Tidyparse: Real-Time Context-free Error Correction

Breandan Considine, Jin Guo, Xujie Si

McGill University, Mila IQIA

bre@ndan.co

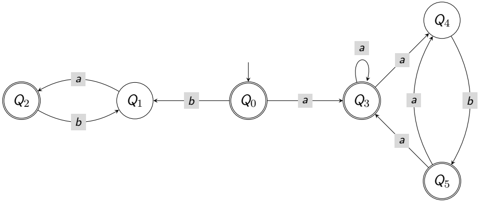
November 26, 2022

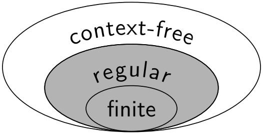
Overview

Background: Regular grammars

A regular grammar (RG) is a quadruple $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ where V are nonterminals, Σ are terminals, $P : V \times (V \cup \Sigma)^{\leq 2}$ are the productions, and $S \in V$ is the start symbol, i.e., all productions are of the form $A \to a$, $A \to aB$ (right-regular), or $A \to Ba$ (left-regular). E.g., the following RG and NFA correspond to the language defined by the *regex* `(a(ab)*)*(ba)*`:

$$
\begin{aligned}
S &\to Q_0 \mid Q_2 \mid Q_3 \mid Q_5 \\
Q_0 &\to \varepsilon \\
Q_1 &\to Q_0 b \mid Q_2 b \\
Q_2 &\to Q_1 a \\
Q_3 &\to Q_0 a \mid Q_3 a \mid Q_5 a \\
Q_4 &\to Q_3 a \mid Q_5 a \\
Q_5 &\to Q_4 b
\end{aligned}
$$



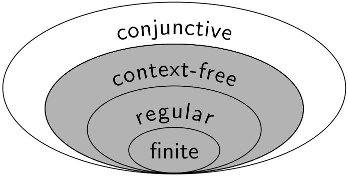


Background: Context-free grammars

In a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ all productions are of the form $P : V \times (V \cup \Sigma)^{+}$, i.e., RHS may contain any number of nonterminals, V. Recognition decidable in n^{ω}, n.b. CFLs are **not** closed under intersection!

For example, consider the grammar $\underline{S \to SS \mid (S) \mid ()}$. This represents the language of balanced parentheses, e.g. $(), ()(), (()), ()(()), (()()), (())() \ldots$

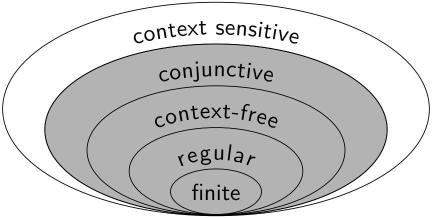
Every CFG has a normal form $P^{*} : V \times (V^{2} \mid \Sigma)$, i.e., every production can be refactored into either $v_0 \to v_1 v_2$ or $v_0 \to \sigma$, where $v_{0\ldots2} : V$ and $\sigma : \Sigma$, e.g., $\{S \to SS \mid (S) \mid ()\} \Leftrightarrow^{*} \{S \to XR \mid SS \mid LR, L \to (, R \to), X \to LS\}$



Background: Conjunctive grammars

Conjunctive grammars naturally extend CFGs with CFL union and intersection, respecting closure under those operations. Equivalent to trellis automata, which are like contractive elementary cellular automata. Language inclusion is decidable in P.

$$\frac{\Gamma \vdash \mathcal{G}_1, \mathcal{G}_2 : \texttt{CG}}{\Gamma \vdash \exists\, \mathcal{G}_3 : \texttt{CG} \,.\, \mathcal{L}_{\mathcal{G}_1} \cap \mathcal{L}_{\mathcal{G}_1} \leftrightarrow \mathcal{G}_3} \cap$$



Background: Closure properties of formal languages

Formal languages are not always closed under set-theoretic operations, e.g., CFL $\cap$ CFL is not CFL in general. Let $\cdot$ denote concatenation, $\star$ be Kleene star, and $\complement$ be complementation:

	$\cup$	$\cap$	$\cdot$	$\star$	$\complement$
Finite[1]	✓	✓	✓	✓	✓
Regular[1]	✓	✓	✓	✓	✓
Context-free[1]	✓	✗	✓	✓	✗
Conjunctive[1,2]	✓	✓	✓	✓	?
Context-sensitive[2]	✓	✓	✓	+	✓
Recursively Enumerable[2]	✓	✓	✓	✓	✗

We would like a language family that is (1) tractable, i.e., has polynomial recognition and search complexity and (2) reasonably expressive, i.e., can represent syntactic properties of real-world programming languages.

