Command injection attacks, continuations, and the Lambek calculus

Hayo Thielecke
School of Computer Science
University of Birmingham
H.Thielecke@cs.bham.ac.uk

This paper shows connections between command injection attacks, continuations, and the Lambek calculus: certain command injections, such as the tautology attack on SQL, are shown to be a form of control effect that can be typed using the Lambek calculus, generalizing the double-negation typing of continuations. Lambek's syntactic calculus is a logic with two implicational connectives taking their arguments from the left and right, respectively. These connectives describe how strings interact with their left and right contexts when building up syntactic structures. The calculus is a form of propositional logic without structural rules, and so a forerunner of substructural logics like Linear Logic and Separation Logic.

1 Introduction

The aim of this paper is to draw connections between three at first sight disparate topics ranging from the practical to the theoretical side of computer science:

1. Command injection attacks;
2. continuations and control effects;
3. the Lambek calculus, a presentation of syntax as a logic or type system.

Depending on the reader's background, the following may serve as an introduction to command injections, the Lambek calculus, or both. Continuations will serve as the glue between these topics, so to speak, and a basic familiarity with control operators and their typing is assumed.

We briefly recall some background on continuations. Continuations in one form or another occur in many areas of computer science, ranging from compiling to logic. Like many fundamental concepts, they have been discovered independently [21], and we may even see Gödel's work on double negation as one of the first such discoveries.

Consider an expression language with a control operator `return`, as given in Figure 1. As shown in some of the earliest work on continuation semantics [23], such a language can be given a semantics by taking a continuation as a parameter.

For example, the expression

$$(\texttt{return}\ 42) + 666$$

evaluates to 42. Intuitively, this is because the evaluation context

$$(\bigcirc + 666)$$

has been discarded by the control operator.

Olivier Danvy & Ugo de'Liguoro (Eds.): WoC'15 Postproceedings
EPTCS 212, 2016, pp. 81–96, doi:10.4204/EPTCS.212.6

$$
\begin{array}{rcl}
E & ::= & E + E \\
 & | & n \\
 & | & \texttt{return}\ n
\end{array}
$$

$$
\begin{array}{rcl}
[\![E_1 + E_2]\!] & = & \lambda k.[\![E_1]\!](\lambda x_1.[\![E_2]\!](\lambda x_2.k(x_1 + x_2))) \\
[\![n]\!] & = & \lambda k.kn \\
[\![\texttt{return}\ n]\!] & = & \lambda k.n
\end{array}
$$

Figure 1: Control operator `return` and its continuation semantics

The typing of the continuation semantics is a generalized double negation:

$$[\![E]\!] : (\texttt{int} \to \texttt{int}) \to \texttt{int}$$

The typical presentation of continuation passing, following Plotkin [19], uses λ-calculus, but much of the machinery of continuations works in a more general situation. If fewer structural rules are assumed (omitting Contraction and Weakening), then connections with Linear Logic emerge [3, 9]. Lambek's syntactic calculus [15] goes further and removes the Exchange rule as well, making it a canonical logic for reasoning about strings. Control operators, by giving access to the current continuation, have an effect of the surrounding evaluation context. Analogously, in the Lambek calculus, a binary operator has a syntactic effect in the sense that it consumes some of its syntactic context, as given by symbols to its left and right.

2 Command injection attacks

In programming language theory, one usually assumes that all matters of parsing have been settled, so that the syntax is given as abstract syntax trees, rather than raw sequences of symbols, before language features such as types or effects are considered. However, complex software increasingly contains parsers and interpreters of various kinds, some for full programming languages, others for more restricted languages such as SQL or XML. Input to them is parsed at runtime, and may originate from untrusted sources. Consequently, syntax becomes a problem again, impacting the safety and security of the interpreters.

Command injection attacks form a large class of attacks on software (for an overview, see texts on secure programming, such as Dowd et al [6]). They may happen whenever user-malleable and potentially malicious fragments in some syntax are spliced into a syntactic context such that the resulting string is parsed and interpreted. It is crucial for the attack that the fragments to be combined are raw text that still has to be parsed, rather than some structured format such as abstract syntax trees. Of course an attacker could just inject syntactically invalid gibberish and provoke parsing errors. Depending on the robustness of error handling, that could amount to a mere nuisance or a denial of service attack. However, command injection attacks are far more pernicious by creating strings that are successfully parsed and therefore interpreted. By gaining access to the interpreter to run code of their choosing, attackers can violate integrity and confidentiality, rather than merely triggering errors.

SQL command injection [12] attacks are perhaps the best known example; in this case the constructed strings are SQL queries that are interpreted by the database management system. In some variants [12] of SQL command injection attacks, the attacker relies on injecting code with side effects, such as a `DROP` statement in SQL that destructively updates the database. In this paper we will however concentrate on a class of attacks that do not require side effects in the injected code and rely purely on *syntactically* subverting the constructed string. The so-called tautology attack is a notorious example. Simply put, a malicious user injects the string `OR 1 = 1`, which when combined with a Boolean test renders that test tautologically true and hence useless.

Command injection attacks are by no means confined to SQL injections. They may arise whenever data is mixed with code in the broadest sense of the word; for instance, XPath injection attacks are a recent example. So even if SQL attacks are defeated by standard secure coding techniques, it is reasonable to expect that more vulnerabilities and attacks will emerge as dynamic scripting languages and XML/HTML based technologies proliferate, in which the mixing of code and data or in-band signalling that security experts warn against is a widespread risk.

