PARSING WITH REGULAR EXPRESSIONS AND EXTENSIONS TO KLEENE ALGEBRA

Niels Bjørn Bugge Grathwohl
DIKU, November 4th 2015
PhD Thesis defense
<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Email</th>
<th>Gender</th>
<th>Phone</th>
<th>Country</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td><a href="mailto:john@gmail.com">john@gmail.com</a></td>
<td>male</td>
<td>123456</td>
<td>DK</td>
<td>John 123456</td>
</tr>
<tr>
<td>2</td>
<td>Benny</td>
<td><a href="mailto:benny@hotmail.com">benny@hotmail.com</a></td>
<td>male</td>
<td>98234</td>
<td>UK</td>
<td>Benny 98234</td>
</tr>
</tbody>
</table>

Want:

- **Streaming** – i.e., output while reading input.
- **Fast** – several Gbps throughput per core.
- Linear running time in the size of the input.

```
main := (row /
\n/)*)
col := /[^,\n]*/
row := ~(col /,,/) col "\t" ~,,/ ~(col /,,/) ~(col /,,/) ~,,/ ~col
```
Program is essentially a regular expression with outputs.

Regular expression syntax

\[ E ::= 0 | 1 | a | E_1 + E_2 | E_1E_2 | E_1^* \]

\((a \in \Sigma)\)

Examples

\((\Sigma = \{a, b\})\)

- \(a\)
- \((ab)^* + (a + b)^*\)
- \((a + b)^*\)
What is regular expression “matching”?

Expression \((ab)^* + (a + b)^*\)

Input \(s = \text{ababab}\)

- **acceptance testing**—is input string member of language? Answer: “Yes!”
- **subgroup matching**—substrings in input for subterms in expression. Answer: \([0, 6], [4, 2]\)
- **parsing**—what is the parse tree of the input?

```
   /\ 
  /   \  
 ab   ab  
    /\   ।
   ab ()
```
Input $s$ matches $E$ iff $s \in \mathcal{L}[E]$.

### Language interpretation

<table>
<thead>
<tr>
<th>$\mathcal{L}[E]$</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{L}[0]$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\mathcal{L}[1]$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$\mathcal{L}[a]$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$\mathcal{L}[E + F]$</td>
<td>${s \mid s \in \mathcal{L}[E]}$ union ${t \mid t \in \mathcal{L}[F]}$</td>
</tr>
<tr>
<td>$\mathcal{L}[EF]$</td>
<td>${s \cdot t \mid s \in \mathcal{L}[E], t \in \mathcal{L}[F]}$</td>
</tr>
<tr>
<td>$\mathcal{L}[E^*]$</td>
<td>$\mathcal{L}[E]^*$</td>
</tr>
</tbody>
</table>
Example

\[
\mathcal{L}[(ab)^* + (a + b)^*] \\
= \mathcal{L}[(ab)^*] \cup \mathcal{L}[(a + b)^*] \\
= \mathcal{L}[ab]^* \cup \mathcal{L}[a + b]^* \\
= \{ab\}^* \cup \{a, b\}^* \\
= \{\epsilon, ab, abab, \ldots\} \cup \{\epsilon, a, b, ab, ba, aba, \ldots\} \\
= \{\epsilon, a, b, aa, ab, aaa, aab, \ldots\}
\]
Construct parse tree from input $s$ such that *flattening* of parse tree is $s$.

### Type interpretation [FC’04;HN’11]

<table>
<thead>
<tr>
<th>$T[0]$</th>
<th>$\emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[1]$</td>
<td>${()}$</td>
</tr>
<tr>
<td>$T[a]$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$T[E+F]$</td>
<td>${\text{inl } v \mid v \in T[E]} \cup {\text{inr } w \mid w \in T[F]}$</td>
</tr>
<tr>
<td>$T[EF]$</td>
<td>$T[E] \times T[F]$</td>
</tr>
<tr>
<td>$T[E^*]$</td>
<td>${[v_1, \ldots, v_n] \mid n \geq 0, v_i \in T[E]}$</td>
</tr>
</tbody>
</table>

Values in $T[E]$ are *parse trees*. 
Example

$\mathcal{T}[[ab]^*+(a+b)^*]]$ contains the parse trees:

- inl $[(a, b), (a, b), (a, b)]$
- inr $[\text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b]$

which are not in $\mathcal{T}[[a+b]^*]]$!

