Inferring Type Rules for Syntactic Sugar

Justin Pombrio
Brown University
Providence, RI, United States
justinpombrio@cs.brown.edu

Shriram Krishnamurthi
Brown University
Providence, RI, United States
sk@cs.brown.edu

Abstract

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

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

Keywords Programming Languages, Syntactic Sugar, Macros, Type Systems, Resugaring

ACM Reference Format:
Justin Pombrio and Shriram Krishnamurthi. 2018. Inferring Type Rules for Syntactic Sugar. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192398

1 Introduction

Syntactic sugar is a central tool in defining programming languages and systems. It follows a longstanding tradition of separating the definition into two parts: a (small) core language and a rich set of convenient and powerful syntax defined in terms of that core. (In this paper we will use *surface* to refer to the language the programmer sees, and *core* for the target of desugaring.) It is now actively used in many practical settings:

- To define useful constructs (such as comprehensions) in many languages such as Python and Haskell.
- Following the Lisp tradition, to enable programmers to extend the language; languages such as Scala, Julia, and Rust offer macro-like facilities.
- To enable tractable semantics for large scripting languages that have many special-case behaviors, such as JavaScript and Python [10, 20, 21].

Overall, syntactic sugar enables a smart trade-off by keeping the language tractable for the language's engineers while making the language convenient for the language's users. It is worth noting that this trade-off does not depend on the language providing syntactic extensibility (à la macros): the sugar could be built into the language itself.

This trade-off, however, depends on the abstraction provided by sugar not leaking [26]. The code generated by desugaring can be large and complicated, creating an onerous comprehension burden; it may even use features of the core language that the user does not know. Therefore, programmers using sugar must not be forced to confront the details of sugar; they should only confront the core language when they use it directly.

Desugaring and Type Checking In this paper, we focus on the interaction between desugaring and *type checking*. Type checking occurs either before or after desugaring, and there can be major problems with each.

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

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

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192398

Inferring a Surface Type System We offer a way out of this dilemma. Given typing rules for the core language, and syntactic sugar written as pattern-based rules, we show how to *infer* type rules for the surface language.

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

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

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

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

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

2 Type Resugaring

Our overall aim is to be able to generate type judgments for the surface language given desugaring rules and judgments for the core. To distinguish this from ordinary type inference, in which *types* are inferred *within a program*, we will call our inference process **type resugaring**[1], in which *type rules* are inferred *through syntactic sugar*. We wish to obtain type rules for the surface language that are faithful to the core language type rules: type checking using resugared type rules should produce the same result as first desugaring and then type checking using the core type rules. Specifically, if $\overline{I_{core}}$ are the core language type rules and $\overline{I_{surf}}$ are the resugared surface type rules, then

Goal 1.

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

[1] We take the term "resugaring" from our previous work [22].

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

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

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

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

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

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

Expanding out the sugar:

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

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

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

and perhaps also its converse:

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

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

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

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

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

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

Since the extension type rules (like t-and$^{\rightarrow}$) always succeed, any type errors will be found in the core language. For example, if the first argument to and was not a boolean, this will be discovered by the t-if rule, not by the t-and$^{\rightarrow}$rule! Thus, while the augmented type system technically obeys goal 1, it breaks the abstraction that ought to be provided by syntactic sugar. Type errors made in the surface language should be reported with respect to surface language constructs. This can be achieved with a second goal:

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

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

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

where the sub-derivations D_α and D_β depend on α and β. Notice that the rest of the derivation is constant: *every* type-derivation of α and β has this form. Thus there is no reason to re-derive it every time we type-check. Instead, we can remove this "cruft" to obtain a simpler type rule for and:

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

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

The important step was determining the "template" derivation. We presented it above without fanfare, but how can it automatically be discovered? Let us look into this with a slightly more complex example, an or sugar[2]:

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

As before, we want to find a derivation for the sugar's RHS (right-hand-side). That is, we should search for a derivation of the judgment:

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

We can begin by applying the core type rules, obtaining a partial derivation, shown at the top of fig. 1. However, the core type rules (unsurprisingly) cannot prove the judgments about pattern variables (marked with ?). Each pattern variable stands for an unknown surface term, so its derivation will vary between different uses of the or sugar. Since we do not know what type it will have, we will assign it a globally fresh type variable, using the rule t-premise:

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

(This rule will be refined in section 4.2 and section 4.3.) We write this rule with a dashed line because it is in a sense incomplete: it serves as a placeholder for a subderivation that would be filled in if the pattern variable were instantiated. Using this rule—together with a t-fresh rule that will be introduced in section 4.3—finishes the derivation, giving the bottom derivation in fig. 1.

As seen, pattern variables introduce type variables. Solving for these type variables in general requires unification. We therefore split the search for a derivation: first we find a potential derivation with equality constraints (as in fig. 1), then we solve these constraints (via an ordinary unification algorithm). Solving the constraints of fig. 1 gives the substitution $\{A = \texttt{Bool}, B = \texttt{Bool}\}$. Finally, gathering the premises and conclusion of the derivation and applying the substitution to them produces the type rule for or:

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

Our Overall Approach Putting all this together, we can describe our type resugaring algorithm. For each desugaring rule, such as the or sugar from above:

1. Construct a generic type judgment from the sugar's RHS, e.g. $\Gamma \vdash \texttt{(let x = } \alpha\ \texttt{in if x then x else } \beta\texttt{)} : t$
2. Search for a derivation of this judgment using the core language type rules plus the t-premise rule from above. Fail if no derivation, or if multiple derivations, are found. For example, this will find the derivation shown in fig. 1.
3. Gather the equality constraints from the derivation. Additionally, if multiple premises (i.e., judgments proved by the t-premise rule) are of the same expression, add equality constraints that these expressions have the same type. Solve the unification problem. (If there are any unconstrained variables, they become free variables in the type rule.)
   For example, in or, there are two equality constraints: $A = \texttt{Bool}$ and $A = B$. The t-premise rule is used only once for α and once for β, so no additional constraints are needed. The solution is $\{A = \texttt{Bool}, B = \texttt{Bool}\}$.
4. Form a type rule whose premises are the judgments proved by t-premise from the derivation in step (2), and whose conclusion is a generic type judgment from the sugar's LHS. Apply the unification from step (3). This is the resugared surface type rule.

We have implemented a prototype of this approach, called SweetT. SweetT is written in Racket [8] (racket-lang.org),

[2] The let in the desugaring of or prevents the duplicate evaluation of α.

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

$$\dfrac{\dfrac{\text{t-premise}\ \dfrac{}{\Gamma \vdash \alpha : A}\ \text{t-let} \qquad \dfrac{\text{t-id}\ \dfrac{}{\Gamma, x : A \vdash \mathtt{x} : A} \qquad A = \mathtt{Bool} \qquad \text{t-id}\ \dfrac{}{\Gamma, x : A \vdash \mathtt{x} : A} \qquad \dfrac{\text{t-premise}\ \dfrac{}{\Gamma \vdash \beta : B}}{\text{t-fresh}\ \dfrac{}{\Gamma, x : A \vdash \beta : B}} \qquad A = B}{\Gamma, x : A \vdash \mathtt{if\ x\ then\ x\ else}\ \beta : B}\ \text{t-if}}{\Gamma \vdash \mathtt{let\ x\ =}\ \alpha\ \mathtt{in\ if\ x\ then\ x\ else}\ \beta : B}$$

Figure 1. Derivation of `or`. Top: an incomplete derivation. Bottom: a complete derivation, using `t-premise`.

and makes use of the semantics engineering tool Redex [6]. *All of the examples in this paper run in our implementation*, albeit with a different, more parenthetical, syntax.