At first sight, command injection attacks appear to be due to side effects in the interpreted language. Indeed, some attacks rely on the presence of effectful operations, such as inserting `UPDATE` or `DROP` in so-called "piggyback" SQL command injection attacks. The tautology attack, however, afflicts the purely functional language of Boolean expressions, and it does so by syntactic means rather than any side effects. The "essence" [24] of such attacks can be made precise in terms of parse trees. Intuitively, the programmer has an implicit understanding that the data supplied by the user should be slotted into the parse tree of the query as a leaf, below the comparison to `password`. Instead, the insertion of the operator `OR` rearranges the parse tree, so that the operator is above the test, rendering it ineffective by disjunction with the tautology. In this paper, we will focus exclusively on such syntactic attacks on purely functional languages.

A simple example of a syntactic command injection attack is known as a tautology attack. Suppose a dynamically constructed SQL query contains a comparison of the string `password` to some string supplied by the user, in order to check the user's authorization. Details of SQL syntax are not important here, but the idea of the syntactically malicious input is as follows. The query is constructed by concatenating a string ending in "`password = '`" with the user input to construct a boolean expression. This test is part of an SQL statement, as in" `SELECT * FROM table WHERE...`". If the user supplies the input "`foo`", the concatenation contains the test "`password = 'foo'`", as intended. In an attack, the user injects an operator, by supplying the input "`foo' OR 1 = 1 --`". The resulting test is

```
password = 'foo' OR 1 = 1
```

which always evaluates to true due to the tautology `1 = 1`. Using this technique, attackers may read confidential data for other users, bypass password authentication, and the like, with many variations on this theme of injecting operators [12].

Given that the phenomenon of syntactic command injection is so general, and independent of many details of the particular technologies being exploited, we aim to address it at the appropriate level of abstraction. We would like to reason about the syntactic effect, as it were, that a malicious input has on its context, similar to the way that a type and effect system [16] lets us reason about side effects. The effect can be subtle, as a malicious input string needs to conform closely to the structure of the surrounding string it is intended to attack. If for instance some delimiters are added to the latter, the original attack string may fail and produce only something syntactically ill formed.

A central thesis of this paper is that the required logical tools are already available in mathematical linguistics—perhaps surprisingly so. Lambek's syntactic calculus [15] describes syntax with two (left

String with a hole: `password = ◯`

Legitimate input: `foo`

Combined string: `password = foo`

Malicious input: `foo OR 1 = 1`

Combined string: `password = foo OR 1 = 1`

Figure 2: Tautology attack

and right) function types that capture how a phrase takes other phrases as arguments from the left or right. Using these connectives, we will explain how the harmless input "`foo`" differs from the malicious input "`foo OR 1 = 1`" that can take its place. In essence, the malicious input is effectful in that it seizes part of its context, just as a control operator does with its evaluation context. See Figure 2 (quotes are elided for simplicity).

3 Syntax and the Lambek calculus

As context-free grammars and parser generators for them are universally used in computer science, we will assume that the language we wish to reason about is given by an unambiguous context-free grammar. We will then use the Lambek calculus on top of the given grammar, not to define the language, but to describe the way its phrases combine.

We recall that a context-free grammar $\mathbf{G} = (\mathbf{T}, \mathbf{N}, \mathbf{P}, S)$ consists of a finite set $\mathbf{T}$ of terminal symbols, a finite set $\mathbf{N}$ of non-terminal symbols, a start symbol $S \in \mathbf{N}$, and a finite relation $\mathbf{P} \subseteq \mathbf{N} \times (\mathbf{N} \cup \mathbf{T})^*$ relating non-terminal symbols to strings of symbols. The elements (A, α) of $\mathbf{P}$ are called the rules or productions of the grammar, and often written as $A ::= \alpha$. (We avoid the common notation $A \to \alpha$, as it clashes with that for function types.)

We follow some notational conventions for grammars [1]. We write terminal symbols in typewriter font, as in "`a`" and "`=`". Non-terminal symbols are ranged over by A, B, C, while X and Y may be a terminal or a non-terminal symbol. Sentential forms (strings that may contain both non-terminals and terminals) are written as α, β, γ and δ. Words (strings of only terminal symbols) are ranged over by w, v, u. The empty sequence is written as ε. The one-step derivation relation $\Rightarrow$ holds between any two strings of the form

$$\beta A \gamma \Rightarrow \beta \alpha \gamma$$

whenever there is a production $(A, \alpha) \in \mathbf{P}$. The reflexive transitive closure of $\Rightarrow$ is written as $\stackrel{*}{\Rightarrow}$.

A grammar is called unambiguous if there is no word w that has two different parse trees with root S. If we assume our grammar to be unambiguous, we are justified in speaking of "the" parse tree of a word. For a non-terminal symbol A, we say A is useless if it does not participate in the derivation of any words, that is, if there are no α, β and w such that

$$S \stackrel{*}{\Rightarrow} \alpha A \beta \stackrel{*}{\Rightarrow} w$$

We will assume that there are no useless non-terminals in the grammar (as deleting them will not change the language of the grammar). If the grammar is unambiguous and contains no useless symbols, the language of each non-terminal is also unambiguous. Unambiguous grammars are important in practice

because compilers and interpreters compute meanings by induction over the parse tree; if there could be more than one such tree for a given input, there might be unintended outcomes.

When first reading about Lambek's syntactic calculus, one may perhaps be puzzled about whether to conceive of it as a form of syntax, a type system, or a logic. It is in a sense all of these, and that flexibility may be an advantage. There are two equivalent presentations of the calculus: the first as subtyping (to use current terminology), the other as a propositional logic in the style of Gentzen's sequent calculus.

