ..., p_n); \Gamma \Rightarrow p'$. Then:

By eval-case:	$L \vdash e_i / p_i = \gamma_i$ for some γ_i for each i	(1)
and	$F \vdash \gamma_i \dot\cup ... \dot\cup \gamma_n \bullet p' = e'$	(2)
and	$L \vdash E[e] \rightsquigarrow E[e']$	
By e-sugar:	$L \vdash (m\, e_1 ... e_n) : t$	
and	$L \vdash m : t_1 ... t_n \to t$	
and	$L \vdash e_i : t_i$ for each i	(3)
By g-sugar:	$L; \Gamma \vdash p_i : t_i$ for each i	(4)
and	$L; \Gamma \vdash p' : t$	(5)

By lemma 5 with (1), (3), and (4), $\gamma_i \vdash \Gamma :$ for each i. By lemma 4, $\gamma_1 \dot\cup ... \gamma_n \vdash \Gamma :$. Finally, by lemma 6 with that last fact together with (2) and (5), $L \vdash e' : t$. □

Lemma 4 (Union of Substitutions)**.** *If* $L \vdash \gamma_1 : \Gamma$ *and* $L \vdash \gamma_2 : \Gamma$*, then* $L \vdash \gamma_1 \dot\cup \gamma_2 : \Gamma$.

TODO. □

Lemma 5 (Matching)**.** *If* $L; \Gamma \vdash p : t$ *and* $L \vdash e : t$ *and* $F \vdash e/p = \gamma$*, then* $L \vdash \gamma : \Gamma$

Proof. Induction on p.

$p = string$

By p-str:	$L; \Gamma \vdash string : \text{String}$	fixes t
By m-str:	$F \vdash string/\text{String} = \{\}$	fixes γ

Finally, by γ-env, $F \vdash \{\} : \Gamma$ (this applies for any Γ).

$p = x \notin F$ (Analogous.)

$p = x \in F$ By p-refn or p-decl, $\Gamma = \{\}$ and t is Refn or Decl. By m-fresh, $e = y$ for some fresh name y, and $\gamma = \{\}$. And the conclusion follows: $L \vdash \{\} : \{\}$. [TODO]

$p = \alpha$

By p-pvar:	$L; \Gamma \vdash \alpha : t$	fixes t
and	$\alpha : t \in \Gamma$	(1)
By m-pvar:	$F \vdash e/\alpha = \{\alpha \mapsto e\}$	fixes γ

Finally, using γ-env on the premise $L \vdash e : t$ gives that $L \vdash \gamma : \{\alpha : t\}, \Gamma'$ for any Γ'. By (1), this is the form of Γ, so we can set Γ' such that $\Gamma = \{\alpha : t\}, \Gamma'$, and we are done.

$p = \{C\ p_1 \ldots p_n\}$

By p-con:	$L; \Gamma \vdash \{C\ p_1 ... p_n\} : A$	fixes t
and	$A \mapsto \{C\ t_1 ... t_n\} \in L$	
and	$L; \Gamma \vdash p_i : t_i$ for each i	(1)
By m-con:	$F \vdash \{C\ e_1 ... e_n\} / \{C\ p_1 ... p_n\} = \gamma$	fixes e
and	$F \vdash e_i / p_i = \gamma_i$ for each i	(2)
and	$\gamma = \gamma_1 \dot\cup ... \dot\cup \gamma_n$	
By e-con:	$L \vdash \{C\ e_1 ... e_n\} : A$	
and	$L \vdash e_i : t_i$ for each i	(3)

Applying the I.H. to (1), (2), and (3) yeilds that $L \vdash \gamma_i : \Gamma$. By lemma 4, $L \vdash \gamma : \Gamma$.

$p = (m\ p_1 \ldots p_n)$ [FILL]

$p = [\epsilon]$ [TODO] By m-empty, $\gamma = \{\}$. By p-empty, $\Gamma = \{\}$. The goal follows: $L \vdash \{\} : \{\}$.