SweetT is available for download at http://cs.brown.edu/research/plt/dl/pldi2018/.

3 Theory

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

- Desugaring rules must be defined using pattern-based rules, and their LHSs must be disjoint (section 3.1).
- The type system used to resugar must support pattern variables and partial derivations, and they must obey some obvious laws (section 3.2).
- The core language type rules must be syntax directed (also section 3.2). This will fail, for instance, on a type system with non-algorithmic subtyping.
- Our implementation of SweetT must be correct (section 3.3). (As must Redex, which we use to find derivations.)
- Finally, SweetT's unification algorithm must be able to handle the sugars given. Section 4.5 gives an example of extending it.

The rest of this section describes these assumptions in more detail. As a prelude, fig. 2 provides a guide to the notation we will use throughout the paper.

Notation Explanation

e	::=	k	(atomic expression)
	|	$(P\ e_1 \ldots e_n)$	(compound expression of syntactic category P)
$\widetilde{e}$	::=	k	
	|	$(P\ e_1 \ldots e_n)$	
	|	α	(pattern variable)
t	::=	*type*	(type)
Γ	::=	$\cdot \mid \Gamma, x : t$	(type environment)
J	::=	$\Gamma \vdash \widetilde{e} : t$	(type judgment)
I	::=	$J_1 \ldots J_n / J$	(inference rule)
$\overline{I}$	::=	$I_1 \ldots I_n$	(set of inference rules)
σ	::=	$\{\alpha \mapsto e, \ldots\}$	(substitution)
$\mathcal{L}$	::=	$\{\widetilde{e} \Rightarrow \widetilde{e}', \ldots\}$	(desugaring rules)

Our approach relies on being able to use $\widetilde{e}$ in two different ways: (i) from one perspective $\widetilde{e}$ is one side of a desugaring rule, and any α inside is a pattern variable; (ii) from the other perspective, $\widetilde{e}$ is an expression inside a type rule, in which α is a metavariable. The convention of the first perspective is to call $\widetilde{e}$ as C, but we choose instead to use $\widetilde{e}$ to emphasize the other perspective.

In addition, to the above notation, we will also write:

- $\overline{I} \Vdash J$ to mean that judgment J is provable under inference rules $\overline{I}$ (i.e., there is a derivation that proves J).
- $\overline{I} \Vdash J_1 \ldots J_n \to J$ to mean that there is a derivation that proves J with unproven leaves J_i.
- $\overline{I} \Vdash J_1 \ldots J_n \,/\, J$ to mean that there is a derivation of depth 1 that proves J with unproven premises J_i.
- $(\sigma \bullet \widetilde{e})$ to denote applying substitution σ to expression $\widetilde{e}$.

Figure 2. Notation explanation

3.1 Requirements on Desugaring

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

$$\begin{array}{lcl}
\mathbb{D}_{\mathcal{L}}(k) & = & k \\
\mathbb{D}_{\mathcal{L}}(\alpha) & = & \alpha \\
\mathbb{D}_{\mathcal{L}}(\sigma \bullet \widetilde{e}) & = & (\mathbb{D}_{\mathcal{L}}(\sigma)) \bullet (\mathcal{L}[\widetilde{e}]) \\
 & & \text{if } \widetilde{e} = (P\ \widetilde{e}_1 \ldots \widetilde{e}_n), \\
 & & \text{and } P \text{ is in the surface langauge} \\
\mathbb{D}_{\mathcal{L}}(P\ e_1 \ldots e_n) & = & (P\ \mathbb{D}_{\mathcal{L}}(e_1) \ldots \mathbb{D}_{\mathcal{L}}(e_n)) \\
 & & \text{if } P \text{ is in the core language}
\end{array}$$

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

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

$$\begin{array}{lcl} \mathbb{D}_{\mathcal{L}}(\Gamma \vdash \widetilde{e} : t) & = & \Gamma \vdash \mathbb{D}_{\mathcal{L}}(\widetilde{e}) : t \\ \mathbb{D}_{\mathcal{L}}(\{x \to \widetilde{e}, \ldots\}) & = & \{x \to \mathbb{D}_{\mathcal{L}}(\widetilde{e}), \ldots\} \end{array}$$

Unsurprisingly, substitution distributes over pattern-based desugaring:

Lemma 3.1 (Distributivity of Substitution and Desugaring)**.**

$$\mathbb{D}_{\mathcal{L}}(\sigma \bullet J) = \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J)$$

Proof. Let $J = \Gamma \vdash \widetilde{e} : t$. By definition, $\mathbb{D}_{\mathcal{L}}(\Gamma \vdash \widetilde{e} : t) = \Gamma \vdash \mathbb{D}_{\mathcal{L}}(\widetilde{e}) : t$. Induct on $\widetilde{e}$.

Base case: it is an atomic expression k:
$\mathbb{D}_{\mathcal{L}}(\sigma \bullet k) = k = \mathbb{D}_{\mathcal{L}}(\sigma) \bullet k$.

Base case: it is a pattern variable α:
$\mathbb{D}_{\mathcal{L}}(\sigma \bullet \alpha) = \mathbb{D}_{\mathcal{L}}(\sigma[\alpha]) = (\mathbb{D}_{\mathcal{L}}(\sigma))[\alpha] = \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \alpha = \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(\alpha)$.

Inductive case: it is a compound term $\{\alpha_1 \mapsto e_1, \ldots\} \bullet \widetilde{e}$, where $\widetilde{e}$ is the LHS of a desugaring rule:

$$\begin{array}{lll} & \mathbb{D}_{\mathcal{L}}(\sigma \bullet (\{\alpha_1 \mapsto e_1, \ldots\} \bullet \widetilde{e})) & \\ = & \mathbb{D}_{\mathcal{L}}(\{\alpha_1 \mapsto (\sigma \bullet e_1), \ldots\} \bullet \widetilde{e}) & \text{(substitution)} \\ = & \{\alpha_1 \mapsto \mathbb{D}_{\mathcal{L}}(\sigma \bullet e_1), \ldots\} \bullet \widetilde{e'} & \text{where } \mathcal{L}[\widetilde{e}] = \widetilde{e'} \\ = & \{\alpha_1 \mapsto (\mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(e_1)), \ldots\} \bullet \widetilde{e'} & \text{(I.H.)} \\ = & \mathbb{D}_{\mathcal{L}}(\sigma) \bullet (\{\alpha_1 \mapsto \mathbb{D}_{\mathcal{L}}(e_1), \ldots\} \bullet \widetilde{e'}) & \text{(substitution)} \\ = & \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(\{\alpha_1 \mapsto e_1, \ldots\} \bullet \widetilde{e}) & \text{(desugar)} \end{array}$$

□

We also assume that the LHSs of each desugaring rule are disjoint, so that there is never any ambiguity as to which resugaring rule to apply. That is:

Assumption 1 (Unique Desugaring)**.** *For every pair $\widetilde{e_1}$ and $\widetilde{e_2}$ of sugar LHSs, there are no substitutions σ_1 and σ_2 such that $\sigma_1 \bullet \widetilde{e_1} = \sigma_2 \bullet \widetilde{e_2}$.*

This is everything we need of desugaring.