Before going into the formal definitions of the calculus, it may be helpful to provide some intuition about its intended meaning, particularly compared to context-free grammars. Suppose we want to express that the operator `OR` takes a truth value T from the left and right, respectively, and produces a truth value. Using context-free grammars, we could write a grammar rule like the following:

$$T ::= T \texttt{ OR } T$$

(To keep the discussion simple, let us ignore the problem of ambiguous grammars for the moment.) In the Lambek calculus, we would express the same syntactic situation differently. We would say that there is a type of of words that produce a T if a T is placed to the left of them, which we write as $T \searrow T$. Moreover, there is a type of words that produce the latter type if another T is placed to the right of them, which is written as $(T \searrow T) \swarrow T$. That gives us a type of binary operators expecting a T on either side. Stating that `OR` is such a binary operator amounts to a subtyping judgement for the type `OR` (which contains exactly the word `OR`):

$$\texttt{OR} \leq (T \searrow T) \swarrow T$$

A useful intuition to bear in mind when reading the syntactic calculus is that the left-hand side is meant to be a subset of the right-hand side. In our example here, the set containing only `OR` is a subset of the set of binary operators, but not necessarily conversely, as there may be other such operators. Note that the order of writing is reversed compared to grammar rules: a grammar rule $A ::= B$ corresponds to $B \leq A$.

If we also have `1 = 1` $\leq T$, then we see that the partial application of `OR` to it still expects a T on its left:

$$\texttt{OR 1 = 1} \leq T \searrow T$$

Thus we can construct various operators by partial application (currying), as is familiar from functional programming. It would be possible to capture the syntax of a language entirely with such judgements, without the need for a context-free grammar. However, in our setting we assume a fixed grammar is given, and we use the Lambek calculus for reasoning about fragments of words like the `OR 1 = 1` above.

We assume that a fixed context-free grammar

$$\mathbf{G} = (\mathbf{T}, \mathbf{N}, \mathbf{P}, S)$$

of interest is given, and we define a version of the syntactic calculus specific to that grammar by using its symbols as the base types and importing its rules as axioms.

Definition 3.1 The types of (our variant of) the Lambek calculus are built up from the (terminal or non-terminal) symbols of our context-free grammar (ranged over by X) using the left and right arrow

$$\frac{\varphi_1 \circ \varphi_2 \leq \psi}{\varphi_2 \leq \varphi_1 \searrow \psi} \qquad \frac{\varphi_1 \circ \varphi_2 \leq \psi}{\varphi_1 \leq \psi \swarrow \varphi_2}$$

$$\frac{\varphi_2 \leq \varphi_1 \searrow \psi}{\varphi_1 \circ \varphi_2 \leq \psi} \qquad \frac{\varphi_1 \leq \psi \swarrow \varphi_2}{\varphi_1 \circ \varphi_2 \leq \psi}$$

$$\frac{}{\varphi \leq \varphi} \qquad \frac{\varphi_1 \leq \varphi_2 \qquad \varphi_2 \leq \varphi_3}{\varphi_1 \leq \varphi_3}$$

$$\frac{}{\varphi_1 \circ (\varphi_2 \circ \varphi_3) \leq (\varphi_1 \circ \varphi_2) \circ \varphi_3}$$

$$\frac{}{(\varphi_1 \circ \varphi_2) \circ \varphi_3 \leq \varphi_1 \circ (\varphi_2 \circ \varphi_3)}$$

Figure 3: Lambek's syntactic calculus, subtyping version

$$\frac{(A, X_1 \ldots X_n) \in \mathbf{P}}{X_1 \circ \ldots \circ X_n \leq A}$$

$$\frac{}{\varepsilon \circ \varphi \leq \varphi} \qquad \frac{}{\varphi \circ \varepsilon \leq \varphi}$$

$$\frac{}{\varphi \leq \varepsilon \circ \varphi} \qquad \frac{}{\varphi \leq \varphi \circ \varepsilon}$$

Figure 4: Additional rules for subtyping

connectives as well as the product connective.

$$
\begin{array}{rcll}
\varphi, \psi, \pi & ::= & X & \text{(Grammar symbol in } \mathbf{N} \cup \mathbf{T}\text{)} \\
& \mid & \varphi \searrow \psi & \text{(Left implication)} \\
& \mid & \psi \swarrow \varphi & \text{(Right implication)} \\
& \mid & \varphi \circ \psi & \text{(Product)} \\
& \mid & \varepsilon & \text{(Empty string type)}
\end{array}
$$

Definition 3.2 (Syntactic calculus, subtyping variant) The syntactic calculus consists of subtyping judgements of the form

$$\varphi \leq \psi$$

where φ and ψ are defined as in Definition 3.1. The rules for $\leq$ are given in Figures 3 and 4.

In the literature, the two implications are written as forward and backward slashes, "/" and "\". Reading such formulas can be tricky, particularly since two conventions exist. Lambek's notation places the result on top and reflects whether parameters are taken from left or right; Steedman's notation instead emphasizes the directionality of functions by placing the parameter on the left and the result on the right. We follow the Lambek style, but add arrowheads, writing "$\searrow$" and "$\swarrow$", to make it easier to see where the parameter and where the result is.

For reading nested implications, it is useful to bear in mind whether the arrows are pointing inward or outward. The following two types are isomorphic:

$$(\varphi_1 \searrow \psi) \swarrow \varphi_2 \text{ and } \varphi_1 \searrow (\psi \swarrow \varphi_2)$$

Intuitively, it makes no difference whether a binary operator consumes its left operand φ_1 or its right operand φ_2 first. We may write $\varphi_1 \searrow \psi \swarrow \varphi_2$ to mean either of them, just as brackets can be omitted due to $\circ$ being associative. By contrast, the two types where ψ occurs in a doubly negative position, as in:

$$(\varphi_1 \swarrow \psi) \searrow \varphi_2 \text{ and } \varphi_2 \swarrow (\psi \searrow \varphi_1)$$

are genuinely different, even when $\varphi_1 = \varphi_2$. Such doubly-negated types will be pertinent later on, particularly in Section 4.