So

$$\mathcal{T}[[ab]^*+(a+b)^*]] \neq \mathcal{T}[[a+b]^*]]$$

whereas

$$\mathcal{L}[[ab]^*+(a+b)^*]] = \mathcal{L}[[a+b]^*]]$$
One input string can be parsed in multiple ways: \texttt{ababab} under \( E = (ab)^* + (a + b)^* \) can be parsed \textit{both} as

\[
\begin{align*}
\text{inl} \ [(a, b), (a, b), (a, b)] \\
\text{and} \\
\text{inr} \ [(\text{inl} \ a, \text{inr} \ b, \text{inl} \ a, \text{inr} \ b, \text{inl} \ a, \text{inr} \ b]
\end{align*}
\]

\textit{Disambiguation policy}: the \textbf{left-most} option is always prioritized. “Greedy parsing.”
One input string can be parsed in multiple ways: \texttt{ababab}
under \( E = (ab)^* + (a + b)^* \) can be parsed \emph{both} as
\[
\begin{align*}
\text{inl } [(a, b), (a, b), (a, b)] \\
\text{and} \\
\text{inr } [\text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b]
\end{align*}
\]

\emph{Disambiguation policy}: the \textbf{left-most} option is always prioritized. \“Greedy parsing.\”
Bit-coded parse trees: only store *choices*. Parse tree as stream of bits; *meaningless* without expression!

**Example**

\[ E = (ab)^* + (a + b)^* \]

\[ \text{ababab}: \]

\[
\begin{array}{l}
\text{inl } [(a, b), (a, b), (a, b)] & \text{00001} \\
\text{inr } [\text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b] & \text{10001000100011}
\end{array}
\]
• Thompsons FSTs with input alphabet $\Sigma$, output alphabet \{0, 1\}.

• Construction:

<table>
<thead>
<tr>
<th>$E$</th>
<th>$\mathcal{N}(E, q^s, q^f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start $\xrightarrow{} q^s$ $q^f$</td>
</tr>
<tr>
<td>1</td>
<td>start $\xrightarrow{} q^s$ $(q^f = q^s)$</td>
</tr>
<tr>
<td>$a$</td>
<td>start $\xrightarrow{a/\epsilon} q^s$ $q^f$</td>
</tr>
</tbody>
</table>
### Finite State Transducers

<table>
<thead>
<tr>
<th>$E$</th>
<th>$\mathcal{N}(E, q^s, q^f)$</th>
</tr>
</thead>
</table>
| $E_1E_2$ | $\begin{array}{c}
\text{start} \rightarrow q^s \\
\mathcal{N}(E_1, q^s, q') \rightarrow q' \\
\mathcal{N}(E_2, q', q^f) \rightarrow q^f
\end{array}$ |
| $E_1 + E_2$ | $\begin{array}{c}
\text{start} \rightarrow q^s \\
\begin{array}{c}
\epsilon/0 \\
\epsilon/1
\end{array} \\
\mathcal{N}(E_1, q^s, q^f) \\
\mathcal{N}(E_2, q^f, q^f)
\end{array}$ |
| $E_0^*$ | $\begin{array}{c}
\text{start} \rightarrow q^s \\
\begin{array}{c}
\epsilon/0 \\
\epsilon/1
\end{array} \\
\mathcal{N}(E_0, q^s, q^f)
\end{array}$ |
Theorem (Brüggemann-Klein 1993, GHNR 2013)

1-to-1 correspondence between

- parse trees for $E$,  
- paths in Thompson FST for $E$,  
- bit-coded parse trees.