Context-free parsing, distilled

Given a CFG $\mathcal{G} := \langle V, \Sigma, P, S \rangle$ in Chomsky Normal Form, we can construct a recognizer $R_{\mathcal{G}} : \Sigma^n \to \mathbb{B}$ for strings $\sigma : \Sigma^n$ as follows. Let 2^V be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as follows:

$$s_1 \otimes s_2 := \{C \mid \langle A, B \rangle \in s_1 \times s_2, (C \to AB) \in P\}$$

e.g., $\{\mathrm{A} \to \mathrm{BC},\ \mathrm{C} \to \mathrm{AD},\ \mathrm{D} \to \mathrm{BA}\} \subseteq \mathrm{P} \vdash \{\mathrm{A, B, C}\} \otimes \{\mathrm{B, C, D}\} = \{\mathrm{A, C}\}$

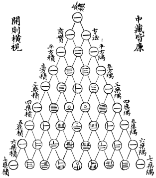
If we define $\sigma_r^{\nrightarrow} := \{w \mid (w \to \sigma_r) \in P\}$, then initialize $M^0_{r+1=c}(\mathcal{G}', e) := \sigma_r^{\nrightarrow}$ and solve for the fixpoint $M^* = M + M^2$,

$$M^0 := \begin{pmatrix} \varnothing & \sigma_1^{\rightarrow} & \varnothing & \cdots & \varnothing \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \varnothing \\ \vdots & & & & \sigma_n^{\uparrow} \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix} \Rightarrow \ldots \Rightarrow M^* = \begin{pmatrix} \varnothing & \sigma_1^{\rightarrow} & \Lambda & \cdots & \Lambda_\sigma^* \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \Lambda \\ \vdots & & & & \sigma_n^{\uparrow} \\ \varnothing & \cdots & \cdots & \cdots & \varnothing \end{pmatrix}$$

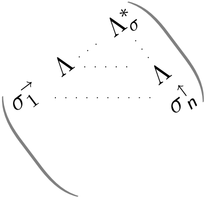
Valiant (1975) shows that $\sigma \in \mathcal{L}_{\mathcal{G}}$ iff $S \in \mathcal{T}$, i.e., $\mathbb{1}_{\mathcal{T}}(S) \iff \mathbb{1}_{\mathcal{L}_{\mathcal{G}}}(\sigma)$.

Lattices, Matrices and Trellises

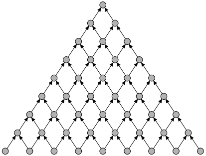
The art of treillage has been practiced from ancient through modern times.



Jia Xian Triangle
Jia, ~1030 A.D.



CYK Parsing
Sakai, 1961 A.D.

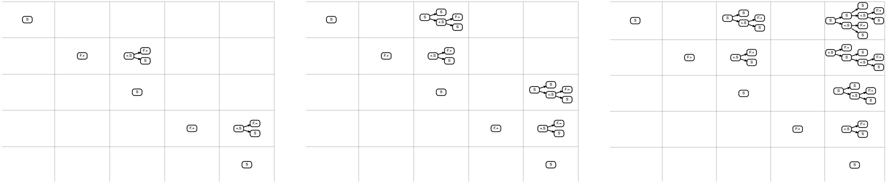


Trellis Automaton
Dyer, 1980 A.D.

A few observations on algebraic parsing

- The matrix $\mathbf{M}^*$ is strictly upper triangular, i.e., nilpotent of degree n
- Recognizer can be translated into a parser by storing backpointers

$\mathbf{M}_1 = \mathbf{M}_0 + \mathbf{M}_0^2$ $\quad$ $\mathbf{M}_2 = \mathbf{M}_1 + \mathbf{M}_1^2$ $\quad$ $\mathbf{M}_3 = \mathbf{M}_2 + \mathbf{M}_2^2 = \mathbf{M}_4$



- The $\otimes$ operator is *not* associative: $S \otimes (S \otimes S) \neq (S \otimes S) \otimes S$
- Built-in error recovery: nonempty submatrices $=$ parsable fragments
- `seekFixpoint { it + it * it }` is sufficient but unnecessary
- If we had a way to solve for $\mathbf{M} = \mathbf{M} + \mathbf{M}^2$ directly, power iteration would be unnecessary, could solve for $\mathbf{M} = \mathbf{M}^2$ above superdiagonal

Satisfiability + holes (our contribution)