$p = [p, ps]$ [FILL]

$p = [p\ *\_l']$

By p-star:	$L; \Gamma \vdash p\ *\_l' : [t]$	fixes t
and	$l' \mapsto [\Gamma'] \in \Gamma$	(1)
and	$L; \Gamma' \vdash p : t$	(2)
By m-star:	$F \vdash [e_1 ... e_n] / [p\ *\_l] = \{l' \mapsto [\gamma_1 \dot\cup ... \gamma_n]\}$	fixes e, γ
and	$F \vdash e_i / p = \gamma_i$ for each i	(3)
By e-list:	$L \vdash [e_1 ... e_n] : [t]$	
and	$L \vdash e_i : t_i$	(4)

By the I.H. together with (2), (3), and (4), $L \vdash \gamma_i : \Gamma'$ for each i. By lemma 4, $L \vdash \gamma_1 \dot\cup ... \gamma_n : \Gamma'$. Finally, by γ-env, $L \vdash \{l' \mapsto [\gamma_1 \dot\cup ... \gamma_n]\} : \{l' \mapsto [\Gamma']\}$, which is compatible with the specification of Γ in (1).

□

Lemma 6 (Substitition)**.** *If $L \vdash \gamma : \Gamma$ and $L; \Gamma \vdash p : t$, and $F \vdash \gamma \bullet p = e$, then $L \vdash e : t$.*

Proof. Induction on p.

$p = string$

$p = string$ By s-str, $F \vdash \gamma \bullet p = string$, so $e = string$. By p-str, $L; \Gamma \vdash p :$ String, so $t =$ String. Finally, by e-str, $L \vdash e :$ String as desired.

$p = x \notin F$ (Analogous.)

$p = x \in F$ By s-fresh, $e = y$ for some fresh name y. By p-refn or p-decl, t is either Refn or Decl. Our goal $L \vdash y : t$ follows by either e-refn or e-decl, respectively.

$p = \alpha$ By rule s-pvar, $\alpha \mapsto e \in \gamma$. By γ-env, $\alpha \mapsto t \in \Gamma$ and $L \vdash e : t$. Which is our goal; we are done. (Note that by γ-env, Γ may have many *other*, unnecessary, bindings to pattern variables, but it must *at least* contain a correct binding for α.)

$p = \{C\ p_1 \ldots p_n\}$

By p-con:	$L; \Gamma \vdash \{C\ p_1 ... p_n\} : A$	fixes t
and	$A \mapsto \{C\ t_1 ... t_n\} \in L$	(1)
and	$L; \Gamma \vdash p_i : t_i$ for each i	(2)
By s-con:	$F \vdash \gamma \bullet \{C\ p_1 ... p_n\} = \{C\ e_1 ... e_n\}$	fixes e
and	$F \vdash \gamma \bullet p_i = e_i$ for each i	(3)

Using the I.H. with (2) and (3) gives that $L \vdash e_i : t_i$ for each i. Using e-con on that fact together with (1) gives that $L \vdash \{C\ e_1 ... e_n\} : A$, so we are done.

$p = (m\ p_1 \ldots p_n)$

By s-sugar:	$F \vdash \gamma \bullet (m\ p_1 ... p_n) = (m\ e_1 ... e_n)$	fixes e
and	$F \vdash \gamma \bullet p_i = e_i$ for each i	(1)
By p-sugar:	$L; \Gamma \vdash (m\ p_1 ... p_n) : t$	fixes t
and	$L; \Gamma \vdash p_i : t_i$ for each i	(2)
and	$L \vdash m : t_1, ..., t_n \to t$	(3)

Using the I.H. with (1) and (2) gives that $L \vdash e_i : t_i$ for each i. Finally, using e-sugar on that fact together with (3) gives that $L \vdash (m\ e_1 ... e_n) : t$.

$p = [\epsilon]$ By s-empty, $F \vdash \gamma \bullet p = []$, so $e = []$. By p-empty, $L; \{\} \vdash [\epsilon] : [t]$ (for some t). Finally, by e-list, $L \vdash [] : [t]$.

