Frameworks for Reasoning about Syntax that Utilize Quotation and Evaluation[*]

William M. Farmer and Pouya Larjani[†]

24 June 2014

Abstract

It is often useful, if not necessary, to reason about the syntactic structure of an expression in an interpreted language (i.e., a language with a semantics). This paper introduces a mathematical structure called a *syntax framework* that is intended to be an abstract model of a system for reasoning about the syntax of an interpreted language. Like many concrete systems for reasoning about syntax, a syntax framework contains a mapping of expressions in the interpreted language to syntactic values that represent the syntactic structures of the expressions; a language for reasoning about the syntactic values; a mechanism called *quotation* to refer to the syntactic value of an expression; and a mechanism called *evaluation* to refer to the value of the expression represented by a syntactic value. A syntax framework provides a basis for integrating reasoning about the syntax of the expressions with reasoning about what the expressions mean. The notion of a syntax framework is used to discuss how quotation and evaluation can be built into a language and to define what quasiquotation is. Several examples of syntax frameworks are presented.

[*]This research was supported by NSERC.

[†]Address: Department of Computing and Software, McMaster University 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada. E-mail: `wmfarmer@mcmaster.ca`, `pouya.larjani@gmail.com`.

1 Introduction

Every calculus student knows that computing the derivative of a function directly from the definition is an excruciating task, while computing the derivative using the rules of differentiation is a pleasure. A differentiation rule is a function, but not a usual function like the square root function or the limit of a sequence operator. Instead of mapping a function to its derivative, it maps one syntactic representation of a function to another. For example, the *product rule* maps an expression of the form

$$\frac{d}{dx}(u \cdot v),$$

where u and v are expressions that may include occurrences of x, to the expression

$$\frac{d}{dx}(u) \cdot v + u \cdot \frac{d}{dx}(v).$$

We call a mapping, like a differentiation rule, that takes one syntactic expression to another syntactic expression a *transformer* [13]. A full formalization of calculus requires a reasoning system in which (1) the derivative of a function can be defined, (2) the differentiation rules can be represented as transformers, and (3) the transformers representing the differentiation rules can be shown to compute derivatives. Such a reasoning system must provide the means to reason about the syntactic manipulation of expressions as well as the connection these manipulations have to the semantics of the expressions. In other words, the reasoning system must allow one to reason about syntax and its relationship to semantics. See [12] for a detailed discussion about the formalization of symbolic differentiation and other syntax-based mathematical algorithms.

An *interpreted language* is a language L such that each expression e in L is mapped to a *semantic value* that serves as the meaning of e. What facilities does a reasoning system need for reasoning about the interplay of the syntax and semantics of an interpreted language L? Here are four candidates:

1. A set of *syntactic values* that represent the syntactic structures of the expressions in L.

2. A language for expressing statements about syntactic values and thereby indirectly about the syntactic structures of the expressions in L.

3. A mechanism called *quotation* for referring to the syntactic value that represents a given expression in L.

4. A mechanism called *evaluation* for referring to the semantic value of the expression whose syntactic structure is represented by a given syntactic value.

Quotation and evaluation together provide the means to integrate reasoning about the syntax of the expressions with reasoning about what the expressions mean.

This paper has three objectives. The first objective is to introduce a mathematical structure called a *syntax framework* that is intended to be an abstract model of a system for reasoning about the syntax of an interpreted language. A syntax framework for an interpreted language L contains four components corresponding to the four facilities mentioned just above:

1. A function called a *syntax representation* that maps each expression e in L to a *syntactic value* that represents the syntactic structure of e.

2. A language called a *syntax language* whose expressions denote syntactic values.

3. A *quotation* function that maps an expression e in L to an expression in the syntax language that denotes the syntactic value of e.

4. An *evaluation* function that maps an expression e in the syntax language to an expression in L whose semantic value is the same as that of the expression in L whose syntactic value is denoted by e.

The second objective is to demonstrate that a syntax framework has the ingredients needed for reasoning effectively about syntax. We discuss the benefits of a syntax framework for reasoning about syntax and particularly for reasoning about transformers like the differentiation rules. We explain how the liar paradox can be avoided when quotation and evaluation are built-in operators. And we define in a syntax framework a notion of quasiquotation which greatly facilitates constructing expressions that denote syntactic values.

The third objective is to show that the notion of a syntax framework embodies a common structure that is found in a variety of systems for reasoning about the interplay of syntax and semantics. In particular, we show that the standard systems in which syntactic structure is represented by strings, Gödel numbers, and members of an inductive type are instances of a syntax framework. We also show that several more sophisticated systems from the literature, including a simplified version of Lisp, can be viewed as syntax frameworks.

Reflection is a technique to embed reasoning about a reasoning system (i.e., metareasoning) in the reasoning system itself. Reflection has been employed in logic [22], theorem proving [21], and programming [8]. Since metareasoning very often involves the syntactic manipulation of expressions, a syntax framework is a natural subcomponent of a reflection mechanism.

The rest of the paper is organized as follow. The next section, section 2, defines the notion of a syntax framework and discusses it benefits. Section 3 presents three standard syntax reasoning systems that are instances of a syntax framework. Section 4 discusses built-in operators for quotation and evaluation as found in Lisp and other languages and explains how the liar paradox is avoided in a syntax framework. Section 5 defines a notion of quasiquotation in a syntax framework. Section 6 identifies some sophisticated syntax reasoning systems in the literature that are instances of a syntax framework. The paper ends with a conclusion in section 7.

2 Syntax Frameworks

In this section we will define a mathematical structure called a *syntax framework*. In the subsequent sections we will give several examples of syntax reasoning systems that can be interpreted as instances of this structure.

The reader should note that the notion of a syntax framework presented here is not adequate to interpret syntax reasoning systems, such as programming languages, that contain context-sensitive expressions (such as mutable variables). To interpret these kinds of systems, a syntax framework must be extended to a *contextual syntax framework* that includes mutable contexts. For further discussion, see Remark 2.7.4.

2.1 Interpreted Languages

Let a *formal language* be a set of expressions each having a unique mathematically precise syntactic structure. We will leave "expression" and "mathematically precise syntactic structure" unspecified. A formal language L is a *sublanguage* of a formal language L' if $L \subseteq L'$.

An interpreted language is a formal language with a semantics:

Definition 2.1.1 (Interpreted Language) An *interpreted language* is a triple $I = (L, D_{\mathrm{sem}}, V_{\mathrm{sem}})$ where:

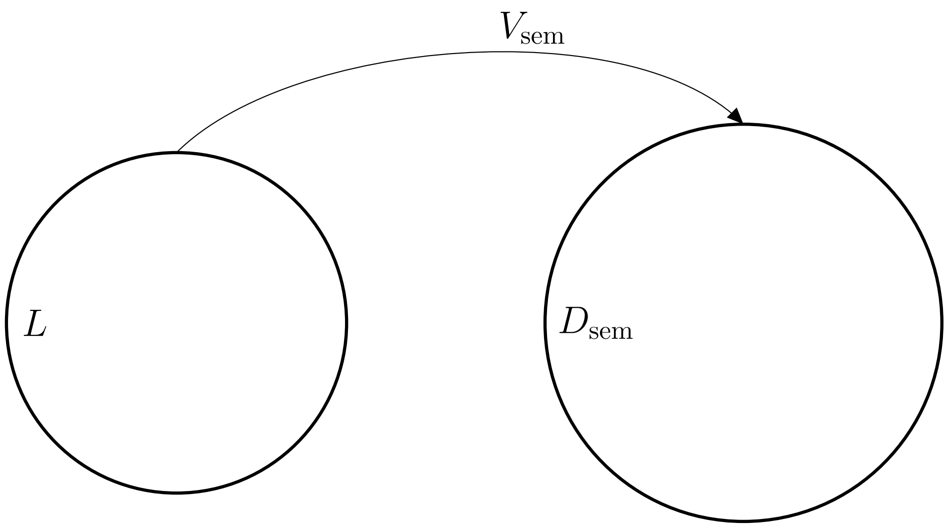

Figure 1: An Interpreted Language

1. L is a formal language.
2. D_{sem} is a nonempty domain (set) of *semantic values*.
3. $V_{\text{sem}} : L \to D_{\text{sem}}$ is a total function, called a *semantic valuation function*, that assigns each expression $e \in L$ a semantic value $V_{\text{sem}}(e) \in D_{\text{sem}}$. □

An interpreted language is thus a formal language with an associated assignment of a semantic meaning to each expression in the language. Each expression of an interpreted language thus has both a syntactic structure and a semantic meaning. There is no restriction placed on what can be a semantic value. An interpreted language is graphically depicted in Figure 1 (we will add elements to this figure as the discussion advances).

Example 2.1.2 (Many-Sorted First-Order Languages) Let L be the set of the terms and formulas of a many-sorted first-order language with sorts $\alpha_1, \ldots, \alpha_n$. Define L_i to be the set of terms of sort α_i for each i with $1 \leq i \leq n$ and L_{f} to be the set of formulas of the many-sorted first-order language.

Let $(D_1, \ldots, D_n, I)$ be a model for the many-sorted first-order language L where each D_i is a nonempty domain and I is an interpretation function for the individual constants, function symbols, and predicate symbols of L.

Let φ_i be a mapping from the variables in L_i to D_i for each i with $1 \leq i \leq n$. The model $(D_1, \ldots, D_n, I)$ and variable assignments $\varphi_1, \ldots, \varphi_n$ determine a semantic valuation function $V_i : L_i \rightarrow D_i$ on terms of sort α_i for each i with $1 \leq i \leq n$ and a semantic valuation function $V_{\mathrm{f}} : L_{\mathrm{f}} \rightarrow \{\mathrm{T}, \mathrm{F}\}$ on formulas. Then

$$(L, D_1 \cup \cdots \cup D_n \cup \{\mathrm{T}, \mathrm{F}\}, V_1 \cup \cdots \cup V_n \cup V_{\mathrm{f}})$$

is an interpreted language. □

2.2 Syntax Representations and Syntax Languages

A syntax representation of a formal language is an assignment of syntactic values to the expressions of the language:

Definition 2.2.1 (Syntax Representation) Let L be a formal language. A *syntax representation* of L is a pair $R = (D_{\mathrm{syn}}, V_{\mathrm{syn}})$ where:

1. D_{syn} is a nonempty domain (set) of *syntactic values*. Each member of D_{syn} represents a syntactic structure.

2. $V_{\mathrm{syn}} : L \rightarrow D_{\mathrm{syn}}$ is an injective, total function, called a *syntactic valuation function*, that assigns each expression $e \in L$ a syntactic value $V_{\mathrm{syn}}(e) \in D_{\mathrm{syn}}$ such that $V_{\mathrm{syn}}(e)$ represents the syntactic structure of e. □

A syntax representation of a formal language is thus an assignment of a syntactic meaning to each expression in the language. Notice that, if $R = (D_{\mathrm{syn}}, V_{\mathrm{syn}})$ is a syntax representation of L, then $(L, D_{\mathrm{syn}}, V_{\mathrm{syn}})$ is an interpreted language.