The rules of the syntactic calculus are divided into those that are taken directly from Lambek's paper [15], gathered in Figure 3, and additional rules we add in this paper for using of the calculus on top of a fixed context-free grammar, presented in Figure 4.

In logical terms, the four rules for the implications in Figure 3 are quite natural if one thinks of implications (or arrow types) as adjoints of conjunctions (or products). In linear logic, the linear implication $\multimap$ is adjoint to $\varphi \otimes (-)$. In separation logic, the separating implication $-\!\!*$ is adjoint to the separating conjunction $\varphi * (-)$. In the Lambek calculus, the product is not commutative, so that $(-) \circ \varphi$ and $\varphi \circ (-)$ are not interchangeable. Consequently, there are two different adjoints $\searrow$ and $\swarrow$. The other four rules state that the subtyping relation $\leq$ is reflexive and transitive, and that the product $\circ$ is associative.

The rules in Figure 3 are the logical core of the calculus that applies to any language. In order to specialize the calculus to a particular language, we need additional axioms. In our case here, we import all productions of the given context-free grammar into the subtyping relation by adding axiom schemas

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \psi\ \Pi \lhd \pi}{\Psi\ (\psi \swarrow \varphi)\ \Phi\ \Pi \lhd \pi}\ (\swarrow\mathrm{L}) \qquad \frac{\Phi\ \varphi \lhd \psi}{\Phi \lhd \psi \swarrow \varphi}\ (\swarrow\mathrm{R})$$

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \psi\ \Pi \lhd \pi}{\Psi\ \Phi\ (\varphi \searrow \psi)\ \Pi \lhd \pi}\ (\searrow\mathrm{L}) \qquad \frac{\varphi\ \Phi \lhd \psi}{\Phi \lhd \varphi \searrow \psi}\ (\searrow\mathrm{R})$$

$$\frac{\Phi\ \varphi\ \psi\ \Psi \lhd \pi}{\Phi\ (\varphi \circ \psi)\ \Psi \lhd \pi}\ (\circ\mathrm{L}) \qquad \frac{\Phi \lhd \varphi \qquad \Psi \lhd \psi}{\Phi\ \Psi \lhd \varphi \circ \psi}\ (\circ\mathrm{R})$$

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \varphi\ \Pi \lhd \psi}{\Psi\ \Phi\ \Pi \lhd \psi}\ (\mathrm{CUT}) \qquad \frac{}{\varphi \lhd \varphi}\ (\mathrm{AX})$$

Figure 5: Sequent calculus variant of Lambek's syntactic calculus

$$\frac{(A, \alpha) \in \mathbf{P}}{\alpha \lhd A}\ (\mathbf{P}\lhd)$$

$$\frac{\Phi\ \Psi \lhd \varphi}{\Phi\ \varepsilon\ \Psi \lhd \varphi}\ (\varepsilon\mathrm{L}) \qquad \frac{}{\lhd\ \varepsilon}\ (\varepsilon\mathrm{R})$$

Figure 6: Additional rules for sequents

stating that the product of the symbols on the right-hand side of the production is a subtype of the non-terminal symbol on the left of the production. Note that the order of the subtyping is the reverse of the way grammars are written; it is in reduction rather than derivation order. As we can have epsilon productions (having an empty string on the right-hand side) in the grammar, we need to represent the empty string ε in the syntactic calculus as well. We do so by adding a type constant called ε and rules making it a left and right unit for product. Logically, ε is a natural addition to the calculus, in that it is the nullary analogue of Lambek's binary $\circ$ connective. These rules are given in Figure 4.

Lambek [15] also defines a sequent calculus, as this yields a decision procedure. In the literature, this sequent presentation is often referred to simply as the Lambek calculus. The calculus has left and right rules for the connectives, and it lacks all structural rules, going even further than Linear Logic and Separation Logic by banishing the Exchange rule [28]. Hence it distinguishes between a left and a right implication connective.

Definition 3.3 (Sequent presentation of the calculus) Let φ, ψ and π range over types as in Definition 3.1. We let the capital Greek letters Φ, Ψ and Π range over sequences of the form $\varphi_1 \ldots \varphi_n$, written without separating commas. Judgements are of the form $\Phi \lhd \varphi$, using the inference rules listed in Figures 5 and 6.

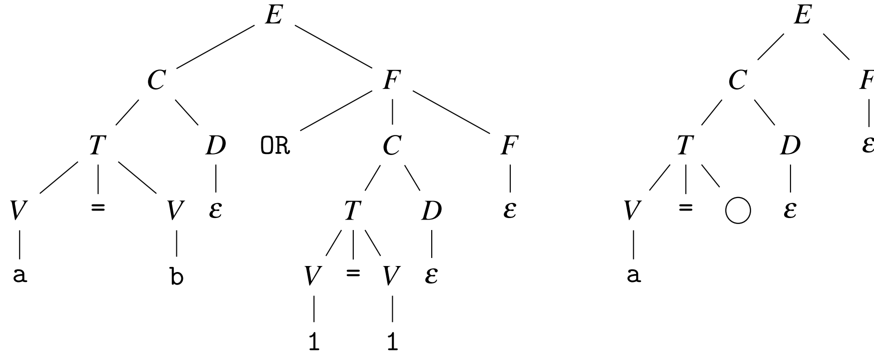

Figure 7: Parse tree for `a = b OR 1 = 1` and partial parse tree for `a =` ◯

The Lambek calculus has a simple denotational semantics. In particular, the two implications are interpreted as left and right language difference, product as concatenation, and judgements are interpreted as language inclusion [28]. For our version of the calculus, built on top of a context-free grammar, its semantics is as follows:

Definition 3.4 The denotation of a type in the syntactic calculus is a set of words, defined inductively as follows:

$$
\begin{aligned}
[\![X]\!] &= \{w \in \mathbf{T}^* \mid X \stackrel{*}{\Rightarrow} w\} \\
[\![\varphi \searrow \psi]\!] &= \{w \in \mathbf{T}^* \mid \forall v \in \mathbf{T}^*. v \in [\![\varphi]\!] \text{ implies } v\,w \in [\![\psi]\!]\} \\
[\![\psi \swarrow \varphi]\!] &= \{w \in \mathbf{T}^* \mid \forall v \in \mathbf{T}^*. v \in [\![\varphi]\!] \text{ implies } w\,v \in [\![\psi]\!]\} \\
[\![\varphi_1 \circ \varphi_2]\!] &= \{w_1\,w_2 \in \mathbf{T}^* \mid w_1 \in [\![\varphi_1]\!] \text{ and } w_2 \in [\![\varphi_2]\!]\} \\
[\![\varepsilon]\!] &= \{\varepsilon\}
\end{aligned}
$$

The semantics of a logical context $\Phi = \varphi_1 \ldots \varphi_n$ is the same as that of the n-fold product of φ_j, unless the sequence is empty, in which case it is the same as ε:

$$
\begin{aligned}
[\![\Phi]\!] &= \{\varepsilon\} \text{ if } \Phi \text{ is the empty context} \\
[\![\varphi_1\ \ldots\ \varphi_n]\!] &= \{w \in \mathbf{T}^* \mid w = w_1 \ldots w_n \text{ where} \\
&\qquad w_1 \in [\![\varphi_1]\!], \ldots, w_n \in [\![\varphi_n]\!]\,\}
\end{aligned}
$$

4 Reasoning about syntactic effects

In this section, we first investigate a command injection attack as an example of reasoning in the syntactic calculus. Building on what can be gleaned from that example, we then place it into a wider context of types and effects.

We define a toy grammar of Boolean expressions that is sufficient for discussing tautology attacks. The grammar uses a standard technique to avoid ambiguity and to ensure that conjunction binds more tightly than disjunction [1]. An expression E is a disjunction of conjunctions C of equality tests T between values V. A series of applications of `AND` is parsed as a C, but no `OR` can appear in a C.

$$
\begin{array}{lcl}
E & ::= & C\,F \\
F & ::= & \texttt{OR}\,C\,F \\
 & | & \varepsilon \\
C & ::= & T\,D \\
D & ::= & \texttt{AND}\,T\,D \\
 & | & \varepsilon \\
T & ::= & V \texttt{=} V \\
V & ::= & 1 \mid \ldots \mid \texttt{a} \mid \texttt{b} \mid \ldots
\end{array}
$$

We will now reason about the interaction between malicious inputs and vulnerable contexts using the syntactic calculus in its logical variant. The presentation as a sequent calculus, with left and right rules for the connectives, may look unfamiliar compared to type systems, which are usually in natural deduction style. Nonetheless, elimination rules for the arrow types are derivable:

$$
\frac{\Phi \lhd \varphi \qquad \Psi \lhd \varphi \searrow \psi}{\Phi\,\Psi \lhd \psi}\ (\searrow\text{E})
$$

A symmetric elimination rule $(\swarrow\text{E})$ exists for $\swarrow$. The proof for deriving $(\searrow\text{E})$ is as follows:

$$
\frac{\Psi \lhd \varphi \searrow \psi \qquad \dfrac{\Phi \lhd \varphi \qquad \overline{\psi \lhd \psi}\ (\text{AX})}{\Phi\,(\varphi \searrow \psi) \lhd \psi}\ (\searrow\text{L})}{\Phi\,\Psi \lhd \psi}\ (\text{CUT})
$$

Now suppose we have some code in which a string variable is concatenated with the string constant “`a = `”. The judgement `a =` $\lhd T \swarrow V$ tells us that the incomplete test expects a value to its right. If we supply such a value, say `b`, we infer using the derived elimination rule:

$$
\frac{\texttt{a=} \lhd T \swarrow V \qquad \texttt{b} \lhd V}{\texttt{a = b} \lhd T}\ (\swarrow\text{E})
$$

Now consider the attack string `b OR 1 = 1`. The essential point is that the attack reverses the role of operator and operand when concatenated with the fragment `a =`. In our calculus that is captured by the judgement

$$
\texttt{b OR 1 = 1} \lhd (T \swarrow V) \searrow E
$$

To infer this, we first note that we can derive in the grammar

$$
E \stackrel{*}{\Rightarrow} T\ \texttt{OR 1 = 1}
$$

which implies T `OR 1 = 1` $\lhd E$. From that, we construct the following proof:

$$
\frac{\dfrac{\dfrac{(V,\texttt{b}) \in \mathbf{P}}{\texttt{b} \lhd V}\ (\mathbf{P}\lhd) \qquad \overline{T \lhd T}\ (\text{AX})}{(T \swarrow V)\ \texttt{b} \lhd T}\ (\swarrow\text{L}) \qquad T\ \texttt{OR 1 = 1} \lhd E}{\dfrac{(T \swarrow V)\ \texttt{b OR 1 = 1} \lhd E}{\texttt{b OR 1 = 1} \lhd (T \swarrow V) \searrow E}\ (\searrow\text{R})}\ (\text{CUT})
$$

The two syntax fragments fit together to build an expression:

$$\frac{\texttt{a=} \lhd T \swarrow V \qquad \texttt{b OR 1 = 1} \lhd (T \swarrow V) \searrow E}{\texttt{a = b OR 1 = 1} \lhd E} (\searrow \text{E})$$

The fragment `a =` is now in the operand position of the application, rather than the operator position it had in `a = b` $\lhd T$.