3.2 Requirements on the Type System

Let us now change focus to the type system. In the and example in section 2, we made implicit assumptions about the core type system. We stated that *every* type derivation of (α and β) must share a common template, and we implicitly assumed that this template could not depend on α or on β. This is certainly not true of every conceivable type system. Type resugaring will rely on three assumptions about the type system in order to make the approach we outlined work.

Before we describe these assumptions, notice that the type derivations found by resugaring (e.g., in fig. 1) contain pattern variables. Thus the type system used by resugaring is not exactly the language's type system: it is an extension of the type system that handles pattern variables (and partial derivations, discussed shortly). It is this extended type system we will be discussing in this section. With that said, we can state the assumptions.

First, we will assume that the type system supports pattern variables: it must be possible to search for type derivations of a judgment whose term contains pattern variables. Furthermore, a judgment with pattern variables must hold iff that judgment holds under all substitutions for those pattern variables:

Assumption 2 (Substitution into Derivations)**.** *A derivation (possibly containing pattern variables) is provable iff it is provable under all substitutions:*

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

Likewise for rules:

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

Next, we assume that the type system supports *partial derivations* that may contain unjustified judgments in their leaves, which we will call their *premises*. If a partial derivation is provable, and its premises are provable, then its conclusion must also be provable:

Assumption 3 (Composition of Derivations)**.** *The composition of provable derivations is provable:*

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

Finally, we would like the core type system to be *deterministic* in a particular way. Say that a judgment is *abstract* if it contains pattern variables, or *concrete* otherwise. We would like that if an *abstract* partial derivation $J_1 \ldots J_n \to J$ applies to a *concrete* judgment $\sigma \bullet J$ that can be proven, then the proof of $\sigma \bullet J$ *must* use $J_1 \ldots J_n \to J$, and thus prove as intermediate steps $\sigma \bullet J_i$ for each $i \in 1..n$. Formally, we define determinism as:

Definition 3.2 (Determinism)**.** A set of inference rules $\overline{I}$ is *deterministic* when, for any concrete judgment $\sigma \bullet J$:

If $\overline{I} \Vdash \sigma \bullet J$ and $\overline{I} \Vdash J_1 \ldots J_n \to J$, then $\overline{I} \Vdash \sigma \bullet J_i$ for each $i \in 1..n$.

Instead of assuming outright that the core language is deterministic, we can prove it from a more conservative assumption. We will assume that there is never any ambiguity as to which type rule applies to a concrete judgment J, i.e., that the type system is syntax directed:

Assumption 4 (Syntax Directedness)**.** *At most one type rule in $\overline{I_{core}}$ ever applies to a concrete judgment J.*

Under this assumption, the core language can be proven deterministic. This will be essential for our proof of goal 1.

Lemma 3.3 (Determinism)**.** *Suppose that at most one type rule in $\overline{I}$ ever applies to a concrete judgment J. Then $\overline{I}$ is deterministic.*

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

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

- $\overline{I} \Vdash \sigma \bullet J'_i$, and
- There is a subset $J_{i_1} \ldots J_{i_l}$ of $J_1 \ldots J_n$ such that $\overline{I} \Vdash J_{i_1} \ldots J_{i_l} \to J'_i$. Since each judgment $J_1 \ldots J_n$ must be used in the derivation $\overline{I} \Vdash J$, the union of these subsets must be the full set $J_1 \ldots J_n$.

For each $i \in 1..m$, by the inductive hypothesis, $\overline{I} \Vdash \sigma \bullet J_{i_1} \ldots \sigma \bullet J_{i_l}$. Since the union of these sets is $\sigma \bullet J_1 \ldots \sigma \bullet J_n$, we are done.

(Note that in the base case, $n = 0$, and the result is vacuously true.) □

Corollary 3.4 (Core Determinism). *If a core language $\overline{I_{core}}$ obeys assumption 4 (syntax-directedness), then it is deterministic.*

Proof. Follows directly from the lemma, together with assumption 4 (syntax-directedness). □

3.3 Requirements on Resugaring

Our final set of requirements is on the behavior of the type resugaring algorithm. Thus it is essentially a specification for our implementation: SweetT is correct iff it obeys the requirements of this subsection.

Let us look at what it means to successfully resugar a desugaring rule $\widetilde{e}_{\text{LHS}} \Rightarrow \widetilde{e}_{\text{RHS}} \in \mathcal{L}$. Resugaring will search for a partial derivation of the sugar's RHS:

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

where $J_1 \ldots J_n$ are provable using the `t-premise` rule and J_{RHS} has the form $J_{\text{RHS}} = \Gamma \vdash \widetilde{e}_{\text{RHS}} : t$.[3] If such a derivation is found, and is unique, then we will write:

$$\mathbb{R}(\overline{I_{core}},\ \widetilde{e}_{\text{LHS}} \Rightarrow \widetilde{e}_{\text{RHS}}) = J_1 \ldots J_n / J_{\text{LHS}}$$

where $J_{\text{LHS}} = \Gamma \vdash \widetilde{e}_{\text{LHS}} : t$, and we will add the type rule $J_1 \ldots J_n / J_{\text{LHS}}$ to $\overline{I_{surf}}$. Therefore:

Assumption 5 (Resugaring). *Suppose that $\mathbb{R}(\overline{I_{core}},\ \widetilde{e}_{LHS} \Rightarrow \widetilde{e}_{RHS}) = J_1 \ldots J_n / (\Gamma \vdash \widetilde{e}_{LHS} : t)$. Then:*

$$\overline{I_{core}} \Vdash J_1 \ldots J_n \to \mathbb{D}_{\mathcal{L}}(\Gamma \vdash \widetilde{e}_{LHS} : t)$$

[3] Our implementation uses Redex's `build-derivations` function to perform this search.

This is the correctness criterion for resugaring.

For the upcoming proof, we will also need that the surface language be deterministic in the sense of definition 3.2. This is provable using assumption 1 (unique-sugar):

Lemma 3.5 (Surface Determinism). *If resugaring succeeds, then I_{surf} is* deterministic. *Repeating the definition of determinism, this means that:*

If $I_{surf} \Vdash \sigma \bullet J$ and $I_{surf} \Vdash J_1 \ldots J_n \to J$, then $I_{surf} \Vdash \sigma \bullet J_i$ for each $i \in 1..n$.