Example 2.2.2 (Expressions as Strings: Syntax Representation) Let L be a many-sorted first-order language. The expressions of L — i.e., the terms and formulas of L — can be viewed as certain strings of symbols. For example, the term $f(x)$ can be viewed as the string `"f(x)"` composed of four symbols. Let $\mathcal{A}$ be the alphabet of symbols occurring in the expressions of L and $\mathsf{strings}_{\mathcal{A}}$ be the set of strings over $\mathcal{A}$. Then the syntactic structure of an expression can be represented by a string in $\mathsf{strings}_{\mathcal{A}}$, and we can define a function $S : L \rightarrow \mathsf{strings}_{\mathcal{A}}$ that maps each expression of L to the string over $\mathcal{A}$ that represents its syntactic structure. S is an injective, total function since, for each $e \in L$, there is exactly one string in $\mathsf{strings}_{\mathcal{A}}$ that represents the syntactic structure of e. Therefore, $(\mathsf{strings}_{\mathcal{A}}, S)$ is a syntax representation of L. □

A syntax language for a syntax representation is a language of expressions that denote syntactic values in the syntax representation:

Definition 2.2.3 (Syntax Language) Let $R = (D_{\text{syn}}, V_{\text{syn}})$ be a syntax representation of a formal language L_{obj}. A *syntax language* for R is a pair (L_{syn}, I) where:

1. $I = (L, D_{\text{sem}}, V_{\text{sem}})$ in an interpreted language.
2. $L_{\text{obj}} \subseteq L$, $L_{\text{syn}} \subseteq L$, and $D_{\text{syn}} \subseteq D_{\text{sem}}$.
3. V_{sem} restricted to L_{syn} is a total function $V'_{\text{sem}} : L_{\text{syn}} \rightarrow D_{\text{syn}}$. □

Notice that, if (L_{syn}, I) is a syntax language for R (as in the definition above), then $(L_{\text{syn}}, D_{\text{syn}}, V'_{\text{sem}})$ is an interpreted language.

Example 2.2.4 (Expressions as Strings: Syntax Language) Let $I = (L, D, V)$ where

$$D = D_1 \cup \cdots \cup D_n \cup \{\text{T}, \text{F}\}$$

and

$$V = V_1 \cup \cdots \cup V_n \cup V_{\text{f}}$$

be the interpreted language given in Example 2.1.2. Recall that L is the set of terms and formulas of a many-sorted first-order language with sorts $\alpha_1, \ldots, \alpha_n$. Suppose $\alpha_1 = \mathsf{Symbol}$, $\alpha_2 = \mathsf{String}$, D_1 is the alphabet of L, and D_2 is the set of strings over D_1. Let $S : L \rightarrow D_2$ be the total function that maps each $e \in L$ to the string in D_2 that represents the syntactic structure of e. Then $R = (D_2, S)$ is a syntax representation of L as in Example 2.2.2 and (L_2, I) is a syntax language for R since $L_2 \subseteq L$, $D_2 \subseteq D$, and V restricted to L_2 is $V_2 : L_2 \rightarrow D_2$. □

2.3 Definition of a Syntax Framework

A syntax framework is a structure that is built from an interpreted language $I = (L, D_{\text{sem}}, V_{\text{sem}})$ in three stages.

The first stage is to choose an object language $L_{\text{obj}} \subseteq L$ and a syntax representation $R = (D_{\text{syn}}, V_{\text{syn}})$ for L_{obj} such that $D_{\text{syn}} \subseteq D_{\text{sem}}$. ($L_{\text{obj}}$ could be the entire language L as in Example 2.2.2.) This first stage is depicted in Figure 2.

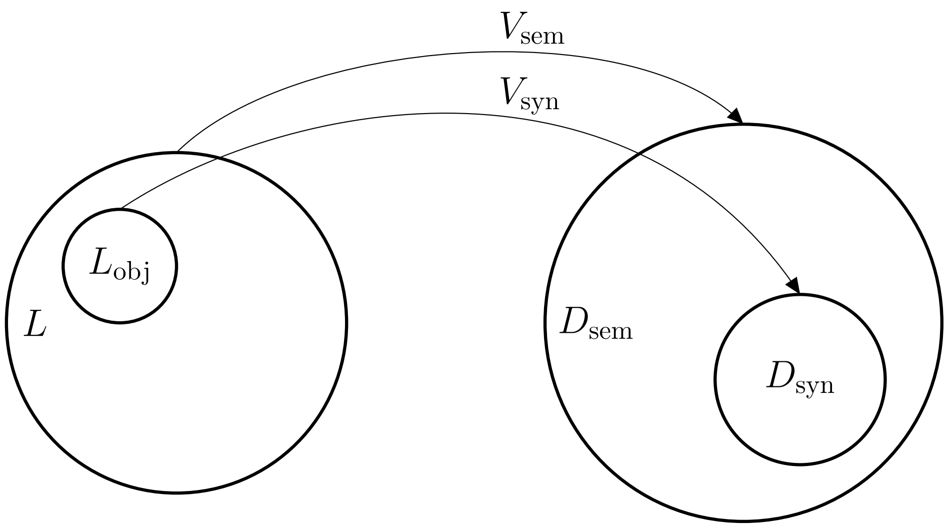

Figure 2: Stage 1 of a Syntax Framework

The second stage is to choose a language $L_{\text{syn}} \subseteq L$ such that (L_{syn}, I) is a syntax language for R. This second stage, depicted in Figure 3, establishes L_{syn} as a language that can be used to make statements in L about the syntax of the object language L_{obj} via the syntax representation established in the stage 1. (V'_{sem} is V_{sem} restricted to L_{syn}.)

The third and final stage is to link L_{obj} and L_{syn} using mappings $Q : L_{\text{obj}} \rightarrow L_{\text{syn}}$ and $E : L_{\text{syn}} \rightarrow L_{\text{obj}}$ as depicted in Figure 4. Q is an injective, total function such that, for all $e \in L_{\text{obj}}$,

$$V_{\text{sem}}(Q(e)) = V_{\text{syn}}(e).$$

For $e \in L_{\text{obj}}$, $Q(e)$ is called the *quotation* of e. $Q(e)$ denotes a value in D_{syn} that represents the syntactic structure of e. E is a (possibly partial) function such that, for all $e \in L_{\text{syn}}$,

$$V_{\text{sem}}(E(e)) = V_{\text{sem}}(V_{\text{syn}}^{-1}(V_{\text{sem}}(e)))$$

whenever $E(e)$ is defined. For $e \in L_{\text{syn}}$, $E(e)$ is called the *evaluation* of e. If it is defined, $E(e)$ denotes the same value in D_{sem} that the expression represented by the value of e denotes. Notice that the equation above implies $E(e)$ is undefined if $V_{\text{sem}}(e)$ is not in the image of L_{obj} under V_{syn}. Since there will usually be different $e_1, e_2 \in L_{\text{syn}}$ that denote the same syntactic value, E will usually not be injective.

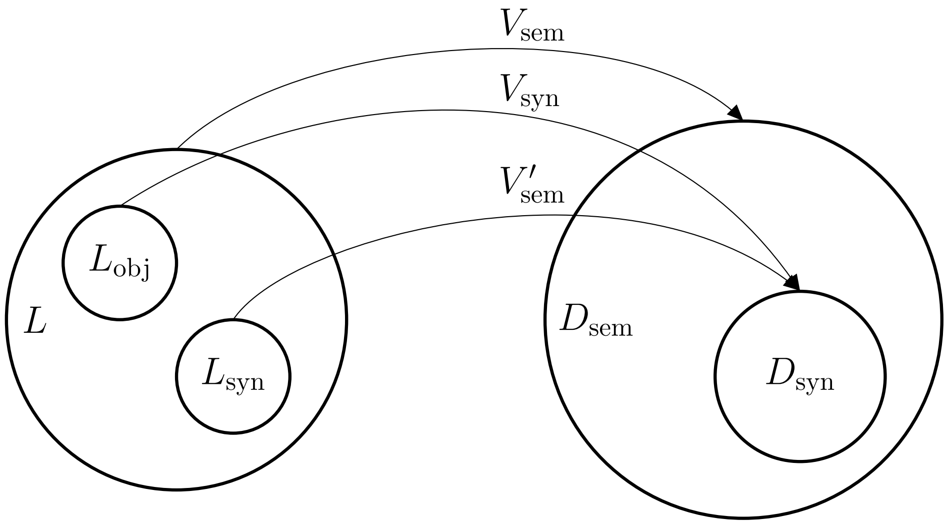

Figure 3: Stage 2 of a Syntax Framework

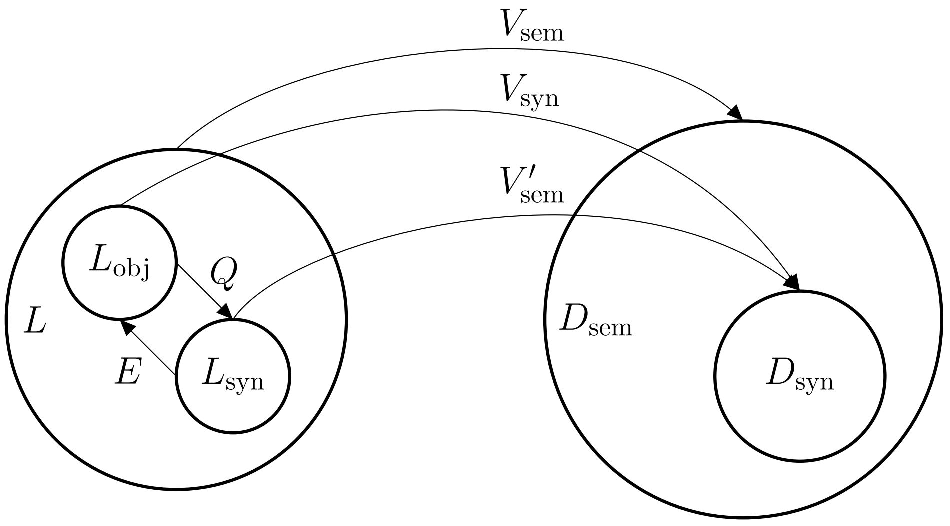

Figure 4: A Syntax Framework

The full definition of a syntax framework is obtained when we put these three stages together:

Definition 2.3.1 (Syntax Framework in an Interpreted Language) Let $I = (L, D_{\rm sem}, V_{\rm sem})$ be an interpreted language and $L_{\rm obj}$ be a sublanguage of L. A *syntax framework* for $(L_{\rm obj}, I)$ is a tuple $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ where:

1. $R = (D_{\rm syn}, V_{\rm syn})$ is a syntax representation of $L_{\rm obj}$.

2. $(L_{\rm syn}, I)$ is syntax language for R.