As Figure 7 shows, the parse tree for `a = b OR 1 = 1` does not arise from completing the partial parse tree for `a =` $\bigcirc$ displayed to its right, where $\bigcirc$ indicates the "hole" position in the partial parse tree and syntax fragment.

The common name in the software security literature is Tautology attack, as it is the tautology that renders the test trivially true. However, in terms of reshaping the parse tree, the crucial ingredient is the fact that the injected operator `OR` has a lower precedence that the adjacent operator `=`, as the low precedence causes the `OR` node to move up in the parse tree.

Whether or not this way of combining pieces of syntax is an attack or a useful way to build up strings depends on what type we consider the function to have.

5 Double negation in command injection and linguistics

Note that the string with the syntactic effect is very sensitive to the context on which it has an effect. If we merely change the order in the latter, replacing `a =` $\bigcirc$ with $\bigcirc$ `= a`, the original attack does not work anymore, producing only an ungrammatical string. (In software security practice, that means attackers may need some reverse engineering skills to craft malicious input that fits into the syntactic context like a key into a lock.) The two connectives $\searrow$ and $\swarrow$ capture such ordering accurately. For injecting into $\bigcirc$ `= a`, the attack string is symmetric to the one above, with all implications reversed:

String	has type	fitting into context
`b OR 1 = 1`	$(T \swarrow V) \searrow E$	`a =` $\bigcirc$
`1 = 1 OR b`	$E \swarrow (V \searrow T)$	$\bigcirc$ `= a`

The main use of the Lambek calculus and related formalisms such as categorial grammar has been in linguistics rather than computer science (although Lambek's original paper discusses examples from logic along with those from natural language). Nonetheless, there are some intriguing parallels to the situations we have discussed.

Consider a naive syntax for English sentences. We have a type **Sen** of sentences and a type **Noun** of nouns. Names like "`Alice`" and "`Bob`" have type **Noun**. In the syntactic calculus, a transitive verb has a type like a binary operator, for instance

$$\texttt{knows} \lhd \mathbf{Noun} \searrow (\mathbf{Sen} \swarrow \mathbf{Noun})$$

So we can derive sentences like" `Alice knows Bob`" in the same way as deriving Boolean expressions like `a = b OR 1 = 1`. Lambek [15] observes that pronouns like `he` and `him` may occur in some positions in which nouns may occur. However, pronouns are more sensitive to their position, because" `he`" has to occur to the left of the verb, whereas "`him`" needs to be to the right of the verb. The calculus captures this grammatical fact by giving the two different double negations as the types of "`he`" and "`him`":

String	has type	fitting into context
`he`	$\mathbf{Sen} \swarrow (\mathbf{Noun} \searrow \mathbf{Sen})$	$\bigcirc$ `knows Alice`
`him`	$(\mathbf{Sen} \swarrow \mathbf{Noun}) \searrow \mathbf{Sen}$	`Alice knows` $\bigcirc$

Compare the difference between injection to the left or the right of the equality test discussed above.

6 Syntactic effects and control effects

Rather than supplying the expected type V, the attack string supplies a kind of generalized double negation of V, or more precisely, a V inside the negative position of two implications, as in

$$(\varphi_1 \swarrow V) \searrow \varphi_2 \text{ and } \varphi_2 \swarrow (V \searrow \varphi_1)$$

This typing generalizes the double negation of a formula A in logic, namely

$$(A \to \bot) \to \bot$$

The raising to a doubly-negated type is reminiscent of control operators in programming languages, and specifically the way that continuation passing style (CPS) introduces a form of double negation.

As a brief reminder of control operators, we consider the following simple use of the control operator `call/cc`:

$$(\texttt{call/cc}(\lambda k.42 + (k\,2))) + 1$$

Operationally, the current continuation is bound to the variable k when the call to `call/cc` is evaluated. Continuations can be represented as evaluation contexts [8], written as terms with a hole. In our example, the continuation bound to k could be written as

$$\bigcirc + 1$$

where $\bigcirc$ stand for the hole. When k is invoked in the subexpression $(k\,2)$, the value 2 is plugged into the hole of the continuation, and the whole expression thereby evaluates to 2+1 = 3. If the operational semantics of control operators is formalized in terms of evaluation contexts [8], a salient feature is that their evaluation can move upward in the surrounding evaluation context. Compare how in Figure 7 the injected operator `OR` moves upward in the parse tree from where it was inserted by, so to speak, elbowing itself across the node labelled T.