Proof. To start, we will show that at most one resugared type rule may apply to a concrete judgment J. Suppose, for the sake of contradiction, that two distinct rules apply, with conclusions J_1 and J_2. Let the expressions in J, J_1, and J_2 be e, e_1, and e_2 respectively. Since both rules can be applied to J, there must be substitutions σ_1 and σ_2 such that $\sigma_1 \bullet J_1 = \sigma_2 \bullet J_2 = J$. Thus $\sigma_1 \bullet e_1 = \sigma_2 \bullet e_2 = e$. However, this contradicts assumption 1 (unique-sugar). Thus at most one type rule in I_{surf} may apply to a concrete judgment.

Then, by lemma 3.3, I_{surf} is deterministic. □

3.4 Main Theorem

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

Theorem 3.6. *Grant assumptions 1–5 from this section, let $\mathcal{L} = \widetilde{e}_{LHS} \Rightarrow \widetilde{e}_{RHS}, \ldots$, and suppose that $\overline{I_{surf}} = \mathbb{R}(\overline{I_{core}},\ \widetilde{e}_{LHS} \Rightarrow \widetilde{e}_{RHS}), \ldots$. Then for all surface type judgments J_{surf}:*

$$\overline{I_{surf}} \Vdash J_{surf} \text{ iff } \overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(J_{surf})$$

Proof. Given in fig. 3. □

Furthermore, resugaring obeys goal 2, essentially by construction:

Lemma 3.7. *Resugaring obeys goal 2: type rules for surface constructs never mention core constructs.*

Proof. Let $\mathbb{R}(I_{core},\ \widetilde{e}_{\text{LHS}} \Rightarrow \widetilde{e}_{\text{RHS}}) = J_1 \ldots J_n / J_{\text{LHS}}$ be any surface rule. We aim to show that $J_1 \ldots J_n$ and J_{LHS} do not mention core constructs P. By assumption 5 (resugaring), $I_{core} \Vdash J_1 \ldots J_n \to \mathbb{D}_{\mathcal{L}}(J_{\text{LHS}})$, where $J_1 \ldots J_n$ are all provable using `t-premise`. We gave the `t-premise` rule in section 2, and generalize it in section 4.3 and section 4.4. However, in *all* of its versions, the judgment must be over a surface term. Thus $J_1 \ldots J_n$ do not mention core constructs.

Finally, the expression in J_{LHS} is the LHS of a desugaring rule, and is thus by definition a surface term. Therefore, given our assumptions listed in this section, resugaring obeys goal 2. □

4 Desugaring Features

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

Proof. Split on the "iff".

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

	$\overline{I_{surf}} \Vdash J_{surf}$	assumption
iff	$\overline{I_{surf}} \Vdash \sigma \bullet J_0$	equality
implies	$\overline{I_{surf}} \Vdash \sigma \bullet J_i$ for $i \in 1..n$	by lemma 3.5 (surface determinism)
implies	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_i)$ for $i \in 1..n$	inductive hypothesis

Also:

	$\overline{I_{core}} \Vdash J_1 \ldots J_n \,/\, \mathbb{D}_{\mathcal{L}}(J_0)$	by assumption 5 (resugaring)
implies	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma) \bullet J_1 \ldots \mathbb{D}_{\mathcal{L}}(\sigma) \bullet J_n \,/\, \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_0)$	by assumption 2 (substitution)
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_1) \ldots \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_n) \,/\, \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_0)$	since $\mathbb{D}_{\mathcal{L}}(J_i) = J_i$
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_1) \ldots \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_n) \,/\, \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_0)$	by lemma 3.1 (distributivity)
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_1) \ldots \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_n) \,/\, \mathbb{D}_{\mathcal{L}}(J_{surf})$	equality

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

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

	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(J_{surf})$	assumption
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_0)$	equality
implies	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_0)$	by lemma 3.1 (distributivity)
Also:	$\overline{I_{core}} \Vdash J_1 \ldots J_n \to \mathbb{D}_{\mathcal{L}}(J_0)$	by assumption 5 (resugaring)
implies	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma) \bullet J_1 \ldots \mathbb{D}_{\mathcal{L}}(\sigma) \bullet J_n$	by corollary 3.4 (core determinism)
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_1) \ldots \mathbb{D}_{\mathcal{L}}(\sigma) \bullet \mathbb{D}_{\mathcal{L}}(J_n)$	since $\mathbb{D}_{\mathcal{L}}(J_i) = J_i$
iff	$\overline{I_{core}} \Vdash \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_1) \ldots \mathbb{D}_{\mathcal{L}}(\sigma \bullet J_1)$	by lemma 3.1 (distributivity)
implies	$\overline{I_{surf}} \Vdash \sigma \bullet J_1 \ldots \sigma \bullet J_n$	inductive hypothesis

Also:

	$\overline{I_{surf}} \Vdash J_1 \ldots J_n \,/\, J_0$	by assumption 5 (resugaring)
implies	$\overline{I_{surf}} \Vdash \sigma \bullet J_1 \ldots \sigma \bullet J_n \,/\, \sigma \bullet J_0$	by assumption 2 (substitution)
iff	$\overline{I_{surf}} \Vdash \sigma \bullet J_1 \ldots \sigma \bullet J_n \,/\, J_{surf}$	equality

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

Figure 3. Proof of theorem 3.6.

4.1 Calculating Types

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

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

What is the missing type? It needs to match the type of α, but there is no way to express this using the kind of desugaring rules we have presented so far. We therefore extend the desugaring language with a feature called `calc-type`. In this example, it can be used as follows:

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

This binds the type variable X to the type of α in the rest of the desugaring.

In general, `calc-type` may be used in expression position on the RHS of a desugaring rule, and its meaning is that:

$$\texttt{calc-type } \widetilde{e_1} \texttt{ as } t \texttt{ in } \widetilde{e_2}$$

desugars to $\widetilde{e_2}$, in which the type t has been unified with the type of $\widetilde{e_1}$, thus allowing the free type variables of t to be used in $\widetilde{e_2}$.[4] Notice that this requires desugaring and type checking to be interspersed. This is not surprising, since the desugaring of `let` involves determining a type.

This feature needs to be reflected in our type system. We do so with the type rule:

[4] `calc-type` can also be used to force a more specific surface type rule than would be inferred. For example, `(calc-type α as List<X> in ...)` will lead to a surface type rule that enforces that α is a list. This is used in the Haskell list comprehension example of section 6.2 and in the `or` example of section 4.5.

$$\text{t-calc-type}\ \frac{\Gamma \vdash \widetilde{e_1} : t_1 \qquad t_1 = t \qquad \Gamma \vdash \widetilde{e_2} : t_2}{\Gamma \vdash (\texttt{calc-type}\ \widetilde{e_1}\ \texttt{as}\ t\ \texttt{in}\ \widetilde{e_2}) : t_2}$$

With this type rule, we can find a type derivation for `let`, shown in fig. 4. It leads to the type rule:

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

Here we can see an advantage of type resugaring. As noted above, to type check `let` in the core, type checking and desugaring must be interspersed. However, to type check `let` in the surface, only this resugared rule is needed.