3. $Q : L_{\rm obj} \rightarrow L_{\rm syn}$ is an injective, total function, called a *quotation function*, such that:

   Quotation Axiom. For all $e \in L_{\rm obj}$,

   $$V_{\rm sem}(Q(e)) = V_{\rm syn}(e).$$

4. $E : L_{\rm syn} \rightarrow L_{\rm obj}$ is a (possibly partial) function, called an *evaluation function*, such that:

   Evaluation Axiom. For all $e \in L_{\rm syn}$,

   $$V_{\rm sem}(E(e)) = V_{\rm sem}(V_{\rm syn}^{-1}(V_{\rm sem}(e)))$$

   whenever $E(e)$ is defined. $\Box$

L is called the *full language* of the F. When $D_{\rm sem}$ and $V_{\rm sem}$ are understood, we will say that F is a syntax framework for $L_{\rm obj}$ in L. Notice that a syntax framework contains three interpreted languages: $(L, D_{\rm sem}, V_{\rm sem})$, $(L_{\rm obj}, D_{\rm syn}, V_{\rm syn})$, and $(L_{\rm syn}, D_{\rm syn}, V'_{\rm sem})$. Notice also that the functions Q and E are part of the metalanguage of L and the expressions of the form $Q(e)$ and $E(e)$ are not necessarily expressions of L. In section 4 we will discuss syntax frameworks in which quotations and evaluations are expressions in L itself.

2.4 Two Basic Lemmas

Let $I = (L, D_{\rm sem}, V_{\rm sem})$ be an interpreted language, $L_{\rm obj}$ be a sublanguage of L, and $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ be a syntax framework for $(L_{\rm obj}, I)$.

Lemma 2.4.1 (Law of Disquotation) *For all $e \in L_{\rm obj}$,*

$$V_{\rm sem}(E(Q(e))) = V_{\rm sem}(e)$$

whenever $E(Q(e))$ is defined.

Proof Let $e \in L_{\rm obj}$ such that $E(Q(e))$ is defined. Then

$$\begin{aligned} V_{\rm sem}(E(Q(e))) &= V_{\rm sem}(V_{\rm syn}^{-1}(V_{\rm sem}(Q(e)))) & (1)\\ &= V_{\rm sem}(V_{\rm syn}^{-1}(V_{\rm syn}(e))) & (2)\\ &= V_{\rm sem}(e) & (3)\end{aligned}$$

(1) follows from the Evaluation Axiom since $E(Q(e))$ is defined. (2) follows from the Quotation Axiom. And (3) is by the fact that $V_{\rm syn}(e)$ is total on $L_{\rm obj}$. □

The Law of Disquotation does not hold universally in general because E may not be total on quotations.

Definition 2.4.2 (Direct Evaluation) Let $E^* : L_{\rm syn} \rightarrow L_{\rm obj}$ to be the (possibly partial) function such that, for all $e \in L_{\rm syn}$, $E^*(e) = V_{\rm syn}^{-1}(V_{\rm sem}(e))$ whenever $V_{\rm syn}^{-1}(V_{\rm sem}(e))$ is defined. E^* is called the *direct evaluation function for F*. □

Lemma 2.4.3 (Direct Evaluation)

1. *E^* satisfies the Evaluation Axiom.*
2. *For all $e \in L_{\rm syn}$, if $E^*(e)$ and $E(e)$ are defined, then*

$$V_{\rm sem}(E^*(e)) = V_{\rm sem}(E(e)).$$

3. *If $V_{\rm syn}$ is surjective, then E^* is total.*

Proof

Part 1 Follows immediate from the definition of E^*.

Part 2 Let $e \in L_{\rm syn}$ such that $E^*(e)$ and $E(e)$ are defined. By the Evaluation Axiom, $V_{\rm sem}(E(e)) = V_{\rm sem}(V_{\rm syn}^{-1}(V_{\rm sem}(e))) = V_{\rm sem}(E^*(e))$.

Part 3 Let $V_{\rm syn}$ be surjective and $e \in L_{\rm syn}$. Then $V_{\rm syn}^{-1}(V_{\rm sem}(e))$ is defined and hence E^* is total by its definition. □

Thus the direct evaluation function is a special evaluation function that is defined for every syntax framework and is total if the syntactic valuation function is surjective.

2.5 Syntax Frameworks in an Interpreted Theory

The notion of a syntax framework can be easily lifted from an interpreted language to an interpreted theory. Let a *theory* be a pair $T = (L, \Gamma)$ where L is a language and Γ is a set of sentences in L (that serve as the axioms of theory). A *model* of T is a pair $M = (D^M_{\rm sem}, V^M_{\rm sem})$ such that $D^M_{\rm sem}$ is a set of values that includes the truth values $\textsc{t}$ (true) and $\textsc{f}$ (false) and $V^M_{\rm sem} : L \rightarrow D^M_{\rm sem}$ is a total function such that, for all sentences $A \in \Gamma$, $V^M_{\rm sem}(A) = \textsc{t}$. An *interpreted theory* is then a pair $I = (T, \mathcal{M})$ where T is a theory and $\mathcal{M}$ is a set of models of T.

A syntax framework in an interpreted theory is a syntax framework with respect to each model of the interpreted theory:

Definition 2.5.1 (Syntax Framework in an Interpreted Theory) Let $I = (T, \mathcal{M})$ be an interpreted theory where $T = (L, \Gamma)$ and $L_{\rm obj}$ be a sublanguage of L. A *syntax framework* for $(L_{\rm obj}, I)$ is a triple $F = (L_{\rm syn}, Q, E)$ where:

1. $L_{\rm syn} \subseteq L$.

2. $Q : L_{\rm obj} \rightarrow L_{\rm syn}$ is an injective, total function.

3. $E : L_{\rm syn} \rightarrow L_{\rm obj}$ is a (possibly partial) function.

4. For all $M = (D^M_{\rm sem}, V^M_{\rm sem}) \in \mathcal{M}$, $F^M = (D^M_{\rm syn}, V^M_{\rm syn}, L_{\rm syn}, Q, E)$ is a syntax framework for $(L_{\rm obj}, (L, D^M_{\rm sem}, V^M_{\rm sem}))$ where $D^M_{\rm syn}$ is the range of $V^M_{\rm sem}$ restricted to $L_{\rm syn}$ and $V^M_{\rm syn} = V^M_{\rm sem} \circ Q$. $\Box$

2.6 Benefits of a Syntax Framework

The purpose of a syntax framework is to provide the means to reason about the syntax of a designated object language. We will briefly examine the specific benefits that a syntax framework offers for this purpose.

Let $I = (L, D_{\rm sem}, V_{\rm sem})$ be an interpreted language, $L_{\rm obj}$ be a sublanguage of L, and $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ be a syntax framework for $(L_{\rm obj}, I)$.

The first, and most important, benefit of F is that it provides a language, $L_{\rm syn}$, for expressing statements in L about the syntactical structure of expressions in $L_{\rm obj}$. These statements refer to the syntax of $L_{\rm obj}$ via the syntax representation of F. For example, if A is a formula in $L_{\rm obj}$, e_A is an expression in $L_{\rm syn}$ that denotes the representation of A, and L is sufficiently expressive, we could express in L a statement of the form $\mathsf{is\text{-}implication}(e_A)$ that *indirectly* says "A is an implication".

Having quotation in F enables statements about the syntax of L_{obj} to be expressed directly in the metalanguage of L. For example, $\mathsf{is\text{-}implication}(Q(A))$ would *directly* say "A is an implication". Quotation also allows us to construct new expressions from deconstructed components of old expressions. For example, if $A \Rightarrow B$ is a formula in L_{obj} and L is sufficiently expressive,

$$\mathsf{build\text{-}implication}(\mathsf{succedent}(Q(A \Rightarrow B)), \mathsf{antecedent}(Q(A \Rightarrow B)))$$

would denote the representation of $B \Rightarrow A$.

Having evaluation in F enables statements about the semantics of the expressions represented by members of D_{syn} to be expressed directly in the metalanguage of L. For example, if c is the expression given in the previous paragraph, then $E(c)$ would be a formula in L_{obj} that asserts $B \Rightarrow A$.

By virtue of these basic benefits, a syntax framework is well equipped to define and specify transformers. As we have mentioned in the introduction, a *transformer* maps expressions to expressions. More precisely, an *n-ary transformer over a language L* maps expressions $e_1, \ldots, e_n$ in L to an expression e in L (where $n \geq 0$). A transformer can be defined by either an algorithm (e.g., a program in a programming language) or a function (e.g., an expression in a logic that denotes a function). Transformers include symbolic computation rules (like the product rule mentioned in the Introduction), rules of inference, rewrite rules, expression simplifiers, substitution operations, decision procedures, etc.

A transformer over a language L is usually defined only in the metalanguage of L and is not defined by an expression in L itself. For example, the rules of inference for first-order logic are not expressions in first-order logic. A syntax framework with a sufficiently expressive language can be used to transfer a transformer over L from the metalanguage of L to L itself. To see this, let $T : L_{\mathrm{obj}} \times \cdots \times L_{\mathrm{obj}} \rightarrow L_{\mathrm{obj}}$ be an n-ary transformer over L_{obj} defined in the metalanguage of L. If L is sufficiently expressive, it would be possible to define an operator $e_T : L_{\mathrm{syn}} \times \cdots \times L_{\mathrm{syn}} \rightarrow L_{\mathrm{syn}}$ in L that denotes a function $f_T : D_{\mathrm{syn}} \times \cdots \times D_{\mathrm{syn}} \rightarrow D_{\mathrm{syn}}$ that represents T. Using quotation, e_T is specified by the following statement in the metalanguage of L:

$$\forall\, e_1, \ldots, e_n : L_{\mathrm{obj}} \,.\, e_T(Q(e_1), \ldots, Q(e_n)) = Q(T(e_1, \ldots, e_n)).$$

The full power of a syntax framework is exhibited in a specification of the semantic meaning of a transformer. Suppose L is a language of natural number arithmetic, the expressions in L_{obj} denote natural numbers, L_{obj} contains a sublanguage L_{nat} of terms denoting natural numbers, and L_{syn} contains a sublanguage L_{num} of terms denoting natural number numerals $Q(0), Q(1), Q(2), \ldots$. Further suppose that add is a binary transformer over L_{nat} that "adds" two natural number terms so that, e.g., $\mathsf{add}(2,3) = 5$. Then, using evaluation, the semantic meaning of e_{add}, the representation of add in L, is specified by the following statement in the metalanguage of L:

$$\forall\, e_1, e_2 : L_{\text{num}} \,.\, E(e_{\mathsf{add}}(e_1, e_2)) = E(e_1) + E(e_2)$$

where $+ : L_{\text{nat}} \times L_{\text{nat}} \rightarrow L_{\text{nat}}$ is a binary operator in L that denotes the sum function.

See [12] for further discussion on how transformers can be formalized using a syntax framework.

