Command injection attacks, continuations, and the Lambek calculus

Hayo Thielecke
University of Birmingham
Introduction

Continuations and double negation

Command injection attacks

The Lambek calculus

Command injections in the Lambek calculus

Related work
Overview

Aim: to connect

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

- No theorems
- Some intuitions (I hope)
Continuations

Consider an expression language with a control operator:

\[
\begin{align*}
\llbracket E_1 + E_2 \rrbracket &= \lambda k. \llbracket E_1 \rrbracket(\lambda x_1.\llbracket E_2 \rrbracket(\lambda x_2. k(x_1 + x_2))) \\
\llbracket n \rrbracket &= \lambda k. k\ n \\
\llbracket \text{return } n \rrbracket &= \lambda k. n
\end{align*}
\]

For example, the expression

\[
(\text{return } 42) + 666
\]

evaluates to 42. The continuation \((\bigcirc + 666)\) has been discarded.

The typing of the continuation semantics is a double negation:

\[
\llbracket E \rrbracket : (\text{int} \to \text{int}) \to \text{int}
\]
Continuations and generalized double negation

Intuitionistic negation:

\[ \neg \tau = \tau \rightarrow \bot \]

\[ \lbrack E \rbrack : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \]

is a generalized double negation using

\[ \neg \tau = \tau \rightarrow \text{int} \]

Other generalizations are possible:

\( \rightarrow \circ \) instead of one of both \( \rightarrow \): linear continuation passing

\[ \neg \tau = \tau \rightarrow \alpha \]: answer type polymorphism

In this talk: two different arrows, \( \downarrow \) and \( \swarrow \) instead of \( \rightarrow \)
SQL injection attack

Source: http://xkcd.com/327/
Malicious input:
Robert ’); DROP TABLE Students; --
Side effects vs syntactic effects

- DROP Table needs side-effects
- But: OR 1 = 1 does not; purely functional language
- String with a hole
- compare: expression with a hole for control operators
- not a side effect in the sense of state
- but still an effect: a control effect, in syntax
- make this precise: as a double negation type
Tautology attack

Purely functional language of Boolean expressions

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
Parsed like: (password = foo) OR (1 = 1)
Evaluates to: true

This is not just plugging the ⬜; something has happened: an effect.
What next: we need a calculus for syntax

In the control operator example, there was a term with a hole

\((\bigcirc + 666)\)

In the tautology injection example, there was a string with a hole

\(\text{password} = \bigcirc\)

For control operators, we have generalized double negation

\(((\neg) \to \text{int}) \to \text{int}\)

For strings, we need to generalize the double negation even more:
Lambek calculus with two arrows \(\downarrow\) and \(\uparrow\).
Lambek calculus

- Lambek’s syntactic calculus
- “The mathematics of sentence structure” (1958)
- A type system or logic for syntax
- Builds on older ideas from mathematical logic and linguistics
- Mainly used in linguistics
- Substructural calculus
- Very simple model in terms of strings
- Forerunner of:
  - Linear logic
  - Separation logic
Left and right arrows example: infix operator OR

Using a grammar:

\[ T ::= T \text{ OR} \ T \]

Using arrows:

\[ \text{OR} \triangleleft (T \downarrow T) \triangleright T \]

Partially applied, still expecting something on the left:

\[ \text{OR} \ 1 = 1 \triangleleft T \downarrow T \]

Fully applied:

\[ 1 = 0 \text{ OR} \ 1 = 1 \triangleleft T \]
Substructural logics

Typical rule in logic and type theory:

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad (\rightarrow I)
\]

In logic, there are substructural rules:

\[
\frac{\Gamma \vdash A}{\Gamma, B \vdash A} \quad \text{(Weakening)}
\]

\[
\frac{\Gamma, B, B \vdash A}{\Gamma, B \vdash A} \quad \text{(Contraction)}
\]

\[
\frac{B, C \vdash A}{C, B \vdash A} \quad \text{(Exchange)}
\]

In a substructural logics, some or all of there rules are absent.
Two implications instead of one

Lambda calculus:

\[ \Gamma, x : A \vdash M : B \]
\[ \Gamma \vdash \lambda x. M : A \rightarrow B \]