$p = [p, ps]$

By s-cons:	$F \vdash \gamma \bullet p = e_1$	(1)
and	$F \vdash \gamma \bullet [ps] = [e_2 ... e_n]$	(2)
and	$F \vdash \gamma \bullet [p, ps] = [e_1 e_2 ... e_n]$	fixes e
By p-cons:	$L; \Gamma \vdash p : t$	(3)
and	$L; \Gamma \vdash [ps] : [t]$	(4)
and	$L; \Gamma \vdash [p, ps] : [t]$	fixes t

We can apply the I.H. using (1) and (3) and the assumption $L \vdash \gamma : \Gamma$ to get that $L \vdash e_1 : t$. Likewise, the I.H. with (2) and (4) gives $L \vdash [e_2 ... e_n] : [t]$. By e-list (in reverse), $L \vdash e_2 : t \cdots L \vdash e_n : t$. Finally, by e-list (forward), $L \vdash [e_1 e_2 ... e_n] : [t]$.

$p = [p\ *\_l]$

By s-star:	$F \vdash \gamma \bullet [p\ *\_l] = [e_1 ... e_n]$	fixes e
and	$l \mapsto [\gamma_1 ... \gamma_n] \in \gamma$	
and	$F \vdash \gamma_i \bullet p = e_i$ for each i	(1)
By γ-env:	$l \mapsto [\Gamma'] \in \Gamma$	
and	$L \vdash \gamma_i : \Gamma'$ for each i	(2)
By p-star:	$L; \Gamma \vdash [p\ *\_l] : [t]$	fixes t
and	$L; \Gamma' \vdash p : t$	(3)

Using the I.H. with (1), (2), and (3) proves that $L \vdash e_i : t$. Then, by e-list, $L \vdash [e_1 ... e_n] : [t]$ as desired.

□

3.5 SCOPE CHECKING

(See fig. 4.)

$$\text{scope-e-decl}\ \frac{\boxed{\Sigma \vdash e : \{x^{\mathrm{D}}\}; \{x^{\mathrm{R}}\}; \{x^{\mathrm{D}}\}; \{x^{\mathrm{R}} \mapsto x^{\mathrm{D}}\}}}{\Sigma \vdash x^{\mathrm{D}} : \{x^{\mathrm{D}}\}; \{\}; \{x^{\mathrm{D}}\}; \{\}}$$

$$\text{scope-e-refn}\ \frac{}{\Sigma \vdash x^{\mathrm{R}} : \{\}; \{x^{\mathrm{R}}\}; \{\}; \{\}}$$

$$\text{scope-e-con}\ \frac{\begin{array}{c}\Sigma[C] = \sigma \\ \Sigma \vdash e_i : P_i; R_i; B_i;\ \text{for } i \in 1..n \\ S = \{x_a^{\mathrm{D}} \mapsto x_b^{\mathrm{D}} \mid x_a^{\mathrm{D}} \in P_i,\ x_b^{\mathrm{D}} \in P_j,\ \texttt{bind}\ j\ \texttt{in}\ i \in \sigma\} \\ B = \{x_a^{\mathrm{R}} \mapsto x_b^{\mathrm{D}} \mid x_a^{\mathrm{R}} \in R_i,\ x_b^{\mathrm{D}} \in P_j,\ \texttt{bind}\ j\ \texttt{in}\ i \in \sigma,\ x_b^{\mathrm{D}} \notin \mathit{domain}(S)\} \\ R = \{x_a^{\mathrm{R}} \mid x_a^{\mathrm{R}} \in R_i,\ \nexists x_b^{\mathrm{D}}.\ x_a^{\mathrm{R}} \mapsto x_b^{\mathrm{D}} \in B\}\} \\ P = \{x_a^{\mathrm{D}} \mid x_a^{\mathrm{D}} \in P_i,\ \texttt{provide}\ i \in \sigma,\ x_a^{\mathrm{D}} \notin \mathit{domain}(S)\}\end{array}}{\Sigma \vdash \{C\, e_1 ... e_n\} : P; R; B;}$$