2.7 Further Remarks

Remark 2.7.1 (Syntax Representation) Although a syntax representation is a crucial component of a syntax framework, very little restriction is placed on what a syntax representation can be. Almost any representation that captures the syntactic structure of the expressions in the object language is acceptable. In fact, it is not necessary to capture the entire syntactic structure of an expression, only the part of the syntactic structure that is of interest to the developer of the syntax framework. $\square$

Remark 2.7.2 (Theories of Quotation) The quotation function Q of a syntax framework is based on the *disquotational theory of quotation* [3]. According to this theory, a quotation of an expression e is an expression that denotes e itself. In our definition of a syntax framework, $Q(e)$ denotes a value that represents e (as a syntactic entity). Andrew Polonsky presents in [33] a set of axioms for quotation operators of this kind. There are several other theories of quotation that have been proposed [3]. $\square$

Remark 2.7.3 (Theories of Truth) When e is a representation of a truth-valued expression e', the evaluation $E(e)$ is a formula that asserts the truth of e'. Thus the evaluation function E of a syntax framework is a *truth predicate* [16]. A truth predicate is the face of a *theory of truth*: the properties of a truth predicate characterize a theory of truth [23]. The definition of a syntax framework imposes no restriction on E as a truth predicate other than that the Evaluation Axiom must hold. What truth is and how it can be formalized is a fundamental research area of logic, and avoiding inconsistencies derived from the liar paradox (which we address below) and similar statements is one of the major research issues in the area (see [20]). □

Remark 2.7.4 (Contextual Syntax Frameworks) We have mentioned already that a syntax framework cannot interpret syntax reasoning systems that contain context-sensitive expressions. This means that a syntax framework is not suitable for programming languages with mutable variables. For programming languages, a syntax framework needs to be generalized to a *contextual syntax framework* that includes a semantic valuation function that takes a *valuation context* as part of its input and returns a modified valuation context as part of its output. *Metaprogramming* is the writing of programs that manipulate other programs. It requires a means to manipulate the syntax of the programs in a programming language. In other words, metaprogramming requires code to be data. Examples of metaprogramming languages include Lisp, Agda [29, 30], F# [25], MetaML [37], MetaOCaml [35], reFLect [19], and Template Haskell [36]. An appropriate contextual syntax framework would provide a good basis for discussing the code manipulation done in metaprogramming. We will present the notion of a contextual syntax framework in a future paper. □

3 Three Standard Examples

We will now present three standard syntax reasoning systems that are examples of a syntax framework.

3.1 Example: Expressions as Strings

We will continue the development of Example 2.2.4. Suppose L contains the following operators:

- An individual constant c_a of sort Symbol for each $a \in \mathcal{A}$.
- An individual constant nil of sort String.
- A function symbol cons of sort $\mathsf{Symbol} \times \mathsf{String} \rightarrow \mathsf{String}$.
- A function symbol head of sort $\mathsf{String} \rightarrow \mathsf{Symbol}$.
- A function symbol tail of sort $\mathsf{String} \rightarrow \mathsf{String}$.

The terms of sort String are intended to denote strings over $\mathcal{A}$ in the usual way. cons is used to describe the construction of strings, while head and tail are used to describe the deconstruction of strings. The terms of sort String can thus be used as a language to reason *directly* about strings over $\mathcal{A}$ and *indirectly* about the syntactic structure of the expressions of L (including the terms of sort String themselves).

This reasoning system for the syntax of L can be strengthened by interconnecting the expressions of L and the terms of sort String. This is done by defining a quotation function Q and an evaluation function E.

$Q : L \to L_2$ maps each expression e of L to a term $Q(e)$ of sort String such that $Q(e)$ denotes $S(e)$, the string over $\mathcal{A}$ that represents e. For example, Q could map $f(x)$ to

$$\mathsf{cons}(c_{\mathtt{f}}, \mathsf{cons}(c_{(}, \mathsf{cons}(c_{\mathtt{x}}, \mathsf{cons}(c_{)}, \mathsf{nil})))),$$

which denotes the string `"f(x)"`. Thus Q provides the means to refer to a representation of the syntactic structure of an expression of L.

$E : L_2 \to L$ maps each term t of sort String to the expression $E(t)$ of L such that the syntactic structure of $E(t)$ is represented by the string denoted by t provided t denotes a string that actually represents the syntactic structure of some expression of L. For example, E maps the term displayed above (i.e., $Q(f(x))$) to $f(x)$. Thus E provides the means to refer to the value of the expression whose syntactic structure is represented by the string that a term of sort String denotes. E is a partial function on the terms of sort String since not every string in D_2 represents the syntactic structure of some expression in L and V_2 is surjective. Notice that, for all expressions e of L, $E(Q(e)) = e$ — that is, the *law of disquotation* holds universally.

We showed previously that $I = (L, D, V)$ is an interpreted language, $R = (D_2, S)$ is a syntax representation of L, and (L_2, I) is a syntax language for R. Q is injective since the syntactic structure of each expression in L is represented by a unique string in D_2. For $e \in L$,

$$V(Q(e)) = V_2(Q(e)) = S(e),$$

and thus Q satisfies the Quotation Axiom if $L_{\rm obj} = L$, $D_{\rm syn} = D_2$, $V_{\rm syn} = S$, and $L_{\rm syn} = L_2$. For $t \in L_2$ such that $E(t)$ is defined,

$$V(E(t)) = V(V_{\rm syn}^{-1}(V_2(t))) = V(V_{\rm syn}^{-1}(V(t))),$$

and thus E satisfies the Evaluation Axiom if $L_{\rm obj} = L$, $D_{\rm syn} = D_2$, $V_{\rm syn} = S$, and $L_{\rm syn} = L_2$.

Therefore,

$$F = (D_2, S, L_2, Q, E)$$

is a syntax framework for (L, I). Notice that E is actually the direct evaluation function for F.

3.2 Example: Gödel Numbering

Let L be the expressions (i.e., terms and formulas) of a first-order language of natural number of arithmetic, and let $\mathcal{A}$ be the alphabet of symbols occurring in the expressions of L. Once again the expressions of L can be viewed as strings over the alphabet $\mathcal{A}$. As Kurt Gödel famously showed in 1931 [17], the syntactic structure of an expression e of L can be represented by a natural number called the Gödel number of e. Define G to be the total function that maps each expression of L to its Gödel number. G is injective since each expression in L has a unique Gödel number. The terms of L, which denote natural numbers, can thus be used to reason *directly* about Gödel numbers and *indirectly* about the syntactic structure of the expressions of L.

We will show that this reasoning system based on Gödel numbers can be interpreted as a syntax framework. Let $L_{\rm t}$ be the set of terms in L and $L_{\rm f}$ be the set of formulas in L. Then

$$I = (L, \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}, V),$$

where $L = L_{\rm t} \cup L_{\rm f}$, $\mathbb{N}$ is the set of natural numbers, and $V = V_{\rm t} \cup V_{\rm f}$, is an interpreted language corresponding to the language given in Example 2.1.2.

Since $G : L \rightarrow \mathbb{N}$ is an injective, total function that maps each expression in L to its Gödel number, $R = (\mathbb{N}, G)$ is a syntax representation of L. Since $L_{\rm t} \subseteq L$, $\mathbb{N} \subseteq \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}$, and V restricted to $L_{\rm t}$ is $V_{\rm t} : L_{\rm t} \rightarrow \mathbb{N}$, $(L_{\rm t}, I)$ is a syntax language for R.

Let $Q : L \rightarrow L_{\rm t}$ be a total function that maps each expression $e \in L$ to a term $t \in L_{\rm t}$ such that $V_{\rm t}(t) = G(e)$. Q is injective since each expression in L has a unique Gödel number. For $e \in L$,

$$V(Q(e)) = V_{\rm t}(Q(e)) = G(e),$$

and thus Q satisfies the Quotation Axiom if $L_{\rm obj} = L$, $D_{\rm syn} = \mathbb{N}$, $V_{\rm syn} = G$, and $L_{\rm syn} = L_{\rm t}$.

Let $E : L_{\rm t} \rightarrow L$ be the function that, for all $t \in L_{\rm t}$, $E(t)$ is the expression in L whose Gödel number is $V_{\rm t}(t)$ if $V_{\rm t}(t)$ is a Gödel number of some expression in L and $E(t)$ is undefined otherwise. For $t \in L_{\rm t}$ such that $E(t)$ is defined,

$$V(E(t)) = V(G^{-1}(V_{\rm t}(t))) = V(G^{-1}(V(t))),$$

and thus E satisfies the Evaluation Axiom if $L_{\rm obj} = L$, $D_{\rm syn} = \mathbb{N}$, $V_{\rm syn} = G$, and $L_{\rm syn} = L_{\rm t}$. Since not every natural number is a Gödel number of an expression in L, $G : L \rightarrow \mathbb{N}$ is not surjective and thus $E : L_{\rm t} \rightarrow L$ is partial. For an expression e of L, $Q(e) = t$ such that $V_{\rm t}(t) = G(e)$ by the definition of Q and then $E(t) = e$ by the the definition of E. Hence $E(Q(e)) = e$ and so the law of disquotation holds universally.

Therefore,

$$F = (\mathbb{N}, G, L_{\rm t}, Q, E)$$

is a syntax framework for (L, I). Notice that E is actually the direct evaluation function for F.

Define $L'_{\rm t}$ to be the sublanguage of $L_{\rm t}$ such that $t \in L'_{\rm t}$ iff $V(t)$ is a Gödel number of some expression in L. Then

$$F' = (\mathbb{N}, G, L'_{\rm t}, Q, E'),$$

where E' is E restricted to $L'_{\rm t}$, is a syntax framework for (L, I) in which the evaluation function E' is total.

3.3 Example: Expressions as Members of an Inductive Type

In the previous two subsections we saw how strings of symbols and Gödel numbers can be used to represent the syntactic structure of expressions. These two syntax representations are very popular, but they are not convenient for practical applications. In this example we will see a much more practical syntax representation in which expressions are represented as members of an inductive type.

Let $L_{\rm prop}$ be a language of propositional logic (with logical connectives for negation, conjunction, and disjunction). An interpreter for the language $L_{\rm prop}$ is a program that receives user input (which we assume is a string), parses the input into a usable internal representation (i.e., a parse or syntax tree), computes the value of the internal representation in the form of a new internal representation, and then displays the new internal representation in a user-readable form (which we again assume is a string). We will describe the components of such an interpreter.

Let `formula` be the type of the internal data structures representing the propositional formulas in $L_{\rm prop}$. This type can be implemented as an inductive type, e.g., in F# [41] as:

```
type formula =
  | True
  | False
  | Var of string
  | Neg of formula
  | And of (formula * formula)
  | Or  of (formula * formula)
```

Notice that the type constructors correspond precisely to the various ways of constructing a well-formed formula in propositional logic.

The interpreter for L_{prop} is the composition of the following functions:

1. A function `parse` of type `string` $\rightarrow$ `formula` which parses a user input string into an internal representation of a well-formed propositional formula — or raises an error if the input does not represent one. For the sake of simplicity, we assume that L_{prop} is chosen so that `parse` is injective.

2. A function `value` of type `formula` $\rightarrow$ `formula` which determines the truth value of a propositional formula of L_{prop} — or simplifies it in cases that contain unknown variables. We will later see how this function also requires an additional input φ of a variable assignment.