Constructing the parse tree corresponds to finding a path through the FST.
Optimally streaming parsing

Output the longest common prefix of possible parse trees after reading each input symbol.

Example

\[ E = (aaa + aa)^* \]

Possible parse tree prefixes after `aaaa`:

\[ \{01011, 000 \ldots \} \]

Possible parse tree prefixes after `aaaaa`:

\[ \{00011, 0000 \ldots \} \]
### Greedy Parsing

<table>
<thead>
<tr>
<th>Parse</th>
<th>Time</th>
<th>Space</th>
<th>Aux</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3-p)&lt;sup&gt;1&lt;/sup&gt;</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>(2-p)&lt;sup&gt;2&lt;/sup&gt;</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>(str.)&lt;sup&gt;3&lt;/sup&gt;</td>
<td>$O(mn + 2^m \log m)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
</tbody>
</table>

*(n size of input, m size of expression)*

<sup>1</sup>Frisch, Cardelli (2004)

<sup>2</sup>Grathwohl, Henglein, Nielsen, Rasmussen (2013)

<sup>3</sup>Grathwohl, Henglein, Rasmussen (2014)
## Greedy Parsing

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
<th>Aux</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse (3-p)&lt;sup&gt;1&lt;/sup&gt;</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>Parse (2-p)&lt;sup&gt;2&lt;/sup&gt;</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>Parse (str.)&lt;sup&gt;3&lt;/sup&gt;</td>
<td>$O(mn + 2^m \log m)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
</tbody>
</table>

$n$ size of input, $m$ size of expression

---

<sup>1</sup> Frisch, Cardelli (2004)  
<sup>2</sup> Grathwohl, Henglein, Nielsen, Rasmussen (2013)  
<sup>3</sup> Grathwohl, Henglein, Rasmussen (2014)
Optimally streaming algorithm

- Preprocessing step of FST: compute coverage of state sets.
- Maintain a path tree during FST simulation, recording the path taken to each state in the FST.
  - Prune states that are covered by higher-prioritized states.
- Output on the stem of the path tree is longest common prefix of any succeeding parse.

Theorem (GHR’14)

Optimally streaming algorithm computes the optimally streaming parsing function in time $O(mn + 2^m \log m)$. 
PATH TREE EXAMPLE: \((aaa + aa)^*\)
PATH TREE EXAMPLE: $(aaa + aa)^*$
PATH TREE EXAMPLE: \((aaa + aa)^*\)
PATH TREE EXAMPLE: \((aaa + aa)^*\)
PATH TREE EXAMPLE: \((aaa + aa)^*\)
PATH TREE EXAMPLE: \((aaa + aa)^*\)
PATH TREE EXAMPLE: \((aaa + aa)^*\)
**Observation**

Approach is not limited to Thompson FSTs outputting bit-coded parse trees.

Kleenex is a surface language for specifying FSTs and their output:

- *grammar* with greedy disambiguation;
- embedded *output actions*.
- Essentially optimally streaming behaviour.
- Linear running time in size of input string.
- *Fast*. >1 Gbps common.
Problem: need to read entire number; no bounded lookahead!

But: each newline ends a number, so output.

Optimal streaming gives this for free!
Path tree algorithm is “NFA simulation with path trees as state sets.”

Compilation of FSTs? Analogy to NFA-DFA determinization with subset construction?
Path tree algorithm is “NFA simulation with path trees as state sets.”

Compilation of FSTs? Analogy to NFA-DFA determinization with subset construction?

Problem: Inifinite number of path trees!
Path tree algorithm is “NFA simulation with path trees as state sets.”

Compilation of FSTs? Analogy to NFA-DFA determinization with subset construction?

Problem: Infinite number of path trees!