Lambek calculus:

\[ \varphi \triangleleft \Phi \triangleleft \psi \]
\[ \Phi \triangleleft \varphi \downarrow \psi \]
\[ (\nabla \text{R}) \]

\[ \Phi \varphi \triangleleft \psi \]
\[ \Phi \triangleleft \psi \checkmark \varphi \]
\[ (\downarrow \text{R}) \]
Semantics as sets of strings

\[ [X] = \{ w \in T^* \mid X \Rightarrow^* w \} \]
\[ [\varphi \downarrow \psi] = \{ w \in T^* \mid \forall v \in T^*. v \in [\varphi] \text{ implies } v \, w \in [\psi] \} \]
\[ [\psi \uparrow \varphi] = \{ w \in T^* \mid \forall v \in T^*. v \in [\varphi] \text{ implies } w \, v \in [\psi] \} \]
\[ [\varphi_1 \circ \varphi_2] = \{ w_1 \, w_2 \in T^* \mid w_1 \in [\varphi_1] \text{ and } w_2 \in [\varphi_2] \} \]
\[ [\epsilon] = \{ \epsilon \} \]
Sequent version of the Lambek calculus

\[ \Phi \varphi \quad \Psi \psi \Pi \varpi \]
\[ \frac{\psi \varphi \Pi \varpi}{\Phi \varphi \psi \Pi \varpi} \quad \text{(\(\check{\varphi}\) L)} \]
\[ \frac{\Phi \varphi \psi \Pi \varpi}{\Psi \left(\left(\varphi \varphi\right) \Pi \varpi\right)} \quad \text{(\(\check{\varphi}\) L)} \]
\[ \frac{\Phi \varphi \psi \Pi \varpi}{\Psi \Phi \left(\left(\varphi \varphi\right) \Pi \varpi\right)} \quad \text{(\(\check{\varphi}\) L)} \]
\[ \frac{\Phi \varphi \psi \Pi \varpi}{\left(\varphi \left(\Phi \varphi \psi \Pi \varpi\right)\right)} \quad \text{(\(\check{\varphi}\) R)} \]
\[ \frac{\Phi \varphi \psi \Pi \varpi}{\Phi \left(\left(\varphi \varphi\right) \Pi \varpi\right)} \quad \text{(\(\check{\varphi}\) R)} \]

Related work
Double negations

\[ \Phi \triangleleft \varphi \quad \text{(DNIL)} \]
\[ \Phi \triangleleft (\psi \triangleright \varphi) \triangleleft \psi \]

Intuitively:
Suppose we have a \( \varphi \).
Then if there is a
\[ (\psi \triangleright \varphi) \]
to the left of the \( \varphi \), we can get a \( \psi \).
So we have a
\[ (\psi \triangleright \varphi) \triangleleft \psi \]
Contrast the arrows with:
\[ \text{OR} \triangleleft (T \triangleleft T) \triangleright T \]
Tautology injection in the Lambek calculus

The malicious inputs have a double negation type:

<table>
<thead>
<tr>
<th>String</th>
<th>has type</th>
<th>fitting into context</th>
</tr>
</thead>
<tbody>
<tr>
<td>b OR 1 = 1</td>
<td>((T \lor V) \downarrow E)</td>
<td>a = (\bigcirc)</td>
</tr>
<tr>
<td>1 = 1 OR b</td>
<td>(E \lor (V \downarrow T))</td>
<td>(\bigcirc = a)</td>
</tr>
</tbody>
</table>
Lambek’s examples from linguistics

The pronoun “he” must be to the left of the verb; “him” must be to the right.

<table>
<thead>
<tr>
<th>String</th>
<th>has type</th>
<th>fitting into context</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>⊣ (Noun ⊢ Sen)</td>
<td>⊙ knows Alice</td>
</tr>
<tr>
<td>him</td>
<td>(Sen ⊣ Noun) ⊢ Sen</td>
<td>Alice knows ⊙</td>
</tr>
</tbody>
</table>
Double negation in control and command injection

Control operators:

```
return 42
```

in direct style, typed as int.

But semantically, a double negation of an int

```
(int → int) → int
```

Command injection:

```
1 = 1 OR b
```

plugged into a context expecting a string.

But actually, a double negation of a string,

```
E ⊨ (V ↘ T)
```
Parse tree surgery by syntactic effects

- Command injection attacks can also be understood in terms of parse trees.
- Connection to Lambek calculus: double negated types fit into a tree with a hole
- BUT: they do not stay inside the hole
- Instead: tree surgery
- Compare: expression tree manipulation by control operators
Parse tree with a hole
Command injection attacks, continuations, and the Lambek calculus

Introduction

Continuations and double negation

Command injection attacks

The Lambek calculus

Command injections in the Lambek calculus

Related work

Parse tree after tautology injection
Security as malicious versions of computer science

- Computer security
  = Satan’s computer

- Command injection attacks
  = Satan’s interpreter

- Command injection without side effects
  = Satan’s parser

- Classic buffer overflow overwriting return address
  = Satan’s continuation passing

- Advanced buffer overflow with return-oriented programming
  = Satan’s combinatory logic + continuation passing
Related work

- Command injection attack defences (Su and Wasserman)
- Grammar-based program analysis (Thiemann)
- Continuations in linguistics (Barker, Shan)
- Semantics of parsing actions (HT)

Directions for future work:
- Are syntactic command injections always characterised by double negations?
- Formally connect control operators and syntax?
- Correctness of program analysis via Lambek calculus?
- The Lambek calculus is cool.