3. A function `print` of type `formula` $\rightarrow$ `string` which prints an internal representation of a formula as a string for the user. We assume that, for each string e representing a well-formed propositional formula of L_{prop}, `print`(`parse`(e)) $= e$.

For example, suppose $e =$ `"p & true"` is a user input string that denotes a propositional formula in L_{prop}. Then $f =$ `parse`(e) $=$ `And (Var "p",True)` is the expression of type `formula` that denotes its internal representation, $f' =$ `value`(f) $=$ `Var "p"` is the expression of type `formula` that denotes its computed value, and $e' =$ `print`(f') $=$ `"p"` is the string representation of its computed value. Hence the interpretation of e is

`print`(`value`(`parse`(e))).

We will show that this system for interpreting propositional formulas can be regarded as a syntax framework. This example demonstrates how to add a syntax representation and a syntax language to a language that does not inherently support reasoning about syntax. It also demonstrates

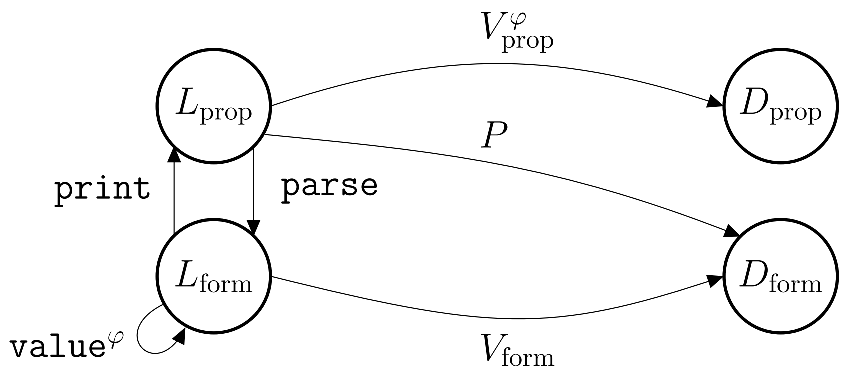

Figure 5: Domains and Mappings related to L_{prop}

that any typical implementation of a formal language can be interpreted as a syntax framework.

Let L_{prop} be the set of well-formed formulas of propositional logic represented by strings as discussed above, $D_{\mathrm{prop}} = \{\mathtt{T}, \mathtt{F}\}$ be the domain of truth values (i.e., the values formulas of propositional logic denote), and $V^{\varphi}_{\mathrm{prop}} : L_{\mathrm{prop}} \rightarrow D_{\mathrm{prop}}$ be the semantic valuation function for propositional logic relative to a variable assignment φ. Then $I_{\mathrm{prop}} = (L_{\mathrm{prop}}, D_{\mathrm{prop}}, V^{\varphi}_{\mathrm{prop}})$ is an interpreted language for propositional logic.

Similarly, let L_{form} be the set of expressions of type `formula`, D_{form} be the members of the inductive type `formula`, and $V_{\mathrm{form}} : L_{\mathrm{form}} \rightarrow D_{\mathrm{form}}$ be the semantic valuation function for the expressions of type `formula`. Then $I_{\mathrm{form}} = (L_{\mathrm{form}}, D_{\mathrm{form}}, V_{\mathrm{form}})$ is also an interpreted language. This secondary interpreted language is the augmentation that we are adding to the language of propositional logic in order to represent the syntax of L_{prop}. Using functions similar to `parse`, `value`, and `print` shown above, we can implement the language I_{prop} in a programming language.

Let $P : L_{\mathrm{prop}} \rightarrow D_{\mathrm{form}}$ be the function such that, for $e \in L_{\mathrm{prop}}$, $P(e)$ is the value of type `formula` denoted by `parse`(e). Then P is an injective, total function since each $e \in L_{\mathrm{prop}}$ has exactly one parse tree that represents the syntactic structure of e. Therefore, $R = (D_{\mathrm{form}}, P)$ is a syntax representation of L_{prop}. The structures I_{prop}, I_{form}, and R are depicted in Figure 5.

Let $L = L_{\mathrm{prop}} \cup L_{\mathrm{form}}$, $D = D_{\mathrm{prop}} \cup D_{\mathrm{form}}$, and $V^{\varphi} = V^{\varphi}_{\mathrm{prop}} \cup V_{\mathrm{form}}$. V^{φ} is a function since the two functions $V^{\varphi}_{\mathrm{prop}}$ and V_{form} have disjoint domains. Then $I = (L, D, V^{\varphi})$ is an interpreted language and (L_{form}, I) is a syntax language for R by construction.

The tuple

$$F = (D_{\text{form}}, P, L_{\text{form}}, \texttt{parse}, \texttt{print})$$

is a syntax framework for (L_{prop}, I) since:

1. $R = (D_{\text{form}}, P)$ is a syntax representation of L_{prop} as shown above.
2. (L_{form}, I) is syntax language for R as shown above.
3. **Quotation Axiom**: For all $e \in L_{\text{prop}}$, $P(e) = V_{\text{form}}(\texttt{parse}(e))$ by definition, and thus

$$V^{\varphi}(\texttt{parse}(e)) = V_{\text{form}}(\texttt{parse}(e)) = P(e).$$

4. **Evaluation Axiom**: For all $e \in L_{\text{form}}$, $P^{-1}(V_{\text{form}}(e)) = \texttt{parse}^{-1}(e) = \texttt{print}(e)$ since $\texttt{print}(\texttt{parse}(e)) = e$, and thus

$$V^{\varphi}(\texttt{print}(e)) = V^{\varphi}(P^{-1}(V_{\text{form}}(e))) = V^{\varphi}(P^{-1}(V^{\varphi}(e))).$$

Since $\texttt{print}(\texttt{parse}(e)) = e$ holds for all expressions in L_{prop}, the Law of Disquotation holds universally.

The syntax framework for this example provides the structure that is needed to understand the function $\texttt{value}^{\varphi}$ shown in Figure 5 as an implementation of the semantic valuation function $V^{\varphi}_{\text{prop}}$. The formula that specifies $\texttt{value}^{\varphi}$,

$$V^{\varphi}(e) = V^{\varphi}(\texttt{print}(\texttt{value}^{\varphi}(\texttt{parse}(e)))),$$

illustrates the interplay of syntax and semantics that is inherent in its meaning.

The approach employed in this third example, in which the syntactic values are members of an inductive type, is commonly used in programming to represent syntax (see [14]). It utilizes a *deep embedding* [2] of the object language L_{obj} into the full underlying formal language L.

3.4 Further Remarks

Remark 3.4.1 (Variable Binding) None of the standard examples discussed above treat variable binding constructions in any special way. There are other syntax representation methods that identify expressions that are the same up to a renaming of the variables that are bound by variable binders. One method is *higher-order abstract syntax* [26, 31] in which the syntactic structure of an expression with variable binders is represented by a term in typed lambda calculus. Another method is *nominal techniques* [15, 32] in which the swapping of variable names can be explicitly expressed. The paper [28] combines quotation/evaluation techniques with nominal techniques. $\Box$

Remark 3.4.2 (Types) The languages in a syntax framework are not required to be typed. However, it is natural that, if an expression e in the object language is of type α, then $Q(e)$ should be of some type $\mathsf{expr}(\alpha)$. The operator expr behaves like the necessity operator $\Box$ in modal logic [7]. An important design decision for such a type system is whether or not every expression of the syntax language equals a quotation of an expression. In other words, should a syntax framework with a type system admit only expressions in the syntax language that denote the syntactic structure of well-formed expressions or should it admit in addition expressions that denote the syntactic structure of ill-formed expressions. Recall that in the example of subsection 3.2 the syntax language of F contains the latter kind of expressions, while the syntax language of F' contains only the former kind. $\Box$

4 Syntax Frameworks with Built-In Operators

The three examples in the previous section illustrate how a syntax framework provides the means to reason about the syntax of a designated object language $L_{\rm obj} \subseteq L$. In all three examples, only *indirect statements* about the syntax of $L_{\rm obj}$ can be expressed in L, while direct statements using Q and E can be expressed in the metalanguage of L. In this section we will explore syntax frameworks in which *direct statements* about the syntax of $L_{\rm obj}$, such as $E(Q(e)) = e$, can be expressed in L itself.

4.1 Built-in Quotation and Evaluation

Let $I = (L, D, V)$ be an interpreted language, $L_{\rm obj}$ be a sublanguage of L, and $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ be a syntax framework for $(L_{\rm obj}, I)$. F has *built-in quotation* if there is an operator (which we will denote as quote) such that, for all $e \in L_{\rm obj}$, $Q(e)$ is the syntactic result of applying the operator to e (which we will denote as $\mathsf{quote}(e)$). F has *built-in evaluation* if there is an operator (which we will denote as eval) such that, for all $e \in L_{\rm syn}$, $E(e)$ is the syntactic result of applying the operator to e (which we will denote as

$\mathsf{eval}(e))$ whenever $E(e)$ is defined.[1] There are similar definitions of built-in quotation and evaluation for syntax frameworks in interpreted theories.

Assume F has both built-in quotation and evaluation. Then quotations and evaluations are expressions in L, and F thus provides the means to reason directly in L about the interplay of the syntax and semantics of the expressions in $L_{\rm obj}$. In particular, it is possible to specify in L the semantic meanings of transformers. The following lemma shows that, since the quotations and evaluations in F begin with the operators quote and eval, respectively, E cannot be the direct evaluation for F.

Lemma 4.1.1 *Suppose F is a syntax framework that has built-in quotation and evaluation. Then $E \neq E^*$.*

Proof Suppose $E = E^*$. Let $e \in L_{\rm obj}$. Then

$$\begin{aligned}
e &= V_{\rm syn}^{-1}(V_{\rm syn}(e)) && (1)\\
&= V_{\rm syn}^{-1}(V_{\rm sem}(\mathsf{quote}(e))) && (2)\\
&= E^*(\mathsf{quote}(e)) && (3)\\
&= E(\mathsf{quote}(e)) && (4)\\
&= \mathsf{eval}(\mathsf{quote}(e)) && (5)
\end{aligned}$$

(1) is by the fact that $V_{\rm syn}$ is total on $L_{\rm obj}$; (2) is by built-in quotation and the Quotation Axiom; (3) is by the definition of the direct evaluation function; (4) is by hypothesis; and (5) is by the fact that E is built in. Hence $e = \mathsf{eval}(\mathsf{quote}(e))$, which is a contradiction since these are syntactically distinct expressions. $\Box$

The syntax framework F is *replete* if the object language of F is equal to the full language of F (i.e., $L_{\rm obj} = L$) and F has both built-in quotation and evaluation. A replete syntax framework whose full language is L has the facility to reason about the syntax of all of L within L itself. F is *weakly replete* if $L_{\rm syn} \subseteq L_{\rm obj}$ and F has both built-in quotation and evaluation. There are similar definitions of replete and weakly replete for syntax frameworks in interpreted theories. We will give two examples of a replete syntax framework, one in the next subsection and one in section 6. We will also give another example in section 6 of a syntax framework that is almost replete.