4.2 Recursive Sugars

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

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

This sugar, unlike those we have seen up to this point, is defined recursively: its RHS contains a list comprehension. Our resugaring algorithm, as described so far, will fail to find a type derivation for this sugar. It will get to the judgment $\Gamma \vdash \texttt{[}\alpha\ \texttt{|}\ \gamma\texttt{]} : \_$, but lack any way to prove this judgment, because the `t-premise` rule does not match.

Our solution is to generalize the `t-premise` rule to allow any judgment about a surface term to be accepted as a premise. Notice that the term $\texttt{[}\alpha\ \texttt{|}\ \gamma\texttt{]}$ is a surface term: when desugaring, pattern variables such as α and γ will only ever be bound to surface terms, and thus they themselves should be considered part of the surface language. We therefore refine the `t-premise` rule as:

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

Furthermore, this is the most general rule we can make: goal 2 states that surface type rules must never mention core constructs, so `t-premise` can allow judgments over surface terms but nothing more.

4.3 Fresh Variables

Take the sugar `const`, which produces a constant function:

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

It is important that x be given a fresh name, or else this sugar might accidentally capture a user-defined variable called x which is used in α. This is easy to add to desugaring: each desugaring rule will specify a set of "capturing" variables that are *not* freshly generated, and all other introduced variables will be given fresh names.[5] (We use a capturing rather than fresh set to choose hygiene by default.)

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

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

to remove unnecessary fresh variables from the type environment, and by modifying `t-premise` to only work on judgments so limited:[6]

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

What exactly is `t-fresh` saying? It is a form of weakening, but with two extra restrictions. First, the variables being weakened are variables that will be given fresh names during desugaring. Second, the expression e is a surface term. Together, these imply that e cannot contain $x_1 \ldots x_n$, so it *should* be safe to remove them from Γ. One way this could fail is if the language does not admit weakening, for example if it has a linear type system. We therefore assume that:

Assumption 6. *The rule:*

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

is admissible in the core type system.

This rule can be used to "reverse" any use of `t-fresh`, so if it is admissible then applying `t-fresh` greedily can never lead a derivation into a dead end.

4.4 Globals

Sugars may rely on library functions. For instance, Haskell's list comprehension sugar makes use of the library function `concatMap` (which is `map` followed by list concatenation). We therefore allow the declaration of "global" names, together with their type, with the understanding that this name will be available to the desugared code (with the given type).[7]

The declared globals effectively form a primordial type environment, available in conjunction with the ordinary type environment. For example, if `+` desugars into a call to a global `plus`, the type rule for `+` is actually (using `N` as shorthand for `Number`):

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

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

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

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

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

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

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

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

Figure 4. Type derivation of `let`

4.5 Variable Arities

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

e^*	::=	ϵ	empty sequence
	|	(cons e e^*)	nonempty sequence
	|	α	pattern variable

SweetT supports these sequences by providing:

- The above grammar production, allowing a language's grammar to refer to e^*.
- Proper handling of sequences in the unification algorithm, allowing them to be resugared.
- Built-in operations for accessing the n'th element of a sequence, and for asserting that a type judgment holds for all expressions in a sequence.

SweetT likewise supports sequences of types, t^*, and records of both expressions and types.

Using this feature, a simple variable-arity `or` sugar can have production rule (`or` e^*), and desugaring rules:

(or (cons α ϵ)) ⇒ α
(or (cons α (cons β γ)))
⇒ if α then true else (or (cons β γ))

Type resugaring produces one type rule for each desugaring rule:

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

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

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

(or (cons α ϵ)) ⇒ calc-type α as Bool in α

at which point the resugared type rule becomes:

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

as probably desired.

5 Implementation

We have implemented a prototype of our tool in PLT Redex [6], a semantics engineering tool. It can be found at cs.brown.edu/research/plt/dl/pldi2018/. Among other features, Redex allows one to define judgment forms, and given a judgment form can search for derivations of it.

SweetT takes as input:

- The syntax of a language, given as a grammar in Redex.
- Core language type rules, defined as a judgment form in Redex. We require that these rules be written using equality constraints: if two premises in a type rule would traditionally describe equality by repeating a type variable, SweetT instead requires that the rule be written using two different type variables, with an equality constraint between them—thus making the unification explicit.[8]
- Desugaring rules, given by a LHS and RHS. Each rule has a *capture list* of variables to be treated unhygienically, as described in section 4.3, and the RHS of a rule may make use of `calc-type`, as described in section 4.1.
- Type definitions of globals, as described in section 4.4.

SweetT then provides a `resugar` function that follows the process outlined at the end of section 2, together with the extensions described in section 4. If `resugar` succeeds, it produces the resugared type rule, as well as the derivation which led to it. If it fails, it announces that no derivation was found (or, less likely, that more than one was found, in violation of assumption 4 (syntax-directedness)).

Assumption 5 (resugaring) is essentially a specification for `resugar`, and we believe our implementation obeys this property. We provide empirical evidence for this fact, and for the power of SweetT, in the next section.

6 Evaluation

There is no standard benchmark for work in this area. Therefore, we evaluate our approach in two ways. First, we try resugaring on a number of sugars we create atop existing *type systems*, to ensure that it can support that variety of type systems. Second, we show some *case studies* which validate that it can handle interesting sugars.

[8] This is necessary because re-using the same type variable would invoke Redex's pattern-matching algorithm. This is usually sufficient, because Redex is meant to type a complete term. However, we are typing a partial term, and instead need a more general unification algorithm. So instead, SweetT gathers equations and performs unification itself.

6.1 Type Systems

We evaluate SweetT by implementing a number of type systems from Types and Programming Languages (TAPL [3]). We tested the type systems in Part II of TAPL (except for references, pg. 167), as well as two later systems (subtyping and existentials). Altogether, this is:

- Booleans (pg. 93)
- Numbers (pg. 93)
- Simply Typed Lambda Calculus (pg. 103)
- Unit (pg. 119)
- Ascription (pg. 122)
- Let binding (pg. 124)
- Pairs (pg. 126)
- Tuples (pg. 128)
- Records (pg. 129)
- Sums (pg. 132)
- Variants (pg. 136)
- General recursion (pg. 144)
- Lists (pg. 147)
- Error handling (pg. 174)
- Algorithmic subtyping (pg. 212)
- Existential types (pg. 366)

We tested each type system by picking one or more sugars that made use of its features, resugaring them to obtain type rules, and validating the resulting type rules by hand. All of them resugared successfully. The full version of the paper will provide an appendix with complete details.

Three type systems required extending SweetT's unification algorithm: records and lists needed built-in support, as described in section 4.5, and subtyping required adding subtyping constraints, as well as a new `t-sub-premise` rule.[9] References (pg. 167) would have required changing the form of judgments, from $\Gamma \vdash e : t$ to $\Gamma, \Sigma \vdash e : t$ where Σ is a store environment, which would be a more extensive change.

6.2 Case Studies

We describe six case studies below.

The first three are simpler than the rest. We describe them briefly, and show them in fig. 6. For each, the figure first shows the relevant core language type rules, then the sugar, then its core derivation, and finally the resugared type rule. To make them fit, we show all of the derivations *after* unification, eliminating equality constraints.