Solution: contract unary paths in path trees and store output in registers.
DETERMINIZATION
\[
\begin{align*}
  x_\epsilon & := 0 \\
  x_0 & := 00 \\
  x_1 & := 1011 \\
  x_{00} & := 0 \\
  x_{01} & := 1
\end{align*}
\]
Determinization

\[ x_\epsilon := 0 \]
\[ x_0 := 00 \]
\[ x_1 := 1011 \]
\[ x_{00} := 0 \]
\[ x_{01} := 1 \]

\[ x'_{\epsilon} := x_\epsilon \cdot x_0 \]
\[ x'_0 := x_{00} \]
\[ x'_1 := x_{01} \]
• Streaming string transducer:
  • deterministic finite automata,
  • each state equipped with fixed number of registers containing strings
  • registers updated on transition by affine function;
  • Alur, D’Antoni, Raghothaman (2015).
Streaming string transducer:
- deterministic finite automata,
- each state equipped with fixed number of registers containing strings
- registers updated on transition by affine function;

**Theorem**

FSTs with greedy order semantics correspond to SSTs.
- States are contracted path trees.
- Edges in contracted path trees $\cong$ registers in SST.
a/ $x_0 := (x_0)(x_{00})$
$a/ x_0, x_{00} := 0$
$a/ x_1, x_{01} := 1$

b/ $x_1 := (x_1)(x_{10})(x_{100})$
$b/ x_{00}, x_{100}, x_{10} := 0$
$b/ x_{01}, x_{101}, x_{11} := 1$

a/ $x_0 := (x_0)(x_{01})$
$a/ x_0, x_{00} := 0$
$a/ x_1, x_{01} := 1$

b/ $x_1 := (x_1)(x_{10})(x_{101})$
$b/ x_{10} := 0$
$b/ x_{11} := 1$
### Haskell implementation

<table>
<thead>
<tr>
<th>Kleenex source</th>
<th>→</th>
<th>FST</th>
<th>→</th>
<th>SST</th>
<th>→</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C code compiled with GCC/clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance comparison with regular expression libraries:

- AWK, Perl, Python, Sed, Tcl
- RE2/RE2j
- Ragel state machine compiler

https://github.com/diku-kmc/kleenexlang
PERFORMANCE

\[
\text{main} := (\text{row} /\backslash n/) *\\
\text{col} := /[^,\backslash n]/*\\
\text{row} := ~(\text{col} /,/) \text{ col } "\backslash t" /,/) ~(\text{col} /,/) ~(\text{col} /,/) \text{ col } /,/) ~\text{col}
\]
FUTURE WORK

• Program fragments as output actions
• Memoization techniques à la NFA/DFA memoization in RE2.
• Applications – bioinformatics, finance, log digging, ....;
• Parallel processing: read >8 bits in parallel;
• Approximate matching — necessary in biological applications;
• Expressiveness, visibly pushdown automata;
• Automatically generate interfaces for various programming languages.
Kleene algebra
Kleene algebra

A structure \((K, +, \cdot, *, 0, 1)\):

- A set of elements \(K\),
- binary operators \(+\) and \(\cdot\),
- unary operator \(*\),
- special elements 0 and 1,

that satisfies the *Kleene algebra axioms*. 
**Semiring**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot (y \cdot z) = (x \cdot y) \cdot z$</td>
<td>$x + (y + z) = (x + y) + z$</td>
</tr>
<tr>
<td>$1 \cdot x = x = x \cdot 1$</td>
<td>$0 + x = x = x + 0$</td>
</tr>
<tr>
<td></td>
<td>$x + y = y + x$</td>
</tr>
<tr>
<td></td>
<td>$x \cdot (y + z) = x \cdot y + x \cdot z$</td>
</tr>
<tr>
<td></td>
<td>$(x + y) \cdot z = x \cdot z + y \cdot z$</td>
</tr>
<tr>
<td></td>
<td>$0 \cdot x = 0 = x \cdot 0$</td>
</tr>
</tbody>
</table>

- idempotence: $x + x = x$
- partial order: $x \leq y \iff x + y = y$
Kleene algebra

Idempotent semiring with * operator:

\[ 1 + x \cdot x^* \leq x^* \]
\[ 1 + x^* \cdot x \leq x^* \]
\[ b + a \cdot x \leq x \implies a^* \cdot b \leq x \]
\[ b + x \cdot a \leq x \implies b \cdot a^* \leq x \]
Any structure with these operators that satisfies the axioms is a Kleene algebra.