[1]If $L_{\rm obj}$ is a typed language, it may be necessary for the eval operator to include a parameter that ranges over the types of the expressions in $L_{\rm obj}$.

Remark 4.1.2 A *biform theory* [4, 9, 13] is a combination of an axiomatic theory and an algorithmic theory. It is a basic unit of mathematical knowledge that consists of a set of *concepts*, *transformers*, and *facts*. The concepts are symbols that denote mathematical values and, together with the transformers, form a language L for the theory. The transformers are programs whose input and output are expressions in L; they represent syntax-based algorithms like reasoning rules. The facts are statements expressed in L about the concepts and transformers. A logic with a replete syntax framework (such as Chiron discussed in subsection 6.3) is well-suited for formalizing biform theories [9]. $\Box$

4.2 Example: Lisp

We will show that the Lisp programming language with a simplified semantics is an instance of a syntax framework with built-in quotation and evaluation.

Choose some standard implementation of Lisp. Let L be the set of S-expressions that do not change the Lisp valuation context when they are evaluated by the Lisp interpreter. Let $V : L \to L \cup \{\bot\}$ be the total function that, for all S-expressions $e \in L$, $V(e)$ is the S-expression the interpreter returns when e is evaluated if the interpreter returns an S-expression in L and $V(e) = \bot$ otherwise. $I = (L, L \cup \{\bot\}, V)$ is thus an interpreted language.

$R = (L, \mathsf{id}_L)$, where id_L is the identity function on L, is a syntax representation of L since each S-expression represents its own syntactic structure. Let L' be the sublanguage of L such that, for all $e \in L$, $e \in L'$ iff $V(e) \neq \bot$. It follows immediately by the definition of L' that (L', I) is a syntax language for R.

Let $Q : L \to L'$ be the total function that maps each $e \in L$ to the S-expression $(\texttt{quote}\ e)$. For $e \in L$, $Q(e) \in L'$ since $V((\texttt{quote}\ e)) = e \neq \bot$. Q is obviously injective. For $e \in L$,

$$V(Q(e)) = V((\texttt{quote}\ e)) = e = \mathsf{id}_L(e),$$

and thus Q satisfies the Quotation Axiom if $L_{\text{obj}} = L$, $D_{\text{syn}} = L$, $V_{\text{syn}} = \mathsf{id}_L$, and $L_{\text{syn}} = L'$.

Let $E : L' \to L$ be the total function that, for all $e \in L'$, $E(e)$ is the S-expression $(\texttt{eval}\ e)$. For all $e \in L'$,

$$V(E(e)) = V((\texttt{eval}\ e)) = V(V(e)) = V(\mathsf{id}_L^{-1}(V(e))).$$

(Notice that $V(V(e))$ is always defined since $e \in L'$.) Thus E satisfies the Evaluation Axiom if $L_{\rm obj} = L$, $D_{\rm syn} = L$, $V_{\rm syn} = \mathsf{id}_L$, and $L_{\rm syn} = L'$.

Therefore,

$$F = (L, \mathsf{id}_L, L', Q, E)$$

is a replete syntax framework for (L, I).

Suppose L were the full set of S-expressions, including the S-expressions that modify the Lisp valuation context when they are evaluated by the interpreter. Then, in order to interpret Lisp as a syntax framework, we would need to extend the notion of a *syntax framework* to the notion of *contextual syntax framework* as mentioned in Remark 2.7.4

4.3 Example: Liar Paradox

The virtue of a syntax framework with built-in quotation and evaluation is that it provides the means to express statements about the interplay of the syntax and semantics of the expressions in $L_{\rm obj}$ in L. On the other hand, the vice of such a syntax framework is that, if L is sufficiently expressive, the liar paradox can be expressed in L using quotation and evaluation.

Let $I = (L, \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}, V)$ be the interpreted language and $F' = (\mathbb{N}, G, L'_{\rm t}, Q, E')$ be the syntax framework for (L, I) given in subsection 3.2. Assume that V is defined so that the axioms of first-order Peano arithmetic are satisfied (see [24]). Assume also that F' has been modified so that it has both built-in quotation and built-in evaluation.

We claim E' cannot be total. Assume otherwise. By the diagonalization lemma [5], there is an expression $A \in L$, such that $V(A) = V(\mathsf{quote}(\neg(\mathsf{eval}(A))))$. Then

$$\begin{aligned} V(\mathsf{eval}(A)) &= V(G^{-1}(V(A))) && (1)\\ &= V(G^{-1}(V(\mathsf{quote}(\neg(\mathsf{eval}(A)))) && (2)\\ &= V(G^{-1}(G(\neg(\mathsf{eval}(A))))) && (3)\\ &= V(\neg(\mathsf{eval}(A))) && (4) \end{aligned}$$

(1) is by built-in evaluation, the totality of E', and the Evaluation Axiom; (2) is by the definition of A; (3) is by built-in quotation and the Quotation Axiom, and (4) is by the fact G is total on L. Hence $V(\mathsf{eval}(A)) = V(\neg(\mathsf{eval}(A)))$, which contradicts the fact that V never assigns a formula and its negation the same truth value. Therefore, E' cannot be total and, in particular, cannot be total on quotations.

The formula $\mathsf{eval}(A)$ expresses the *liar paradox* and the argument above is a proof of Alfred Tarski's 1933 theorem on the undefinability of truth [38, 39, 40], which says that built-in evaluation cannot serve as a truth predicate over all formulas. This example demonstrates why evaluation is allowed to be partial in a syntax framework: if evaluation were required to be total, the notion of a syntax framework would not cover reasoning systems with built-in quotation and evaluation in which the liar paradox can be expressed.

4.4 Example: Gödel Numbering with Built-In Quotation

A syntax framework without built-in quotation and evaluation can sometimes be modified to have built-in quotation or evaluation.

Let $I = (L, \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}, V)$ be the interpreted language and $F' = (\mathbb{N}, G, L'_{\rm t}, Q, E')$ be the syntax framework for (L, I) given in subsection 3.2. Extend L to the language L^* and $L'_{\rm t}$ to $L^*_{\rm t}$ by adding a new operator quote so that $\mathsf{quote}(e) \in L^*_{\rm t}$ for all $e \in L^*$. Extend G to $G^* : L^* \rightarrow \mathbb{N}$ so that $G^*(e)$ is the Gödel number of e for all $e \in L^*$. Extend V to $V^* : L^* \rightarrow \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}$ so that $V^*(\mathsf{quote}(e)) = G^*(e)$ for all $e \in L^*$. And, finally, define $Q^*(e)$ to be $\mathsf{quote}(e)$ for all $e \in L^*$. (We do not need to change the definition of E'.) Then $I^* = (L^*, \mathbb{N} \cup \{\textsc{t}, \textsc{f}\}, V^*)$ is an interpreted language and

$$F^* = (\mathbb{N}, G^*, L^*_{\rm t}, Q^*, E')$$

is a syntax framework for (L^*, I^*) that has built-in quotation.

See [12] for further discussion on the challenges involved in modifying a traditional logic to embody the structure of a replete syntax framework.

5 Quasiquotation

Quasiquotation is a parameterized form of quotation in which the parameters serve as holes in a quotation that are filled with the values of expressions. It is a very powerful syntactic device for specifying expressions and defining macros. Quasiquotation was introduced by Willard Quine in 1940 in the first version of his book *Mathematical Logic* [34]. It has been extensively employed in the Lisp family of programming languages [1].[2]

We will show in this section how quasiquotation can be defined in a syntax framework. Let $I = (L, D, V)$ be an interpreted language, $L_{\rm obj}$ be a sublanguage of L, and $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ be a syntax framework for $(L_{\rm obj}, I)$.

[2]In Lisp, the standard symbol for quasiquotation is the backquote (‘) symbol, and thus in Lisp, quasiquotation is usually called *backquote*.

5.1 Marked Expressions

Suppose $e \in L$. A *subexpression* of e is an occurrence in e of some $e' \in L$. We assume that there is a set of *positions* in the syntactic structure of e such that each subexpression of e is indicated by a unique position in e. Two subexpressions e_1 and e_2 of e are *disjoint* if e_1 and e_2 do not share any part of the syntactic structure of e.

Let $e \in L_{\rm obj}$. A *marked expression* derived from e is an expression of the form $e[(p_1, e_1), \ldots, (p_n, e_n)]$ where $n \geq 0$, $p_1, \ldots, p_n$ are positions of pairwise disjoint subexpressions of e, and $e_1, \ldots, e_n$ are expressions in L. Define $L^{\rm m}_{\rm obj}$ to be the set of marked expressions derived from members of $L_{\rm obj}$.

Let $S : L^{\rm m}_{\rm obj} \rightarrow L_{\rm obj}$ be the function that, given a marked expression $m = e[(p_1, e_1), \ldots, (p_n, e_n)] \in L^{\rm m}_{\rm obj}$, simultaneously replaces each subexpression in e at position p_i with $E^*(e_i)$ (the application of the direct evaluation function for F to e_i) for all i with $1 \leq i \leq n$. $S(e)$ will be undefined if either $E^*(e_i)$ is undefined or $E^*(e_i)$ does not have the same type as the subexpression at position p_i for some i with $1 \leq i \leq n$.

5.2 Quasiquotation

Define $\overline{Q} : L^{\rm m}_{\rm obj} \rightarrow L_{\rm syn}$ to be the (possibly partial) function such that, if $m = e[(p_1, e_1), \ldots, (p_n, e_n)] \in L^{\rm m}_{\rm obj}$, then $\overline{Q}(m) = Q(S(m))$. $\overline{Q}(m)$ is defined iff $S(m)$ is defined. For $m \in L^{\rm m}_{\rm obj}$, $\overline{Q}(m)$ is called the *quasiquotation* of m.[3]

F has *built-in quasiquotation* if there is an operator (which we will denote as $\mathsf{quasiquote}$) such that, for all $m = e[(p_1, e_1), \ldots, (p_n, e_n)] \in L^{\rm m}$, $\overline{Q}(m)$ is the syntactic result of applying the operator to $e, p_1, \ldots, p_n, e_1, \ldots, e_n$ (which we will denote as $\mathsf{quasiquote}(m)$).

5.3 Backquote in Lisp

Let us continue the example in subsection 4.2 involving Lisp with a simplified semantics. In Lisp, a *backquote* of L is an expression of the form `‘e` where e is an S-expression in L in which some of the subexpressions of e are marked by a comma (`,`). For example,

```
‘(+ 2 ,(+ 3 1))
```

is a backquote in which `(+ 3 1)` is a subexpression marked by a comma. We will restrict our attention to unnested backquotes. The Lisp interpreter

[3]The position-expression pairs (p_i, e_i) in a quasiquotation $\overline{Q}(e[(p_1, e_1), \ldots, (p_n, e_n)])$ are sometimes called *antiquotations*.

normally returns an S-expression when it evaluates a backquote `` `$e \in L$. In this case the S-expression returned is obtained from $e$ by replacing each subexpression $e'$ in $e$ marked by a comma with the S-expression $V(e')$. For example, when evaluating `` `(+ 2 ,(+ 3 1))``, the interpreter returns `(+ 2 4)`. Let L be extended to L^* to include the backquotes of L and $V^* : L^* \to L^* \cup \{\bot\}$ be the total function such that, for all S-expressions and backquotes $e \in L^*$, $V^*(e)$ is the S-expression the interpreter returns when e is evaluated if the interpreter returns an S-expression and $V^*(e) = \bot$ otherwise.