The last three case studies are more complex, so we discuss them more but do not show their type derivations (which do not fit on a page).

Letrec The `letrec` sugar (fig. 6) introduces recursive bindings using λ and `fix` (the fixpoint operator).

λret The `λret` sugar (fig. 6) implements `return` in functions using TAPL-style exceptions (using `String` as the fixed exception type). The variable `return` is marked as capturing in the sugar, and thus appears explicitly in the resulting type rule.

Upcast The `upcast` sugar (fig. 6) converts an expression to a supertype of its type via η-expansion. Notice that the core language type system contains subtyping judgments, as mentioned in section 6.1.

Foreach We consider a functional `foreach` loop, that performs a map on a list, and also provides `break` within the loop. If `break` is called, the loop halts and returns the elements processed so far. Its desugaring is:

```
foreach x list body
=>
letrec loop : ((List a) -> (List b) -> (List b)) =
  (λ (lst : (List a)) (acc : (List b))
    if (isnil lst)
    then acc
    else
      try
        let break = (λ (_ : Unit) raise "") in
        let x = head lst in
        loop (tail lst) (cons body acc)
      with (λ (_ : String) acc))
in reverse (loop list nil)
```

where `reverse` is a global (section 4.4) with type `[i] -> [i]`, and where `list` and `body` are pattern variables (instead of α and β, as in the rest of the paper). In addition, this sugar is declared to capture the variable `break` (see section 4.3).

The resugared type rule for `foreach` is show in fig. 5. It demonstrates how different variables must be handled. In the desugaring, when `body` is used, several variables are in scope: `loop`, `lst`, `acc`, `break`, and `x`. However, in the resugared type rule, only `break` and `x` are in scope in the judgment for `body`: `x` because it is an argument to the sugar, and `break` because it is declared as capturing.

Haskell List Comprehensions List comprehensions [18, section 3.11] are given by the following transformation:

```
[e | True]          = [e]
[e | q]             = [e | q, True]
[e | b, Q]          = if b then [e | Q] else []
[e | p <- l, Q]     = let ok p = [e | Q]
                          ok _ = []
                      in concatMap ok l
[e | let decls, Q] = let decls in [e | Q]
```

A Haskell list comprehension has the form `[e | Q]`, where `e` is an expression and `Q` is a list of *qualifiers*. There are three kinds of qualifiers, which are visible in the rules above: (i) boolean guards `b` perform a filter; (ii) generators `p <- l` perform a map; and (iii) `let decls` declare local bindings.

[9] The `t-sub-premise` rule is like `t-premise`, but for subtyping judgments instead of type judgments.

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

Figure 5. `foreach` type rule

We will ignore first two rules (which are uninteresting base cases), and focus on the last three, that introduce qualifiers. To resugar these three kinds of qualifiers, we declare `concatMap` as a global with type `(i -> [o]) -> [i] -> [o]`, as described in section 4.4. We also simplify the generator desugaring to consist of a single variable binding, because that is what is available in the TAPL core language we are desugaring to. Finally, we use `calc-type` (section 4.1) to determine the type of elements in generators. Thus, we are resugaring these slightly modified rules (using Greek letters for pattern variables to match this paper's notation):

```
[α | β, γ] = if β then [α | γ] else []
[α | x <- β, γ] = calc-type β as [t] in
    concatMap (\(x :: t) -> [α | γ]) β
[α | let x = β, γ] = (let x = β in [α | γ])
```

SweetT resugars these rules, producing the following type rules (transcribed into Haskell syntax)[10]:

$$\text{t-hlc-guard}\ \frac{\Gamma \vdash [\alpha \mid \gamma] : \mathrm{C} \qquad \Gamma \vdash \beta : \mathit{Bool}}{\Gamma \vdash [\alpha \mid \beta,\ \gamma] : \mathrm{C}}$$

$$\text{t-hlc-gen}\ \frac{\Gamma, x : t \vdash [\alpha \mid \gamma] : [\mathrm{o}] \qquad \Gamma \vdash \beta : [\mathrm{t}]}{\Gamma \vdash [\alpha \mid x \texttt{ <- } \beta,\ \gamma] : [\mathrm{o}]}$$

$$\text{t-hlc-let}\ \frac{\Gamma, x : A \vdash [\alpha \mid \gamma] : \mathrm{B} \qquad \Gamma \vdash \beta : \mathrm{A}}{\Gamma \vdash [\alpha \mid \texttt{let}\ x = \beta,\ \gamma] : \mathrm{B}}$$

Newtype Let us now look at a desugaring of `new-type` into existential types. The core language will have constructs for packing and unpacking existentials:

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

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

We define a `new-type` sugar that presents a concrete type `T` as an abstract type `X`, and provides wrapping and unwrapping functions (with user-chosen names) that convert from `T` to `X` and from `X` to `T` respectively. The desugaring is:

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

[10] "hlc" stands for "Haskell list comprehension".

where `id` is a global (section 4.4) identity function.

This sugar is successfully resugared to give the type rule:

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

Notice that this type rule does not mention existentials in any way, thereby hiding the underlying implementation method and sparing the programmer from needing to understand anything but `new-type` itself.

7 Related Work

Work with the Same Goal We know of a few pieces of work with the same end goal as us: to take a language with syntactic sugar, and type check it without allowing for the possibility of a user seeing a type error in the core language.

In Lorenzen and Erdweg's *SoundExt* [14], desugaring comes *before* type checking. Their formalism takes (i) a type system for the core language, (ii) a type system for the surface language, and (iii) desugaring rules. It then statically verifies that the surface type system is consistent with the core type system. More precisely, they ensure that, for any program, *if* that program type-checks in the surface language, *then* its desugaring must type-check in the core language. Our approach has a critical advantage over theirs: we do not require type rules to be written for the surface language, but rather infer them. This simplifies the process of extending the language, restoring the adage "oh, that's *just* syntactic sugar". We believe this is especially valuable to authors of, say, domain-specific languages, who are experts in a domain but may not be in the definition of type systems.

Lorenzen and Erdweg's later *SoundX* [15] shows how to *integrate* desugaring and type rules, so that the *same* rule can serve both to extend desugaring and to extend the type system. Essentially, the LHS of a desugaring rule is given as a type rule, and the RHS is given as an expression (per usual). Again, the difference with our work is that we do not require type rules to be written for the surface language.

In a similar vein, both Granz et al.'s MacroML [9] and Mainland's MetaHaskell [16] are staged programming languages. They provide the same guarantee as SoundExt and SoundX: in the words of Mainland, "Well-typed metaprograms should only generate well-typed object terms." Therefore, as with our work, a user is guaranteed never to see a type error in desugared code. Unlike SweetT, these staged systems allow macro definitions to be interspersed with code. On the other

hand, they do not allow macros to check the type of an expression (as in our `calc-type`, section 4.1), or to inspect code (they can only *build* code up from smaller fragments).