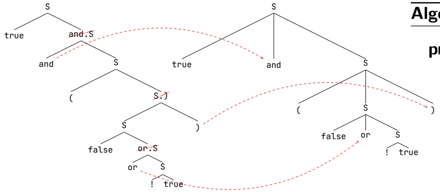
- Can be lowered onto a Boolean tensor $\mathbb{B}_2^{n\times n\times|V|}$ (Valiant, 1975)
- Binarized CYK parser can be efficiently compiled to a SAT solver
- Enables sketch-based synthesis in either σ or $\mathcal{G}$: just use variables!
- We simply encode the characteristic function, i.e. $\mathbb{1}_{\subseteq V} : V \to \mathbb{Z}_2^{|V|}$
- $\oplus, \otimes$ are defined as $\boxplus, \boxtimes$, so that the following diagram commutes:

$$\begin{tikzcd}
2^V \times 2^V \arrow[r, "\oplus/\otimes"] \arrow[d, shift left, "\mathbb{1}^2"] & 2^V \arrow[d, shift left, "\mathbb{1}"] \\
\mathbb{Z}_2^{|V|} \times \mathbb{Z}_2^{|V|} \arrow[r, "\boxplus/\boxtimes"'] \arrow[u, shift left, "\mathbb{1}^{-2}"] & \mathbb{Z}_2^{|V|} \arrow[u, shift left, "\mathbb{1}^{-1}"]
\end{tikzcd}$$

- These operators can be lifted into matrices/tensors in the usual way
- In most cases, only a few nonterminals are active at any given time
- More sophisticated representations are known for $\binom{n}{0\leq k}$ subsets
- If density is desired, possible to use the Maculay representation
- If you know of a more efficient encoding, please let us know!

Chomsky Denormalization

Chomksy normalization is needed for matrix-based parsing, however produces lopsided parse trees. We can denormalize them using a simple recusive procedure to restore the natural shape of the original CFG:



Pre-Denormalization Post-Denormalization

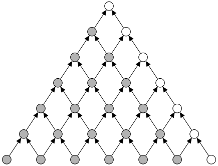
Algorithm Rewrite procedure for tree denormalization

procedure DENORMALIZE(`t: Tree`)
  `stems` $\leftarrow \{$ DENORMALIZE(`c`) $\mid$ `c` $\in$ `t.children` $\}$
  if `t.root` $\in V_{\mathcal{G}'} \setminus V_{\mathcal{G}}$ **then**
    return `stems` ▷ Drop synthetic nonterminals.
  else ▷ Graft the denormalized children on root.
    return $\{$ `Tree(root, stems)` $\}$
  end if
end procedure

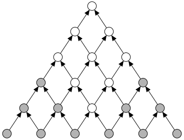
All synthetic nonterminals are excised during Chomsky denormalization. Rewriting improves legibility but does not alter the underlying semantics.

Incremental parsing

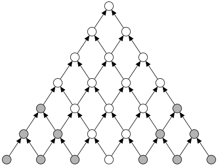
Should only need to recompute submatrices affected by individual edits. In the worst case, each edit requires quadratic complexity in terms of $|\Sigma^*|$, assuming $\mathcal{O}(1)$ cost for each CNF-nonterminal subset join, $\mathbf{V}_1' \otimes \mathbf{V}_2'$.



Append
$\mathcal{O}(n+1)$



Delete
$\mathcal{O}\left(\frac{1}{4}(n-1)^2\right)$



Insert
$\mathcal{O}\left(\frac{1}{4}(n+1)^2\right)$

Related to **dynamic matrix inverse** and **incremental transitive closure** with vertex updates. With a careful encoding, we can incrementally update SAT constraints as new keystrokes are received to eliminate redundancy.

Conjunctive parsing

It is well-known that the family of CFLs is not closed under intersection. For example, consider $\mathcal{L}_\cap := \mathcal{L}_{\mathcal{G}_1} \cap \mathcal{L}_{\mathcal{G}_2}$:

$$P_1 := \{\ S \rightarrow LR, \quad L \rightarrow ab \mid aLb, \quad R \rightarrow c \mid cR\ \}$$
$$P_2 := \{\ S \rightarrow LR, \quad R \rightarrow bc \mid bRc, \quad L \rightarrow a \mid aL\ \}$$

Note that $\mathcal{L}_\cap$ generates the language $\{\ a^d b^d c^d \mid d > 0\ \}$, which according to the pumping lemma is not context-free. To encode $\mathcal{L}_\cap$, we intersect all terminals $\Sigma_\cap := \bigcap_{i=1}^{c} \Sigma_i$, then for each $t_\cap \in \Sigma_\cap$ and CFG, construct an equivalence class $E(t_\cap, \mathcal{G}_i) = \{w_i \mid (w_i \rightarrow t_\cap) \in P_i\}$ as follows:

$$\bigwedge_{t \in \Sigma_\cap} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_\cap, \mathcal{G}_j) \equiv_{\sigma_i} E(t_\cap, \mathcal{G}_{j+1}) \tag{1}$$