The application $(k\,2)$ appears to be of type `int`, in that it can be used as an argument of the operator $+$. The surrounding context, expecting an integer to be supplied, can be thought of as a function from `int` to some answer type `Ans`. If a value occurs in the context, it is passed to the function, yielding an answer. However, if the expression inside the context has control effects, it does not simply supply a value to its context. Instead, it takes the context as an argument and manipulates it (in the example above, by discarding it and using the continuation bound to k instead). Hence an expression with control effects of direct-style type `int` has a continuation-passing type that is a double negation of `int`:

$$(\texttt{int} \to \texttt{Ans}) \to \texttt{Ans}$$

In programming language semantics, these double negations are inserted by continuation passing style transforms [19]. The resulting connection [11] to classical logic has been studied intensely. As a further

refinement of this typing of control effects, an effect system can constrain how far up in the context the effect may reach [16, 14, 25]. In an effect system, we can control how effectful the argument of a function is. Suppose a function $f : C \to B$ is intended to be pure, which means it has no effect. The function type for pure functions is written as $C \xrightarrow{\emptyset} B$. However, if f calls a function passed as its argument, that function also needs to be pure. In the effect system, we can express this by giving a type of this form:

$$f : (A \xrightarrow{\emptyset} B) \xrightarrow{\emptyset} B$$

In an effect system, one often has a notion of sub-effecting, where a function that has fewer latent effects can be used where one with potentially more effects is expected. This fits well with our view here that a word with type φ also has the two double negations of φ as its type, but not conversely.

To sum up, we would like to draw the following analogy between an expression with control effects and a syntactic command injection attack string:

Expression	Context expects	CPS type
$(k\,2)$	`int`	$(\texttt{int} \to \texttt{Ans}) \to \texttt{Ans}$
`b OR 1 = 1`	V	$(T \swarrow V) \searrow E$
`1 = 1 OR b`	V	$E \swarrow (V \searrow T)$

Whereas the transformation of lambda terms into continuation passing style introduces nests of additional lambda abstractions, its analogue in the Lambek calculus is silent, so to speak. If a word w expects some word v on its right, we can regard v as expecting such a w on its left. So if v is a φ and w a $\psi \swarrow \varphi$, then we can equally regard the same word v as a $(\psi \swarrow \varphi) \searrow \psi$.

More formally, there are two derivable rules for introducing double negation:

$$\frac{\Phi \lhd \varphi}{\Phi \lhd (\psi \swarrow \varphi) \searrow \psi}\,(\text{DNIL}) \qquad \frac{\Phi \lhd \varphi}{\Phi \lhd \psi \swarrow (\varphi \searrow \psi)}\,(\text{DNIR})$$

These rules are derivable as follows:

$$\frac{\dfrac{\Phi \lhd \varphi \qquad \dfrac{}{\psi \lhd \psi}\,(\text{AX})}{(\psi \swarrow \varphi)\ \Phi \lhd \psi}\,(\swarrow\text{L})}{\Phi \lhd (\psi \swarrow \varphi) \searrow \psi}\,(\searrow\text{R}) \qquad \frac{\dfrac{\Phi \lhd \varphi \qquad \dfrac{}{\psi \lhd \psi}\,(\text{AX})}{\Phi\ (\varphi \searrow \psi) \lhd \psi}\,(\searrow\text{L})}{\Phi \lhd \psi \swarrow (\varphi \searrow \psi)}\,(\swarrow\text{R})$$

We recognize the syntactic control effects in the Lambek calculus as a form of continuation passing that goes even further in banishing structural rules than linear continuations [9] or linearly used continuations [3].

It is instructive to compare and contrast the two double-negation introductions in the Lambek calculus with double-negation introduction in intuitionistic and linear logic. Let us consider linear logic (as we can move from linear to intuitionistic logic by adding the Weakening and Contraction rules). There is no distinction between left and right implications, with only a single introduction and a single elimination rule for the linear implication $\multimap$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,(\multimap I) \qquad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\,(\multimap E)$$

These rules give rise to a double negation introduction. Its proof relies on the ability to exchange formulas in the context:

$$\dfrac{\dfrac{\dfrac{\overline{A \multimap R \vdash A \multimap R} \qquad \Gamma \vdash A}{A \multimap R, \Gamma \vdash R}\,(\multimap E)}{\Gamma, A \multimap R \vdash R}\,(\textsc{Exchange})}{\Gamma \vdash (A \multimap R) \multimap R}\,(\multimap I)$$

The corresponding proof term is $\lambda k.kx$, or more precisely:

$$x : A \vdash \lambda k.kx : (A \multimap R) \multimap R$$

In the Lambek calculus, by contrast, there is no need for λ-abstraction and application. The continuation passing version of a word w is just w itself.

7 Conclusions

In computer science generally, the Lambek calculus, particularly when presented as sequent calculus, is perhaps chiefly recognized as an early instance of a substructural logic, dating from 1958. As such, it precedes Linear Logic [10] and the Bunched Implications [20] logic underlying Separation Logic [22]. See van Benthem's overview [28] for a comparison to Linear Logic.

It is interesting to note that the other main scourges of software security apart from command injection are memory corruption and unsafe resource usage, and that substructural logics have been successful in reasoning about memory and resource usage [13, 17, 22, 18].

Our view here of command injection as a kind of control effect that seizes its context evolved from calculi for continuations [5, 8, 7] and type-and-effect systems that make such control effects explicit in the types [11, 16, 14, 25]. Behind each continuation, it is possible to introduce another level of continuations, sometimes called meta-continuations [5]. These additional levels of continuations are particularly vivid in the syntactic calculus, as they are implicitly always present due to the silent double-negation introduction, without the need to write additional λ-abstractions. In linguistics, the double negation introduction is also known as "type raising". There are further examples of effects similar to those of control operators, such as Montague's semantics of quantification. For an introduction aimed at computer scientists, see Barker's survey article [2].