$$\text{scope-p-con}\ \frac{\begin{array}{c}\Sigma[C] = \sigma \\ \Sigma \vdash p_i : P_i; R_i; B_i;\ \text{for } i \in 1..n \\ S = \{a \mapsto b \mid a \in P_i,\ b \in P_j,\ \texttt{bind}\ j\ \texttt{in}\ i \in \sigma,\ F \vdash a \sim_{\mathit{shadow}} b\} \\ B = \{a \mapsto b \mid a \in R_i,\ b \in P_j,\ \texttt{bind}\ j\ \texttt{in}\ i \in \sigma,\ b \notin \mathit{domain}(S),\ F \vdash a \sim_{\mathit{bind}} b\} \\ R = \{a \mid a \in R_i,\ (\nexists b.\ a \mapsto b \in B\} \text{ or } a \text{ is a pattern var})\} \\ P = \{a \mid a \in P_i,\ \texttt{provide}\ i \in \sigma,\ a \notin \mathit{domain}(S)\}\end{array}}{\Sigma \vdash \{C\, p_1 ... p_n\} : P; R; B;}$$

Two checks to make: fresh vars don't bind to non-fresh vars, and named vars only bind to vars of the same name:

$$\frac{}{F \vdash x_1^{\mathrm{R}} \sim_{\mathit{bind}} x_2^{\mathrm{D}}} \qquad \frac{x \notin F}{F \vdash \alpha \sim_{\mathit{bind}} x_1^{\mathrm{D}}} \qquad \frac{}{F \vdash x_1^{\mathrm{D}} \sim_{\mathit{shadow}} x_2^{\mathrm{D}}}$$

$$\frac{x \notin F}{F \vdash x_1^{\mathrm{R}} \sim_{\mathit{bind}} \alpha} \qquad \frac{}{F \vdash \alpha \sim_{\mathit{bind}} \beta} \qquad \frac{x \notin F}{F \vdash x_1^{\mathrm{D}} \sim_{\mathit{shadow}} \alpha}$$

Figure 4: Scope Checking Rules

4

APPENDIX

4.0.1 *Terms*

We will call AST terms *expressions* and write them in s-expression form. Atomic terms are either variables or literals (i.e. syntactic constants), and compound terms are built with *term constructors* P:

$$\begin{array}{lcll} e & ::= & \textbf{lit} & \text{(literal)} \\ & | & \mathrm{x} & \text{(variable)} \\ & | & (P\, e_1 ... e_n) & \text{(AST node)} \end{array}$$

4.0.2 *Tree Grammars*

A *tree grammar* [CITE] is to trees as a context-free grammar is to strings. Thus it can be viewed either as a set of instructions for how to iteratively and nondeterministically rewrite a starting *nonterminal* to a final tree; *or* it can be viewed as a specification of a grammar that a tree may or may not follow. We will take the latter view.

Definitionally, a tree grammar consists of a number of *productions* that map *nonterminals* s to *patterns*:

$$G \quad ::= \quad \begin{cases} s_1 \leftarrow \mathit{pattern}_1 \\ \ldots \\ s_n \leftarrow \mathit{pattern}_n \end{cases}$$

$$\begin{array}{lcll} \mathit{pattern} & ::= & (P\, s_1 ... s_n) & \text{(regular pattern)} \\ & | & (P\, s_1 ... s_n\, s^*_{n+1}) & \text{(ellipses pattern)} \\ & | & \texttt{literal} & \text{(matches literals)} \\ & | & \texttt{var} & \text{(matches variables)} \end{array}$$

The *meaning* of a tree grammar (again, we are viewing the grammar as a *specification*) is that for each production "$s \leftarrow \mathit{pattern}$", if a term matches the *pattern*, then it also matches the nonterminal s. Formally:

$$\boxed{G \vdash e : s}$$

$$\text{G-literal}\ \frac{}{G \vdash \textbf{lit} : \texttt{literal}} \qquad \text{G-variable}\ \frac{}{G \vdash \texttt{x} : \texttt{variable}}$$

$$\text{G-node}\ \frac{\forall i \in 1..n.\ G \vdash e_i : s_i \qquad s \leftarrow (P\, s_1 \ldots s_n) \in G}{G \vdash (P\, e_1 \ldots e_n) : s} \qquad \text{G-ellipses}\ \frac{\forall i \in 1..n.\ G \vdash e_i : s_i \qquad \forall j \in 1..k.\ G \vdash e_{n+j} : s \qquad s \leftarrow (P\, s_1 \ldots s_n\, s^*) \in G}{G \vdash (P\, e_1 \ldots e_n \ldots e_{n+k}) : s}$$