- Languages: \((L, \cup, \cdot, *, \emptyset, \{\epsilon\})\).
  - \(L\) is set of strings over an alphabet,
  - \(\cup\) is set union,
  - \(\cdot\) is string concatenation,
  - \(*\) is repetition of strings,
  - partial order \(\leq\) is subset inclusion \(\subseteq\).

*Language interpretation* of regular expressions from before.

- Relation model, tropical semiring, ...
"Regular expressions:" syntax to describe elements in a Kleene algebra.

**Canonical interpretation**

The *canonical interpretation* of a term $E$ is the regular language interpretation:

- $L_{\Sigma}(x) = \{x\}$
- $L_{\Sigma}(e_0 + e_1) = L_{\Sigma}(e_0) \cup L_{\Sigma}(e_1)$
- $L_{\Sigma}(0) = \emptyset$
- $L_{\Sigma}(e_0 e_1) = \{vw \mid v \in L_{\Sigma}(e_0), \ w \in L_{\Sigma}(e_1)\}$
- $L_{\Sigma}(1) = \{\epsilon\}$
- $L_{\Sigma}(e^*) = \bigcup_{n \geq 0} L_{\Sigma}(e^n)$. 
### Polynomials

Given idempotent semiring $C$ and a set of variables $X$, form *polynomials* over $C$ and $X$:

$0, \ a, \ ax^2 + bxy^3 + 1, \ 1 + a + ax + by$

### System of polynomial inequalities

- $1 + aB + bA \leq S$
- $A + aS + bAA \leq A$
- $bS + aBB \leq B$
Solution: valuation of variables in $X$ such that the inequalities are satisfied.

A semiring $C$ is *algebraically closed* if all finite systems of polynomials have *least* solutions.

**Definition**

*A Chomsky algebra* is an algebraically closed idempotent semiring.
Context-free languages over symbols from $X$ are not Kleene algebras, but they are Chomsky algebras.

Context-free grammar corresponds to system of polynomial inequalities:

$$
S \rightarrow \epsilon | aB | bA \quad \text{1 + } aB + bA \leq S \\
A \rightarrow aS | bAA \quad \text{aS + bAA} \leq A \\
B \rightarrow bS | aBB \quad \text{bS + aBB} \leq B
$$
- Regular expressions: denote elements in Kleene algebra.
- $\mu$-terms: denote elements in Chomsky algebra.

### $\mu$-terms

$TX$ are $\mu$-terms over an alphabet $X$:

$$t ::= 0 \mid 1 \mid x \mid t + t \mid t \cdot t \mid \mu x.t \quad x \in X$$
### $n$-fold composition

<table>
<thead>
<tr>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \cdot x \cdot t \equiv 0$</td>
</tr>
<tr>
<td>$(n + 1) \cdot x \cdot t \equiv t[x/n \cdot x \cdot t]$</td>
</tr>
</tbody>
</table>

### Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \cdot a \cdot x \cdot b + 1 = 0$</td>
<td></td>
</tr>
<tr>
<td>$1 \cdot a \cdot x \cdot b + 1 = a(0 \cdot a \cdot x \cdot b + 1)b + 1 = a0b + 1 = 1$</td>
<td></td>
</tr>
<tr>
<td>$2 \cdot a \cdot x \cdot b + 1 = a(1 \cdot a \cdot x \cdot b + 1)b + 1 = ab + 1$</td>
<td></td>
</tr>
</tbody>
</table>
Given interpretation of literals: $\sigma : X \rightarrow C$, interpretation of $\mu$-terms over Chomsky algebra $C$.