A backquote `` ` ``e in L^* corresponds to a marked expression $m = e[(p_1, e_1), \ldots, (p_n, e_n)] \in L^{\mathrm{m}}$ where each p_i is the position of a subexpression `,`e_i in e marked by a comma for all i with $1 \le i \le n$. Let `` ` ``$e \in L^*$ be a backquote and $m = e[(p_1, e_1), \ldots, (p_n, e_n)] \in L^{\mathrm{m}}$ be a marked expression that corresponds to it. We will show that the semantic value of the backquote `` ` ``e, when it is not $\bot$, is the same as the semantic value of the quasiquotation $\overline{Q}(m)$. Assume $V^*(\text{`}e) \neq \bot$. Then

$$\begin{aligned} V^*(\text{`}e) &= S(m) && (1)\\ &= V(Q(S(m))) && (2)\\ &= V(\overline{Q}(m)) && (3) \end{aligned}$$

(1) is by the semantics of backquote and the definition of S since

$$V(e_i) = \mathsf{id}_L^{-1}(V(e_i)) = V_{\mathrm{syn}}^{-1}(V(e_i)) = E^*(e_i)$$

for each i with $1 \le i \le n$. (2) is by the Quotation Axiom and the fact that V_{syn} is the identity function. And (3) is by the definition of $\overline{Q}(m)$.

6 Examples from the Literature

6.1 Example: Lambda Calculus

In 1994 Torben Mogensen [27] introduced a method of self representing and interpreting terms of lambda calculus. We will analyze this method and demonstrate how the self-interpretation of lambda calculus is almost an instance of a replete syntax framework.

Let $\Lambda = V \mid \Lambda\ \Lambda \mid \lambda V\,.\,\Lambda$ be the set of λ-terms where V is a countable set of variables. Λ is the language of lambda calculus consisting of all the λ-terms. A λ-term is a *normal form* if β-reduction cannot be applied to it. Given a λ-term M, let the *normal form of* M, NF_M, be the normal form that results from repeatedly applying β-reduction to M until a normal form is obtained. The normal form of M is undefined if a normal form is never obtained after repeatedly applying β-reduction to M. We will introduce two different syntax representations of this language. The first syntax representation of Λ uses an inductive type similar to subsection 3.3 such that V_A is the syntactic valuation function where:

$$\begin{aligned} V_A(x) &= \texttt{Var}(x) && (1)\\ V_A(M\ N) &= \texttt{App}(V_A(M), V_A(N)) && (2)\\ V_A(\lambda x\ .\ M) &= \texttt{Abs}(\lambda x\ .\ V_A(M)) && (3) \end{aligned}$$

Let D_A be the domain of values of this inductive type. Then $R_A = (D_A, V_A)$ is a syntax representation of Λ.

Mogensen [27] suggests a different syntax representation of lambda calculus. Let $\lceil\cdot\rceil : \Lambda \to \mathrm{NF}_\Lambda$ be a *representation schema* for lambda calculus such that:

$$\begin{aligned} \lceil x\rceil &= \lambda abc\ .\ a\ x && (1)\\ \lceil M\ N\rceil &= \lambda abc\ .\ b\ \lceil M\rceil\ \lceil N\rceil && (2)\\ \lceil \lambda x\ .\ M\rceil &= \lambda abc\ .\ c\ (\lambda x\ .\ \lceil M\rceil) && (3) \end{aligned}$$

where a, b, c are variables not occurring free in the λ-terms M and N. This representation of λ-terms is an equivalent representation to the method described earlier which utilizes the constructs of lambda calculus itself instead of an external data type.

Then $R_\Lambda = (\mathrm{NF}_\Lambda, \lceil\cdot\rceil)$ is a syntax representation of Λ and $(\mathrm{NF}_\Lambda, I_\Lambda)$ is a syntax language for R_Λ. Notice that, since $\lceil M\rceil$ is in normal form for any $M \in \Lambda$, then trivially $\lceil M\rceil \twoheadrightarrow_\beta \lceil M\rceil$.

Let a *self-interpreter* E be a λ-term such that for any $M \in \Lambda$, $E\lceil M\rceil$ is β-equivalent to M, i.e., $E\lceil M\rceil =_\beta M$ (which means $\mathrm{NF}_{E\lceil M\rceil}$ and NF_M are α-convertible when these normal forms exist). Mogensen proves that the λ-term

$$E = Y\ \lambda e\ .\ \lambda m\ .\ m\ (\lambda x\ .\ x)\ (\lambda mn\ .\ (e\ m)\ (e\ n))\ (\lambda m\ .\ \lambda v\ .\ e(m\ v)),$$

where Y is the Y-combinator, is a self-interpreter. Define $E_\Lambda : \mathrm{NF}_\Lambda \to \Lambda$ to be the partial function such that $E_\Lambda(M) = E\ M$ if $M = \lceil N_M\rceil$ for some λ-term N_M and is undefined otherwise.

Theorem 6.1.1 *Let Λ be the language of lambda calculus and $I_\Lambda = (\Lambda, \mathrm{NF}_\Lambda \cup \{\bot\}, \twoheadrightarrow_\beta)$ be the interpreted language of lambda calculus as defined* *earlier. Let $\lceil \cdot \rceil$ be the representation schema of Λ and E_Λ be the function defined above. Then*

$$F_\Lambda = (\mathrm{NF}_\Lambda, \lceil \cdot \rceil, \mathrm{NF}_\Lambda, \lceil \cdot \rceil, E_\Lambda)$$

is a syntax framework for (Λ, I_Λ)*.*

Proof F_Λ is a syntax framework since it satisfies the four conditions of Definition 2.3.1:

1. $R_\Lambda = (\mathrm{NF}_\Lambda, \lceil \cdot \rceil)$ is a syntax representation of Λ.

2. $(\mathrm{NF}_\Lambda, I_\Lambda)$ is syntax language for R_Λ.

3. $\lceil \cdot \rceil : \Lambda \rightarrow \mathrm{NF}_\Lambda$ is an injective, total function such that, for all $M \in \Lambda$, $\lceil M \rceil \twoheadrightarrow_\beta \lceil M \rceil$ (Quotation Axiom).

4. $E_\Lambda : \mathrm{NF}_\Lambda \rightarrow \Lambda$ is a partial function such that, for all $M \in \mathrm{NF}_\Lambda$ with $M = \lceil N_M \rceil$ for some λ-term N_M, $E_\Lambda(M) = E\ M = E \lceil N_M \rceil =_\beta N_M$ (Evaluation Axiom) since E is a self-interpreter.

□

F_Λ is almost replete: Λ is both the object and full language of F_Λ and F_Λ has built-in evaluation, but F_Λ does not have built-in quotation.

6.2 Example: The Ring Tactic in Coq

Coq [6] is an interactive theorem prover based on the calculus of inductive constructions. Let R be a ring with the associative, commutative binary operators $+$ and $*$ and the constants 0 and 1 that are the identities of $+$ and $*$, respectively. A *polynomial* in R is an expression that consists of the constants of R, the operators $+$ and $*$, and variables $v_0, v_1, \ldots$ of type R.

The *ring tactic* in Coq is a polynomial simplifier that converts any polynomial to its equivalent *normal form*. The normal form of a polynomial is defined as the ordered sum of unique monomials in lexicographic order.

Earlier we mentioned that syntax-based operations such as (symbolically) computing derivatives require a syntax framework to manipulate and reason about syntax using quotation and evaluation. Polynomial simplification is a term rewriter that uses the quotation and evaluation mechanisms. The `ring` tactic in Coq automatically quotes and simplifies every polynomial expression.

Internally, when the `ring` tactic is applied, the polynomials are represented by an inductive type `polynomial`. The Coq reference manual [6] defines this type as:

```
Inductive polynomial : Type :=
  | Pvar : index -> polynomial
  | Pconst : A -> polynomial
  | Pplus : polynomial -> polynomial -> polynomial
  | Pmult : polynomial -> polynomial -> polynomial
  | Popp : polynomial -> polynomial.
```

which represents polynomials similar to the inductive type example in subsection 3.3.

Let L be the language of Coq, D be the semantic domain of values in the calculus of inductive constructions, and V be the semantic interpreter of Coq, then $I = (L, D, V)$ is the interpreted language for Coq. Let $L_R \subseteq L$ be the language of polynomials of type R (i.e., expressions in L that are built with operators and constants of R and variables $v_0, v_1, \ldots$ as defined earlier), $L_{\text{poly}} \subseteq L$ be the language of expressions belonging to the inductive type `polynomial`, $D_{\text{poly}} \subseteq D$ be the image of L_{poly} under V, and V_{poly} be the internal quotation mechanism of Coq the `ring` tactic uses to lift polynomial expressions in L_R to expressions in L_{poly}. Then $(D_{\text{poly}}, V_{\text{poly}})$ is a syntax representation and (L_{poly}, I) is a syntax language for this syntax representation which is suitable for describing the `ring` tactic in Coq.

Coq's ring normalization library (`Ring_normalize.v`) also defines an interpretation function that transforms a polynomial expression of type `polynomial` back to a ring value of type R:

```
Fixpoint interp_p (p:polynomial) : A :=
  match p with
  | Pconst c => c
  | Pvar i => varmap_find Azero i vm
  | Pplus p1 p2 => Aplus (interp_p p1) (interp_p p2)
  | Pmult p1 p2 => Amult (interp_p p1) (interp_p p2)
  | Popp p1 => Aopp (interp_p p1)
  end.
```

To finish a definition of a syntax framework for the `ring` tactic in Coq, we need to construct two functions $Q : L_R \to L_{\text{poly}}$ and $E : L_{\text{poly}} \to L_R$ in the metalanguage of Coq. Their definitions are:

1. For all $e \in L_R$, $Q(e)$ is the $e' \in L_{\text{poly}}$ such that $V(e') = V_{\text{poly}}(e)$.

2. For all $e' \in L_{\text{poly}}$, $E(e')$ is the $e \in L_R$ such that $V(e) = V(\texttt{interp\_p})(V(e'))$.

Then $F = (D_{\text{poly}}, V_{\text{poly}}, L_{\text{poly}}, Q, E)$ is a syntax framework for (L_R, I).

Notice that the two functions Q and E are not normally present in Coq and were constructed by the machinery in Coq described above specifically to satisfy the requirements of a syntax framework. Although the concepts of the syntax language and the syntax representation arose naturally from the internal mechanism for the `ring` tactic in Coq, a syntax framework for the `ring` tactic does *not* reside in Coq as explicitly as our previous examples.