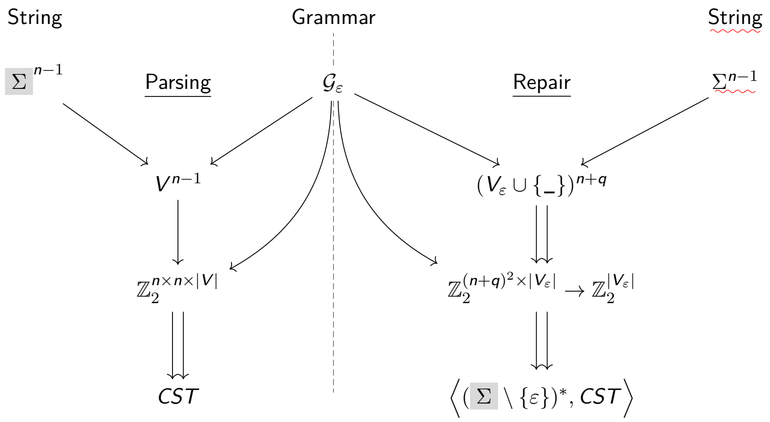
A birds eye view of the algorithm

We can lower Valiant's algorithm onto a polynomial system of equations over finite fields, allowing us to solve for holes and parse trees.

$$\begin{tikzcd}
\langle \mathcal{G}', \Sigma^{n-1} \rangle \arrow[dr, leftrightarrow] & \text{Set} & \text{Bit} & \langle \mathcal{G}', \Sigma^{n-1} \rangle \arrow[dr] & \text{SAT} \\
\text{Rubix} & M \times M \arrow[r, leftrightarrow, "\mathbb{1}^{2^{n\times n}}", "\mathbb{1}^{-2^{n\times n}}"'] \arrow[d, "\oplus"', "\otimes"] & \mathbb{Z}_2^{|V|^{n\times n}} \times \mathbb{Z}_2^{|V|^{n\times n}} \arrow[rr, tail, "\varphi^{2^{n\times n}}", "\varphi^{-2^{n\times n}}"'] \arrow[d, "+"', "*"] & & \mathcal{M} \times \mathcal{M} \arrow[d, leftrightarrow, "+"', "*"] \\
\text{Matrix} & 2^V \times 2^V \arrow[r, leftrightarrow, "\mathbb{1}^2", "\mathbb{1}^{-2}"'] \arrow[d, "\oplus"', "\otimes"] & \mathbb{Z}_2^{|V|} \times \mathbb{Z}_2^{|V|} \arrow[rr, tail, "\varphi^2", "\varphi^{-2}"'] \arrow[d, "\boxplus"', "\boxtimes"] & & \mathcal{V} \times \mathcal{V} \arrow[d, leftrightarrow, "\boxplus"', "\boxtimes"] \\
\text{Vector} & 2^V \arrow[r, leftrightarrow, "\mathbb{1}", "\mathbb{1}^{-1}"'] & \mathbb{Z}_2^{|V|} \arrow[rr, tail, "\varphi", "\varphi^{-1}"'] & & \mathcal{V}
\end{tikzcd}$$

So far, we only consider Cartesian closed categories, however, we can also consider other categories, such as the category of CFLs under conjunction, which allows us to encode the intersection of two CFGs.

A birds eye view of the algorithm



Our algorithm produces set of concrete syntax trees (CSTs) for a given valid string. Otherwise, if the string contains an error, the algorithm generates a set of admissible corrections, alongside their CSTs.

Error Correction: Levenshtein q-Balls

Now that we have a reliable method to fix *localized* errors, $S : \mathcal{G} \times (\Sigma \cup \{\varepsilon, \_\})^n \rightarrow \{\Sigma^n\} \subseteq \mathcal{L}_\mathcal{G}$, given some unparseable string, i.e., $\underset{\sim}{\sigma_1 \ldots \sigma_n} : \boxed{\Sigma}^n \cap \mathcal{L}_\mathcal{G}^\complement$, where should we put holes to obtain a parseable $\sigma' \in \mathcal{L}_\mathcal{G}$? One way to do so is by sampling repairs, $\boldsymbol{\sigma} \sim \Sigma^{n \pm q} \cap \Delta_q(\underset{\sim}{\sigma})$ from the Levenshtein q-ball centered on $\underset{\sim}{\sigma}$, i.e., the space of all admissible edits with Levenshtein distance $\leq q$ (this is loosely analogous to a finite difference approximation). To admit variable-length edits, we first add an ε^+-production to each unit production:

$$\frac{\mathcal{G} \vdash \varepsilon \in \Sigma}{\mathcal{G} \vdash (\varepsilon^+ \rightarrow \varepsilon \mid \varepsilon^+\,\varepsilon^+) \in P}\;\varepsilon\textsc{-dup}$$

$$\frac{\mathcal{G} \vdash (A \rightarrow B) \in P}{\mathcal{G} \vdash (A \rightarrow B\,\varepsilon^+ \mid \varepsilon^+\,B \mid B) \in P}\;\varepsilon^+\textsc{-int}$$