Function $\sigma : TX \rightarrow C$ where:

\[
\begin{align*}
\sigma(0) &= 0 \\
\sigma(1) &= 1 \\
\sigma(a + b) &= \sigma(a) + \sigma(b) \\
\sigma(a \cdot b) &= \sigma(a) \cdot \sigma(b)
\end{align*}
\]

$\sigma(\mu x.t) = \text{least } a \in C \text{ such that } \sigma[x/a](t) \leq a$
### Canonical interpretation

Canonical interpretation as *context-free languages*:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_X(x) )</td>
<td>( { x } )</td>
</tr>
<tr>
<td>( L_X(0) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( L_X(1) )</td>
<td>( { \epsilon } )</td>
</tr>
<tr>
<td>( L_X(t_0 + t_1) )</td>
<td>( L_X(t_0) \cup L_X(t_1) )</td>
</tr>
<tr>
<td>( L_X(t_0 \cdot t_1) )</td>
<td>( { vw \mid v \in L_X(t_0), \ w \in L_X(t_1) } )</td>
</tr>
<tr>
<td>( L_X(\mu x.t) )</td>
<td>( \bigcup_{n \geq 0} L_X(nx.t) ).</td>
</tr>
</tbody>
</table>
\[ \sum_{n \geq 0} t_n \] denotes *supremum* with respect to partial order \( \leq \)

\[ \mu\text{-continuity} \]

A Chomsky algebra \( C \) is \( \mu \)-continuous if

\[ \sigma (a(\mu x.t)b) = \sum_{n \geq 0} \sigma (a(nx.t)n) \]

for any interpretation \( \sigma \) over \( C \).

Canonical interpretation as context-free language is \( \mu \)-continuous:

\[ L_X(\mu x.t) = \bigcup_{n \geq 0} L_X(nx.t). \]
Theorem

The following are equivalent:

(i) $s = t$ holds in all $\mu$-continious Chomsky algebras,
(ii) $L_X(s) = L_X(t)$ holds in the canonical interpretation as a context-free language over variables $X$. 
Two context free languages $L_X(s)$ and $L_X(t)$ are equivalent if and only if $s = t$ is provable from the axioms of $\mu$-continuous Chomsky algebra:

**Axioms**

\[
\begin{align*}
\cdot \cdot \cdot \cdot 
\end{align*}
\]

\[
\begin{align*}
x \cdot (y \cdot z) &= (x \cdot y) \cdot z & x + (y + z) &= (x + y) + z \\
1 \cdot x &= x &= x \cdot 1 & 0 + x &= x &= x + 0 \\
x \cdot (y + z) &= x \cdot y + x \cdot z & x + y &= y + x \\
(x + y) \cdot z &= x \cdot z + y \cdot z & x + x &= x \\
0 \cdot x &= 0 &= x \cdot 0
\end{align*}
\]

\[
a(\mu x.t)b = \sum_{n \geq 0} a(nx.t)b
\]
μ-continuity axiom is **infinitary**:

\[ a(nx.t)b \leq a(\mu x.t)b, \quad n \geq 0 \]

\[
\left( \bigwedge_{n \geq 0} (a(nx.t)b \leq w) \right) \implies a(\mu x.t)b \leq w
\]

- Equivalence of context-free languages is **undecidable**.
- To use inference, one must establish infinitely many premises.
• Extend Chomsky algebra with test symbols, analogously to Kleene algebra with tests.
• Coalgebraic treatment of Chomsky algebra?
• Applications to program verification, like Kleene algebra?
• “Visibly pushdown” Chomsky algebra?
• KAT+B! is an extension to Kleene algebra with tests adding mutable state:
  • elements correspond to square matrices with regular language entries.
  • extend Kleenex with mutable state?
Thank you