6.3 Example: Chiron

Chiron [10, 11], is a derivative of von-Neumann-Bernays-Gödel (NBG) set theory [18, 24] that is intended to be a practical, general-purpose logic for mechanizing mathematics. Unlike traditional set theories such as Zermelo-Fraenkel (ZF) and NBG, Chiron is equipped with a type system, and unlike traditional logics such as first-order logic and simple type theory, Chiron admits undefined terms. The most noteworthy part of Chiron is its facility for reasoning about the syntax of expressions that includes built-in quotation and evaluation.

We will assume that the reader is familiar with the definitions concerning Chiron in [11]. Let L be a language of Chiron, $\mathcal{E}_L$ be the set of expressions in L, M be a standard model for L, D_M be the set of values in M, V be the valuation function in M, and φ be an assignment into M. Then $I = (\mathcal{E}_L, D_M, V_\varphi)$ is an interpreted language.

D_M includes certain sets called *constructions* that are isomorphic to the syntactic structures of the expressions in $\mathcal{E}_L$. H is a function in M that maps each expression in $\mathcal{E}_L$ to a construction representing it. Let D_{syn} be the range of H and $\mathcal{T}_{\text{syn}}$ be the set of terms a such that $V_\varphi(a) \in D_{\text{syn}}$. For $e \in \mathcal{E}_L$, define $Q(e) = (\mathsf{quote}, e)$. For $a \in \mathcal{T}_{\text{syn}}$, define $E(a)$ as follows:

1. If $V_\varphi(a)$ is a construction that represents a type and $H^{-1}(V_\varphi(a))$ is eval-free, then $E(a) = (\mathsf{eval}, a, \mathsf{type})$.

2. If $V_\varphi(a)$ is a construction that represents a term, $H^{-1}(V_\varphi(a))$ is eval-free, and $V_\varphi(H^{-1}(V_\varphi(a))) \neq \bot$, then $E(a) = (\mathsf{eval}, a, \mathsf{C})$.

3. If $V_\varphi(a)$ is a construction that represents a formula and $H^{-1}(V_\varphi(a))$ is eval-free, then $E(a) = (\mathsf{eval}, a, \mathsf{formula})$.

4. Otherwise, $E(a)$ is undefined.

Theorem 6.3.1 *$F = (D_{\mathrm{syn}}, H, \mathcal{T}_{\mathrm{syn}}, Q, E)$ is a syntax framework for $(\mathcal{E}_L, I)$.*

Proof F is a syntax framework since it satisfies the four conditions of the Definition 2.3.1:

1. H maps each $e \in \mathcal{E}_L$ to a construction that represents the syntactic structure of e. Thus D_{syn} is a set of values that represent syntactic structures and $H : \mathcal{E}_L \to D_{\mathrm{syn}}$ is injective and total. So R is a syntax representation of $\mathcal{E}_L$.

2. I is an interpreted language. $\mathcal{E}_L \subseteq \mathcal{E}_L$. $\mathcal{T}_{\mathrm{syn}} \subseteq \mathcal{E}_L$. $D_{\mathrm{syn}} \subseteq D_M$ (since since D_{syn} is the range of H, $H : \mathcal{E}_L \to D_{\mathrm{v}}$, and $D_{\mathrm{v}} \subseteq D_M$). And V_φ restricted to $\mathcal{T}_{\mathrm{syn}}$ is a total function $V' : \mathcal{T}_{\mathrm{syn}} \to D_{\mathrm{syn}}$. So $(\mathcal{T}_{\mathrm{syn}}, I)$ is a syntax language for R.

3. Let $e \in \mathcal{E}_L$. Then $V_\varphi(Q(e)) = V_\varphi((\mathsf{quote}, e)) = H(e)$ by the definition of Q and the definition of V_φ on quotations. So $Q : \mathcal{E}_L \to \mathcal{T}_{\mathrm{syn}}$ is an injective, total function such that, for all $e \in \mathcal{E}_L$, $V_\varphi(Q(e)) = H(e)$.

4. Let $a \in \mathcal{T}_{\mathrm{syn}}$ such that $E(a)$ is defined. Hence $V_\varphi(a)$ is a construction that represents a type, term, or formula. If $V_\varphi(a)$ represents a type, term, or formula, let k be type, C, or $\mathsf{formula}$, respectively. Then $V_\varphi(E(a)) = V_\varphi((\mathsf{eval}, a, k)) = V_\varphi(H^{-1}(V_\varphi(a)))$ by the definition of E and the definition of V_φ on evaluations. So $E : \mathcal{T}_{\mathrm{syn}} \to \mathcal{E}_L$ is a partial function such that, for all $a \in \mathcal{T}_{\mathrm{syn}}$, $V_\varphi(E(a)) = V_\varphi(H^{-1}(V_\varphi(a)))$ whenever $E(a)$ is defined.

Finally, F is replete since $\mathcal{E}_L$ is both the object and full language of F and F has built-in quotation and evaluation. □

Quasiquotation is a notational definition in Chiron; it is not a built-in operator in Chiron as quotation and evaluation are [11]. The quasiquotation defined in Chiron is semantically equivalent to the notion of quasiquotation defined in the previous section.

7 Conclusion

We have introduced a mathematical structure called a *syntax framework* consisting of six major components:

1. A formal language L with a semantics.
2. A sublanguage L_{obj} of L that is the object language of the syntax framework.
3. A domain D_{syn} of values that represent the syntactic structures of expressions in L_{obj}.
4. A sublanguage L_{syn} of L whose expressions denote values in D_{syn}.
5. A quotation function $Q : L_{\text{obj}} \to L_{\text{syn}}$.
6. An evaluation function $E : L_{\text{syn}} \to L_{\text{obj}}$.

A syntax framework provides the means to reason about the interplay of the syntax and semantics of the expressions in L_{obj} using quotation and evaluation. In particular, it provides three basic syntax activities:

1. Expressing statements in L about the syntax of L_{obj}.
2. Constructing expressions in L_{syn} that denote values in D_{syn}.
3. Employing expressions in L_{syn} as expressions in L_{obj}.

These activities can be used to specify, and even implement, transformers that map expressions in L_{obj} to expressions in L_{obj}. They are needed, for example, to specify the rules of differentiation and to prove that these rules correctly produce representations of expressions that denote derivatives [12]. A syntax framework also provides a basis for defining a notion of quasiquotation which is very useful for the second basic activity.

When a syntax framework has built-in quotation and evaluation, it provides the means to reason *directly* in L about the syntax and semantics of the expressions in L_{obj}. However, in this case, the evaluation function E cannot be the direct evaluation function (Lemma 2.4.3) and, if L is sufficiently expressive, E cannot be total on quotations (subsection 4.3) and thus the Law of Disquotation (Lemma 2.4.1) cannot hold universally.

We showed that the notion of a syntax framework embodies a common structure found in a variety of systems for reasoning about the interplay of syntax and semantics. We did this by showing how several examples of such systems can be regarded as syntax frameworks. Three of these examples were the standard syntax-reasoning systems based on expressions as strings, Gödel numbers, and members of an inductive type. The other, more sophisticated, examples were taken from the literature.

We have also mentioned that a syntax framework is not adequate for modeling syntax reasoning in programming languages with mutable variables. This requires a generalization of a *syntax framework* to a *contextual framework* that will be presented in a future paper.

Acknowledgments

The authors are grateful to Marc Bender, Jacques Carette, Michael Kohlhase, Russell O'Connor, and Florian Rabe for their comments about the paper.

References

[1] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999. Technical report BRICS-NS-99-1, University of Aarhus, 1999.

[2] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156. North-Holland, 1993.

[3] H. Cappelen and E. LePore. Quotation. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2012 edition, 2012.

[4] J. Carette and W. M. Farmer. High-level theories. In A. Autexier, J. Campbell, J. Rubio, M. Suzuki, and F. Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 232–245. Springer-Verlag, 2008.

[5] R. Carnap. *Die Logische Syntax der Sprache*. Springer-Verlag, 1934.

[6] Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.4*, 2012. Available at `http://coq.inria.fr/distrib/V8.4/refman/`.

[7] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48:555–604, 2001.

[8] F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: A short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[9] W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.

[10] W. M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.

[11] W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007. Revised 2012.

[12] W. M. Farmer. The formalization of syntax-based mathematical algorithms using quotation and evaluation. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag, 2013.

[13] W. M. Farmer and M. von Mohrenschildt. An overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.

[14] D. P. Friedman and M. Wand. *Essentials of Programming Languages.* The MIT Press, 2008.

[15] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. *Formal Aspects of Computing*, 13:341–363, 2002.

[16] M. Glanzberg. Truth. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Spring 2013 edition, 2013.

[17] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[18] K. Gödel. *The Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory*, volume 3 of *Annals of Mathematical Studies.* Princeton University Press, 1940.

[19] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16, 2006.

[20] V. Halbach. *Axiomatic Theories of Truth.* Cambridge University Press, 2011.

[21] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.ps.gz`.

[22] P. Koellner. On reflection principles. *Annals of Pure and Applied Logic*, 157:206–219, 2009.

[23] H. Leitgeb. What theories of truth should be like (but cannot be). *Philosophy Compass*, 2:276–290, 2007.

[24] E. Mendelson. *Introduction to Mathematical Logic.* Taylor & Francis, Inc., fifth edition, 2009.

[25] Microsoft F# Developer Center. F#. `http://msdn.microsoft.com/en-us/fsharp`, 2011.

[26] D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd et al., editor, *Computational Logic — CL 2000*, volume 1861 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 2000.

[27] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2:345–364, 1994.

[28] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *Journal of Functional Programmming*, 15:893–939, 2005.

[29] U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory.* PhD thesis, Chalmers University of Technology, 2007.

[30] U. Norell. Dependently typed programming in Agda. In A. Kennedy and A. Ahmed, editors, *TLDI*, pages 1–2. ACM, 2009.

[31] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208. ACM Press, 1988.

[32] A. M. Pitts. Nominal Logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[33] A. Polonsky. Axiomatizing the Quote. In Marc Bezem, editor, *Computer Science Logic (CSL'11) — 25th International Workshop/20th Annual Conference of the EACSL*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 458–469, Dagstuhl, Germany, 2011. Schloss Dagstuhl — Leibniz-Zentrum für Informatik.

[34] W. V. O. Quine. *Mathematical Logic: Revised Edition.* Harvard University Press, 2003.

[35] Rice University Programming Languages Team. Metaocaml: A compiled, type-safe, multi-stage programming language. `http://www.metaocaml.org/`, 2011.

[36] T. Sheard and S. P. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37:60–75, 2002.

[37] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248:211–242, 2000.

[38] A. Tarski. Pojęcie prawdy w językach nauk dedukcyjnych (The concept of truth in the languages of the deductive sciences). *Prace Towarzystwa Naukowego Warszawskiego*, 3(34), 1933.

[39] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1935.

[40] A. Tarski. The concept of truth in formalized languages. In J. Corcoran, editor, *Logic, Semantics, Meta-Mathematics*, pages 152–278. Hackett, second edition, 1983.

[41] The F# Software Foundation. F#. `http://fsharp.org/`, 2014.