Error Correction: d-Subset Sampling

Next, suppose $U : \mathbb{Z}_2^{m\times m}$ is a matrix whose structure is shown in Eq. 2, wherein C is a primitive polynomial over $\mathbb{Z}_2^m$ with coefficients $C_{1\ldots m}$ and semiring operators $\oplus := \veebar, \otimes := \wedge$:

$$U^t V = \begin{pmatrix} C_1 & \cdots & \cdots & \cdots & C_m \\ \top & \circ & \cdots & \cdots & \circ \\ \circ & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \circ & \cdots & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ \vdots \\ \vdots \\ \vdots \\ V_m \end{pmatrix} \tag{2}$$

Since C is primitive, the sequence $\mathbf{S} = (U^{0\ldots 2^m-1}V)$ must have *full periodicity*, i.e., for all $i, j \in [0, 2^m)$, $\mathbf{S}_i = \mathbf{S}_j \Rightarrow i = j$. To uniformly sample $\boldsymbol{\sigma}$ without replacement, we first form an injection $\mathbb{Z}_2^m \rightharpoonup \left\{{n \atop d}\right\} \times \Sigma_\varepsilon^{2d}$ using a combinatorial number system, cycle over $\mathbf{S}$, then discard samples which have no witness in $\left\{{n \atop d}\right\} \times \Sigma_\varepsilon^{2d}$. This method requires $\widetilde{\mathcal{O}}(1)$ per sample and $\widetilde{\mathcal{O}}\left(\binom{n}{d}|\Sigma + 1|^{2d}\right)$ to exhaustively search $\left\{{n \atop d}\right\} \times \Sigma_\varepsilon^{2d}$.

Error Correction: Sketch Templates

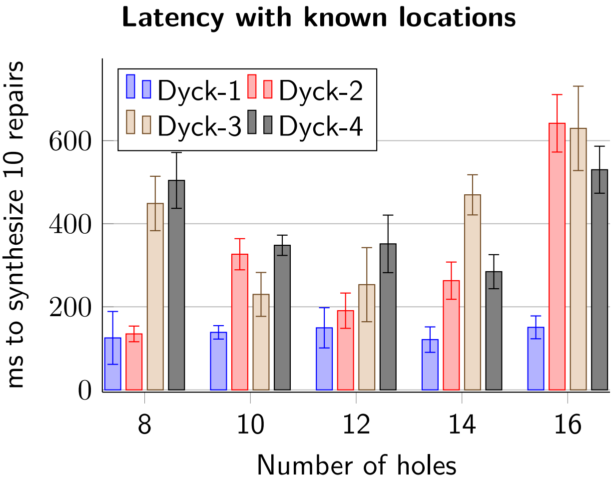
Finally, to sample $\boldsymbol{\sigma} \sim \Delta_q(\underset{\sim}{\sigma})$, we enumerate a series of sketch templates $H(\sigma, i) = \sigma_{1\ldots i-1}\ \_\ \_\ \sigma_{i+1\ldots n}$ for each $i \in \cdot \in \left\{ {n \atop d} \right\}$ and $d \in 1 \ldots q$, then solve for $\mathcal{M}^*_{\boldsymbol{\sigma}}$. If $S \in \Lambda^*_{\boldsymbol{\sigma}}?$ has a solution, each edit in each $\sigma' \in \boldsymbol{\sigma}$ will match exactly one of the following seven edit patterns:

$$\text{Deletion} = \left\{ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots \quad \gamma_{1,2} = \varepsilon \right.$$

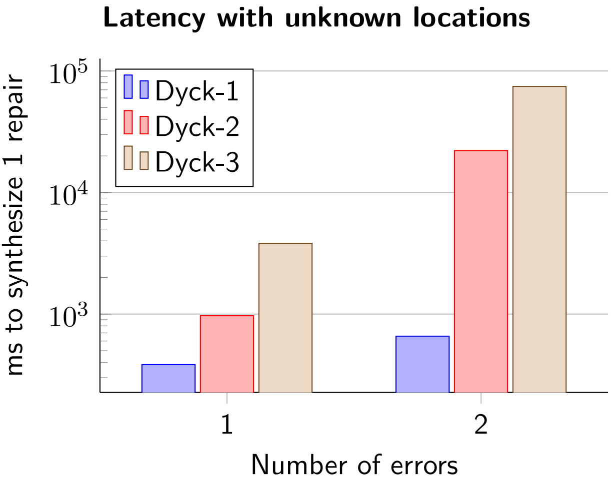
$$\text{Substitution} = \begin{cases} \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 \neq \varepsilon \wedge \gamma_2 = \varepsilon \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 = \varepsilon \wedge \gamma_2 \neq \varepsilon \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \{\gamma_1, \gamma_2\} \cap \{\varepsilon, \sigma_i\} = \varnothing \end{cases}$$

$$\text{Insertion} = \begin{cases} \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 = \sigma_i \wedge \gamma_2 \notin \{\varepsilon, \sigma_i\} \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_1 \notin \{\varepsilon, \sigma_i\} \wedge \gamma_2 = \sigma_i \\ \ldots \sigma_{i-1}\ \gamma_1\ \gamma_2\ \sigma_{i+1} \ldots & \gamma_{1,2} = \sigma_i \end{cases}$$