Omar et al. provide a syntactic extension mechanism for Wyvern called "type-specific languages" (TSLs) [17]. They note that syntactic extensions often conflict with each other, but can be resolved based on the type that the syntax is checked against. As a simple example, Python uses the same syntax `{...}` for both sets and dictionaries. The expression `{}` is thus ambiguous, but this could be resolved by checking whether the type context expected a set or a dictionary. This is the purpose of TSLs. Like MacroML and MetaHaskell, Wyvern TSLs can only construct code, and cannot inspect or deconstruct it. (This is sufficient for their main intended use case, which is defining language literals.)

Finally, Heeren et al., and later Serrano and Hage, show how to augment a type system with new hand-written error messages [11, 24]. They do so in the context of embedded DSLs that are implemented without syntactic sugar (which is why their work does not immediately apply to our situation). When coding in such an embedded DSL, programmers would normally be confronted with type errors arising from the implementation of the DSL. This line of work allows the DSL author to write custom error messages that instead frame the error in terms of the DSL.

Work with Similar Goals There are many systems that type check *after* desugaring; they potentially show a programmer a type error in code the programmer did not write. Some of these are type systems retrofitted onto languages with macros, such as Type Racket [28] and Typed Clojure [1]. They at best use sourcemaps, providing an accurate line number for a potentially confusing message. There are also metaprogramming systems added to languages with types, such as those of Haskell [25], Ocaml [5], and Scala [2]. They permit grammar extension, and allow desugaring to be defined as an arbitrary function from AST to AST. However, while their metasyntactic types capture the syntactic category of an expression (for instance `Exp` vs. `Name` in Template-Haskell), they do not reflect the object types (e.g., expressions of type `Int` vs. expressions of type `String`). As a result, they need to type check after desugaring. (Contrast this to MetaHaskell and MacroML, described above.)

Fish and Shivers' Ziggurat [7] is a framework for defining a hierarchy of language levels, that makes it easy to attach static analysis of any sort to each level. However, it does not analyze the analysis, so it is possible for one level's analysis to conflict with that of another.

Similar work has been done for *scope rules*. Herman and Wand present λ_m, in which scope annotations on macros are statically checked [12]. λ_m ensures that if a surface term is well-scoped according to the annotations, then after desugaring it will still be well-scoped. Stansifer and Wand continue in this direction with a more powerful system called Romeo [27]. In previous work [23], we go further, *inferring* rather than checking surface scope annotations, directly analogous to this work.

Work with a Different Goal Our work could be contrasted with Chang et al.'s *Turnstile* [4]. Turnstile is a macro-based framework for defining type systems. However, while it uses desugaring in the *implementing* language, it has no support for sugar in the *implemented* language. To this end, it is a competitor to other lightweight language modeling tools (like Redex), and we could have used Turnstile instead of Redex as the basis for our work (we settled on Redex for various practical reasons).

8 Discussion and Conclusion

We have presented an algorithm and system for type resugaring: given syntactic sugar over a typed language, it reconstructs type rules for that sugar. These rules can be added to a type-checker to check the sugar directly (and produce error messages at the level of the sugar, rather than its expanded code), and also be added to the documentation of the surface language. We show that the system can handle a variety of language constructs, and that it successfully suppresses the details of what the sugar expands to.

The paper discusses restrictions on the pattern language of sugars in section 3.1 and the underlying type system in sections 3.2 and 3.3. It also presents some limitations of the implementation in section 6.1. It is worth investigating to see if these restrictions can be lifted to make this idea even more broadly applicable.

In principle, not much in our work has specifically been about *types*. Therefore, this idea could just as well be applied to other syntax-driven deductive systems, such as a natural semantics [13] or structural operational semantics [19]. This would correspondingly enable the creation of semantic rules at the level of the surface language, which can not only enrich a language's documentation but also facilitate its use in, say, a proof assistant.

Acknowledgments

We would like to thank our anonymous reviewers for their very thorough feedback. We appreciate the help of our shepherd, Sukyoung Ryu. This work was partially supported by the US NSF.

Sugars: Letrec and λret

Core Type Rules:

$$\text{t-lambda}\ \frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{e} : \mathtt{U}}{\Gamma \vdash \lambda\ \mathtt{x{:}T.\ e} : (\mathtt{T} \to \mathtt{U})} \qquad \text{t-apply}\ \frac{\Gamma \vdash \mathtt{f} : \mathtt{T} \to \mathtt{U} \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash (\mathtt{f\ e}) : \mathtt{U}} \qquad \text{t-fix}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \to \mathtt{T}}{\Gamma \vdash (\mathtt{fix\ e}) : \mathtt{T}}$$

$$\text{t-raise}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{Str}}{\Gamma \vdash (\mathtt{raise\ e}) : \mathtt{T}} \qquad \text{t-try}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma \vdash \mathtt{e}_{\mathtt{catch}} : \mathtt{Str} \to \mathtt{T}}{\Gamma \vdash \mathtt{try\ e\ with\ e}_{\mathtt{catch}} : \mathtt{T}}$$

Desugaring Rules:

$$\mathtt{letrec\ x : C = } \alpha \mathtt{\ in } \beta \ \Rightarrow\ (\lambda\ \mathtt{x{:}C.}\ \beta)\ (\mathtt{fix}\ (\lambda\ \mathtt{x{:}C.}\ \alpha))$$

$$\lambda\mathtt{ret\ x{:}T.}\ \beta \ \Rightarrow\ \lambda\ \mathtt{x{:}T.\ try\ (let\ return = (}\lambda\ \mathtt{v{:}Str.\ raise\ v)\ in}\ \beta)\ \mathtt{with}\ (\lambda\ \mathtt{v{:}Str.\ v})$$

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

Core Derivations:

$$\text{t-letrec}^{\to}\ \frac{\text{t-lambda}\ \frac{\text{t-premise}\ \frac{\_\ \_\ \_\ \_\ \_\ \_\ \_\ \_}{\Gamma, \mathtt{x} : \mathtt{C} \vdash \beta : \mathtt{D}}}{\Gamma \vdash (\lambda\ \mathtt{x{:}C.}\ \beta) : \mathtt{C} \to \mathtt{D}} \qquad \text{t-fix}\ \frac{\text{t-lambda}\ \frac{\text{t-premise}\ \frac{\_\ \_\ \_\ \_\ \_\ \_\ \_\ \_}{\Gamma, \mathtt{x} : \mathtt{C} \vdash \alpha : \mathtt{C}}}{\Gamma \vdash (\lambda\ \mathtt{x{:}C.}\ \alpha) : \mathtt{C} \to \mathtt{C}}}{\Gamma \vdash (\mathtt{fix}\ (\lambda\ \mathtt{x{:}C.}\ \alpha)) : \mathtt{C}}}{\Gamma \vdash \mathtt{letrec\ x\ :\ C = } \alpha \mathtt{\ in } \beta : \mathtt{D}}$$