For security policies or safety properties of programming languages, there are usually dynamic (run-time) and static (compile-time) approaches. A number of tools have been developed that defend against command injection attacks in a variety of languages [24]. For such tools, a major engineering challenge is to integrate them with existing technologies such as SQL and scripting languages with minimal intervention by programmers. While the use of parsing in such defences is one of the starting points of the present paper, the focus here is much more theoretical. Thiemann's Grammar-based Analysis of String Expressions [27] uses a language of types that appears closely related to the fragment of the Lambek calculus without implications $\searrow$ and $\swarrow$.

It remains a problem for future research to establish a formal connection between syntactic effects (such as those due to command injections) and control operators in the *semantics* of the language, given by parsing actions [26]. The semantic action of a string with a syntactic effect (such as those arising in command injections) may be conjectured to be equivalent to an expression with a suitable control operator, most likely a form of delimited continuation, such as shift/reset [4].

References

[1] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman (1985): *Compilers - Principles, Techniques and Tools*. Addison Wesley.

[2] Chris Barker (2004): *Continuations in Natural Language*. In: *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04)*, University of Birmingham Computer Science technical report CSR-04-1.

[3] Josh Berdine, Peter W. O'Hearn, Uday Reddy & Hayo Thielecke (2002): *Linear Continuation Passing*. *Higher-order and Symbolic Computation* 15(2/3), pp. 181–208, doi:10.1023/A:1020891112409.

[4] Olivier Danvy & Andrzej Filinski (1990): *Abstracting control*. In: *LISP and functional programming*, ACM, pp. 151–160, doi:10.1145/91556.91622.

[5] Olivier Danvy & Andrzej Filinski (1992): *Representing Control, a Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2(4), pp. 361–391, doi:10.1017/S0960129500001535.

[6] Mark Dowd, John McDonald & Justin Schuh (2006): *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison Wesley.

[7] Matthias Felleisen (1988): *The theory and practice of first-class prompts*. In: *Principles of Programming Languages (POPL)*, ACM, pp. 180–190, doi:10.1145/73560.73576.

[8] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker & Bruce F. Duba (1986): *Reasoning with Continuations*. In: *Logic in Computer Science (LICS)*, IEEE.

[9] Andrzej Filinski (1992): *Linear Continuations*. In: *Principles of Programming Languages (POPL)*, pp. 27–38, doi:10.1145/143165.143174.

[10] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[11] Timothy G. Griffin (1990): *A Formulae-as-Types Notion of Control*. In: *Principles of Programming Languages (POPL)*, ACM, pp. 47–58, doi:10.1145/96709.96714.

[12] William G.J. Halfond, Jeremy Viegas & Alessandro Orso (2006): *A Classification of SQL Injection Attacks and Countermeasures*. In: *ISSSE'06*, IEEE.

[13] Samin S. Ishtiaq & Peter O'Hearn (2001): *BI as an Assertion Language for Mutable Data Structures*. In: *Principles of Programming Languages (POPL)*, ACM, pp. 14–26.

[14] Pierre Jouvelot & David K. Gifford (1988): *Reasoning about Continuations with Control Effects*. In: *Programming Language Design and Implementation (PLDI)*, ACM, pp. 218–226, doi:10.1145/73141.74837.

[15] Joachim Lambek (1958): *The mathematics of sentence structure*. *American Mathematical Monthly* 65, pp. 154–170, doi:10.2307/2310058.

[16] John M. Lucassen & David K. Gifford (1988): *Polymorphic Effect Systems*. In: *Principles of Programming Languages (POPL '88)*, ACM, pp. 47–57, doi:10.1145/73560.73564.

[17] Gregory Morrisett, F. Smith & D. Walker (2000): *Alias Types*. In: *Proceedings European Symposium on Programming (ESOP)*, *LNCS* 1782, Springer, pp. 366–381, doi:10.1007/3-540-46425-5_24.

[18] Peter W. O'Hearn, Hongseok Yang & John C. Reynolds (2004): *Separation and Information Hiding*. In: *POPL'04*, pp. 268–280, doi:10.1145/1498926.1498929.

[19] Gordon D. Plotkin (1975): *Call-by-name, Call-by-value, and the λ-calculus*. *Theoretical Computer Science* 1(2), pp. 125–159, doi:10.1016/0304-3975(75)90017-1.

[20] David J. Pym (2002): *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers doi:10.1007/978-94-017-0091-7.

[21] John C. Reynolds (1993): *The Discoveries of Continuations*. *Lisp and Symbolic Computation* 6(3/4), pp. 233–247, doi:10.1007/BF01019459.

[22] John C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *Logic in Computer Science (LICS)*, IEEE, pp. 55–74, doi:10.1109/LICS.2002.1029817.

[23] Christopher Strachey & C. P. Wadsworth (1974): *Continuations: A Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG-11, Oxford University Computing Laboratory.

[24] Zhendong Su & Gary Wassermann (2006): *The essence of command injection attacks in web applications*. In: *Principles of Programming Languages (POPL)*, ACM, pp. 372–382, doi:10.1145/1111037.1111070.

[25] Hayo Thielecke (2003): *From Control Effects to Typed Continuation Passing*. In: *Principles of Programming Languages (POPL'03)*, ACM, pp. 139–149, doi:10.1145/640128.604144.

[26] Hayo Thielecke (2014): *On the Semantics of Parsing Actions*. *Science of Computer Programming* 84, pp. 52–76, doi:10.1016/j.scico.2013.04.010.

[27] Peter Thiemann (2005): *Grammar-based analysis of string expressions*. In: *TLDI*, ACM, pp. 59–70, doi:10.1145/1040294.1040300.

[28] Johan van Benthem (1991): *Language in Action*. *Journal of Philosophical Logic* 20(3), pp. 225–263, doi:10.1007/BF00250539.