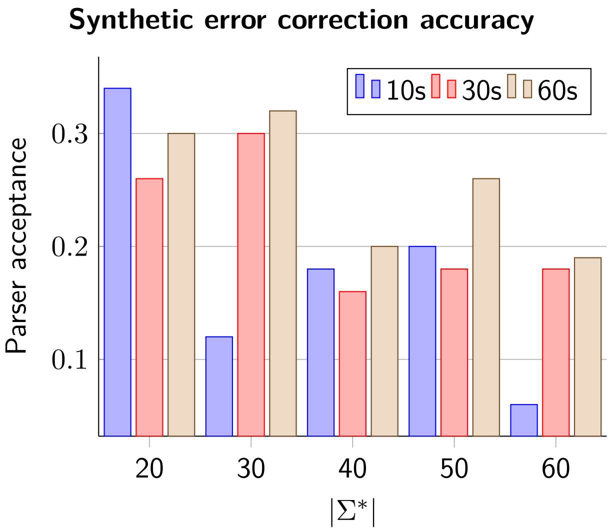
Level I: Known Error Locations



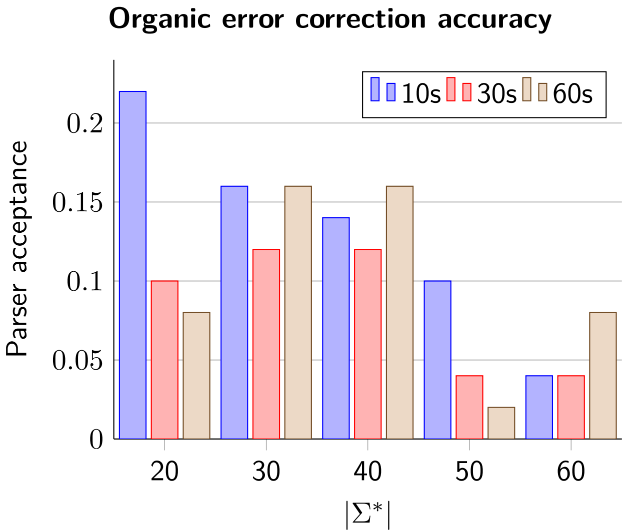
Level II: Unknown Locations, Fixed Error Count



Level III: MiniGithub, Unknown Count, Synthetic Errors



Level IV: BIFI Dataset, Unknown Count, Real Errors

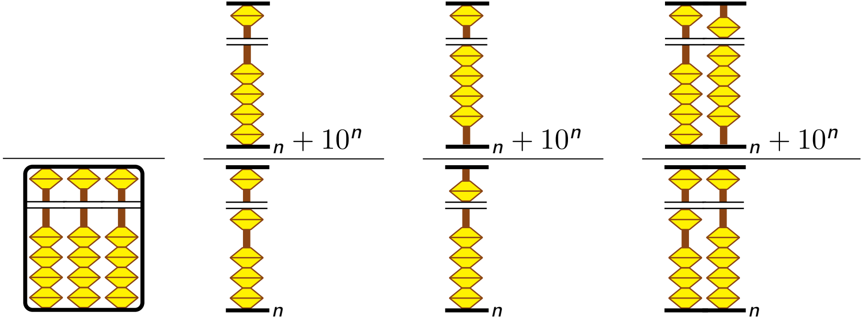


Abbreviated history of algebraic parsing

- Chomsky & Schützenberger (1959) - The algebraic theory of CFLs
- Cocke–Younger–Kasami (1961) - Bottom-up matrix-based parsing
- Brzozowski (1964) - Derivatives of regular expressions
- Earley (1968) - top-down dynamic programming (no CNF needed)
- Valiant (1975) - first realizes the Boolean matrix correspondence
  - Naïvely, has complexity $\mathcal{O}(n^4)$, can be reduced to $\mathcal{O}(n^\omega)$, $\omega < 2.763$
- Lee (1997) - Fast CFG Parsing $\iff$ Fast BMM, formalizes reduction
- Might et al. (2011) - Parsing with derivatives (Brzozowski $\Rightarrow$ CFL)
- Bakinova, Okhotin et al. (2010) - Formal languages over GF(2)
- Bernady & Jansson (2015) - Certifies Valiant (1975) in Agda
- Cohen & Gildea (2016) - Generalizes Valiant (1975) to parse and recognize mildly context sensitive languages, e.g. LCFRS, TAG, CCG
- **Considine, Guo & Si (2022) - SAT + Valiant (1975) + holes**