$$\text{t-let}\ \frac{\text{t-}\lambda\ \frac{\text{t-raise}\ \frac{\text{t-id}\ \frac{}{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{v} : \mathtt{Str} \vdash \mathtt{v} : \mathtt{Str}}}{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{v} : \mathtt{Str} \vdash \mathtt{raise\ v} : \mathtt{A}}}{\Gamma, \mathtt{x} : \mathtt{T} \vdash (\lambda\ \mathtt{v{:}Str.\ raise\ v}) : \mathtt{Str} \to \mathtt{A}} \qquad \text{t-prem.}\ \frac{\_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_}{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{return} : \mathtt{Str} \to \mathtt{A} \vdash \beta : \mathtt{Str}}}{\text{t-try}\ \frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash (\mathtt{let\ return = (}\lambda\ \mathtt{v{:}Str.\ raise\ v)\ in}\ \beta) : \mathtt{Str} \qquad \text{t-}\lambda\ \frac{\text{t-id}\ \frac{}{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{v} : \mathtt{Str} \vdash \mathtt{v} : \mathtt{Str}}}{\Gamma, \mathtt{x} : \mathtt{T} \vdash (\lambda\ \mathtt{v{:}Str.\ v}) : \mathtt{Str} \to \mathtt{Str}}}{\Gamma, \mathtt{x} : \mathtt{T} \vdash (\mathtt{try\ (let\ return = (}\lambda\ \mathtt{v{:}Str.\ raise\ v)\ in}\ \beta)\ \mathtt{with}\ (\lambda\ \mathtt{v{:}Str.\ v})) : \mathtt{Str}}$$

$$\text{t-}\lambda\text{ret}^{\to}\ \frac{\text{t-}\lambda\ \frac{\Gamma \vdash \lambda\ \mathtt{x{:}T.\ (try\ (let\ return = (}\lambda\ \mathtt{v{:}Str.\ raise\ v)\ in}\ \beta)\ \mathtt{with}\ (\lambda\ \mathtt{v{:}Str.\ v})) : \mathtt{T} \to \mathtt{Str}}{\Gamma \vdash (\lambda\mathtt{ret\ x{:}T.}\ \beta) : \mathtt{T} \to \mathtt{Str}}$$

Resugared Type Rules:

$$\text{t-letrec}\ \frac{\Gamma, \mathtt{x} : \mathtt{C} \vdash \alpha : \mathtt{C} \qquad \Gamma, \mathtt{x} : \mathtt{C} \vdash \beta : \mathtt{D}}{\Gamma \vdash \mathtt{letrec\ x\ :\ C = } \alpha \mathtt{\ in } \beta : \mathtt{D}} \qquad \text{t-}\lambda\text{ret}\ \frac{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{return} : (\mathtt{Str} \to \mathtt{A}) \vdash \beta : \mathtt{Str}}{\Gamma \vdash (\lambda\mathtt{ret\ x{:}T.}\ \beta) : \mathtt{T} \to \mathtt{Str}}$$

Sugar: Upcast

Core Type Rules:

$$\text{t-id}\ \frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{t-lambda}\ \frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{e} : \mathtt{U}}{\Gamma \vdash \lambda\ \mathtt{x{:}T.\ e} : (\mathtt{T} \to \mathtt{U})} \qquad \text{t-apply}\ \frac{\Gamma \vdash \mathtt{f} : \mathtt{T} \to \mathtt{U} \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}}{\Gamma \vdash (\mathtt{f\ e}) : \mathtt{U}}$$

Desugaring Rule:

$$\mathtt{upcast\ } \alpha \mathtt{\ as\ C} \ \Rightarrow\ (\lambda\ \mathtt{x{:}C.\ x})\ \alpha$$

Core Derivation:

$$\text{t-upcast}^{\to}\ \frac{\text{t-lambda}\ \frac{\text{t-id}\ \frac{}{\Gamma, \mathtt{x} : \mathtt{C} \vdash \mathtt{x} : \mathtt{C}}}{\Gamma \vdash (\lambda\ \mathtt{x{:}C.\ x}) : \mathtt{C} \to \mathtt{C}} \qquad \text{t-premise}\ \frac{\_\ \_\ \_\ \_\ \_}{\Gamma \vdash \alpha : \mathtt{A}} \qquad \text{t-sub-premise}\ \frac{\_\ \_\ \_\ \_}{\mathtt{A} <: \mathtt{C}}}{\Gamma \vdash ((\lambda\ \mathtt{x{:}C.\ x})\ \alpha) : \mathtt{C}}}{\Gamma \vdash \mathtt{upcast\ } \alpha \mathtt{\ as\ C} : \mathtt{C}}$$

Resugared Type Rule:

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

Figure 6. Derivation examples

References

[1] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-49498-1_4

[2] Eugene Burmako. 2013. Scala macros: let our powers combine!. In *Scala Workshop*. ACM, New York, NY, USA. https://doi.org/10.1145/2489837.2489840

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

[4] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *Principles of Programming Languages*. ACM, New York, NY, USA. https://doi.org/10.1145/3093333.3009886

[5] Daniel de Rauglaudre. 2007. Camlp5 - Reference Manua. (2007). http://pauillac.inria.fr/ ddr/camlp5/doc/pdf/camlp5-5.06.pdf.

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

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

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

[9] Steve Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *International Conference on Functional Programming*. ACM, New York, NY, USA. https://doi.org/10.1145/507546.507646

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *European Conference on Object-oriented Programming*. Springer-Verlag, Berlin, Heidelberg, 25. https://doi.org/citation.cfm?id=1883978.1883988

[11] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the Type Inference Process. In *International Conference on Functional Programming*. ACM, New York, NY, USA. https://doi.org/citation.cfm?doid=944746.944707

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

[13] Gilles Kahn. 1987. Natural Semantics. In *International Symposium on Theoretical Aspects of Computer Software*.

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

[15] Florian Lorenzen and Sebastian Erdweg. 2016. Sound type-dependent syntactic language extension. In *Principles of Programming Languages*. ACM, New York, NY, USA. https://doi.org/10.1145/2837614.2837644

[16] Geoffrey Mainland. 2012. Explicitly Heterogeneous Metaprogramming with MetaHaskell. In *International Conference on Functional Programming*. ACM, New York, NY, USA. https://doi.org/10.1145/2398856.2364572

[17] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *European Conference on Object-Oriented Programming*. Springer-Verlag, New York, NY, USA. https://doi.org/10.1007/978-3-662-44202-9_5

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

[19] Gordon D. Plotkin. 1981. *A Structured Approach to Operational Semantics*. Technical Report DAIMI FN-19. Computer Science Department, Aarhus University, Aarhus, Denmark.

[20] Joe Gibbs Politz, Matt Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Dynamic Languages Symposium*.

[21] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty: A Tested Semantics for the Python Programming Language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York, NY, USA, 217–232. https://doi.org/10.1145/2544173.2509536

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

[23] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring Scope through Syntactic Sugar. In *International Conference on Functional Programming*. ACM, New York, NY, USA. https://doi.org/10.1145/3110288

[24] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *European Symposium on Programming Languages and Systems*. ACM, New York, NY, USA. https://doi.org/citation.cfm?id=2958906

[25] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop*. ACM, New York, NY, USA. https://doi.org/10.1145/636517.636528

[26] Joel Spolsky. 2002. The Law of Leaky Abstractions. Blog post: Joel on Software. (2002). https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

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

[28] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *Programming Languages Design and Implementation*. ACM, New York, NY, USA. https://doi.org/10.1145/1993316.1993514