Abacus arithmetic

- Computational complexity of arithmetic is notation-dependent(!)
- For example, $\pm$ in unary arithmetic is concatenation and decatenation
- Multiplication and division by natural powers of the radix is $\mathcal{O}(1)$
- We can describe the abacus as a kind of abstract rewriting system



Abacus dependent types

```
sealed class B<X, P : B<X, P>>(open val x: X? = null) {
  val T: T<P> get() = T(this as P)
  val F: F<P> get() = F(this as P)
}

class U(val i: Int) : B<Any, U>() // Checked at runtime

object Ø: B<Ø, Ø>(null) // Denotes the end of a bitlist

class T<X>(override val x: X = Ø as X) : B<X, T<X>>(x)
  { companion object: T<Ø>(Ø) }

class F<X>(override val x: X = Ø as X) : B<X, F<X>>(x)
  { companion object: F<Ø>(Ø) }

val b0: F<Ø> = F
val b1: T<Ø> = T
val b2: F<T<Ø>> = T.F // Note the raw type is reversed
val b4: F<F<T<Ø>>> = T.F.F
```

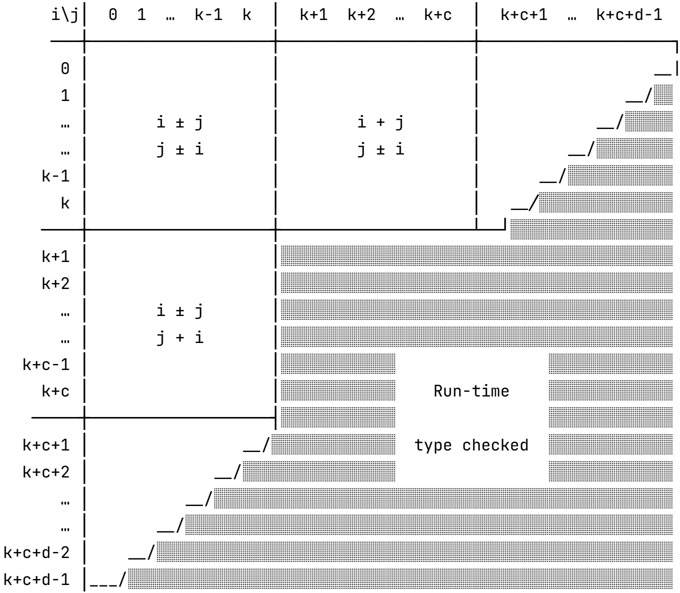
Abacus dependent types

```
typealias B_0<K> = F<K> // Type synonyms for legibility
typealias B_1<K> = T<K>
typealias B_2<K> = F<T<K>>
typealias B_3<K> = T<T<K>>
typealias B_4<K> = F<F<T<K>>>
typealias B_7<K> = T<T<T<K>>>
typealias B_8<K> = F<F<F<T<K>>>>

// Calculates k + 1 for all k == 2ⁿ - 1, 0 ≤ n < 4
operator fun Ø.plus(t: T<Ø>) = b1
operator fun B_0<Ø>.plus(t: T<Ø>) = b1
operator fun B_1<Ø>.plus(t: T<Ø>): B_2<Ø> = F(x + b1)
operator fun B_3<Ø>.plus(t: T<Ø>): B_4<Ø> = F(x + b1)
operator fun B_7<Ø>.plus(t: T<Ø>): B_8<Ø> = F(x + b1)

// Calculates k + 1 for all k ≡ 2ⁿ - 1 (mod 2ⁿ⁺¹), 1 ≤ n < 4
operator fun <K: B<*, *>> B_0<K>.plus(t: T<Ø>) = T(x)
operator fun <K: B<*, *>> B_1<F<K>>.plus(t: T<Ø>) = F(x + b1)
operator fun <K: B<*, *>> B_3<F<K>>.plus(t: T<Ø>) = F(x + b1)
operator fun <K: B<*, *>> B_7<F<K>>.plus(t: T<Ø>) = F(x + b1)
```

Abacus dependent types: birds eye view



Annotated history of typed eDSLs

- Canning et al. (1989) - F-Bounded Polymorphism is first invented
- Cheney & Hinze (2003) - Phantom types (good for type-safe builders)
- Meijer et al. (2006) - Language integrated querying (LINQ)
- Eder (2011) - Commercial reimplementation LINQ in Java/jOOQ
- Grigore (2016) - Java Generics shown to be Turing Complete
- Erdős (2017) - Encodes Boolean logic into Java type system
- Nakamaru et al. (2017) - Silverchain: a fluent API generator
- **Considine (2019) - Shape-safe matrix multiplication in Kotlin∇**
- Gil & Roth (2019) - Fling, a fluent API parser generator
- Cheng (2020) - Automatic theorem proving in the Scala type system
- Roth (2021) - Encodes CFL into Nominal Subtyping with Variance
- **Considine (2021) - Arithmetic in Kotlin via typelevel abacus**
- We know how to lower parsing onto types, what about vis versa?

Can we lower type checking onto parsing?

First, let us consider the untyped version:

```
Exp -> 0 | 1 | ... | T | F
Exp -> Exp Op Exp | if ( Exp ) Exp else Exp
Op -> and | or | + | *
```

Now, let us consider the GADT/HOAS version:

```
Exp<Bool> -> T | F
Op<Bool> -> and | or
Exp<Int> -> 0 | 1 | ... | 9
Op<Int> -> + | *
Exp<E> -> Exp<E> Op<E> Exp<E> // Es must be exactly the same!
Exp<E> -> if ( Exp<Bool> ) Exp<E> else Exp<E>
```

We can eliminate contextuality by concretizing over `E -> Bool | Int`:

```
Exp<Bool> -> T | F
Exp<Bool> -> Exp<Bool> or Exp<Bool> | Exp<Bool> and Exp<Bool>
Exp<Bool> -> if ( Exp<Bool> ) Exp<Bool> else Exp<Bool>
Exp<Int> -> 0 | 1 | ... | 9
Exp<Int> -> Exp<Int> + Exp<Int> | Exp<Int> * Exp<Int>
Exp<Int> -> if ( Exp<Bool> ) Exp<Int> else Exp<Int>
```

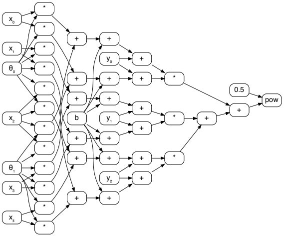
Classical programs are graphs

Programs can be compiled into DFGs and represented using a big matrix.

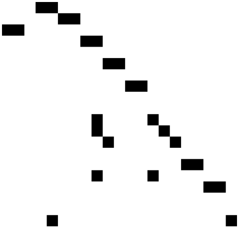
Program

```
sum = 0
l = [0, 0, 0, 0]
for i in range(0, 4):
 l[i] += θ[i] * x[i]
for i in range(0, 4):
 l[i] -= y[i] - b
for i in range(0, 4):
 l[i] *= l[i]
for i in range(0, 4):
 sum += l[i]
l = sqrt(sum)
```

Dataflow Graph



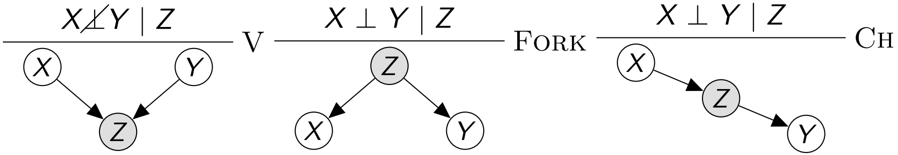
Matrix



This representation allows us to solve for their fixedpoints as eigenvectors.

Probabilistic programs are also graphs

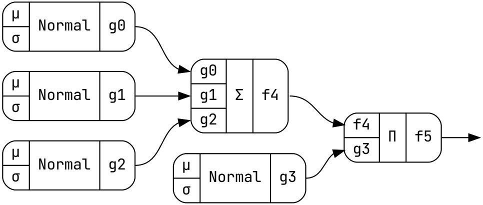
A Bayesian Belief Network (BN) is an acyclic DGM of the following form:



$$P(x_1, \ldots, x_D) = \prod_{i=1}^{D} P(x_i \mid \texttt{parents}(x_i))$$

Translatable to a probabilistic circuit a.k.a. Sum Product Network (SPN):

$$PC \to v \sim \mathcal{D}$$
$$PC \to PC \oplus PC$$
$$PC \to PC \otimes PC$$



Message passing & path algebras

A semiring algebra, denoted $(S, \oplus, \otimes, ⓪, ①)$, is a set together with two binary operators $\oplus, \otimes : S \times S \to S$ such that $(S, \oplus, ⓪)$ is a commutative monoid and $(S, \otimes, ①)$ is a monoid. Furthermore, we have distributivity:

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c}\textsc{Assoc} \qquad \frac{a \bullet ①}{a}\textsc{Neutral} \qquad \frac{a \bullet b}{b \bullet a}\textsc{Comm}$$

$$\frac{(a \oplus b) \otimes c}{(a \otimes c) \oplus (b \otimes c)}\textsc{Dist} \qquad \frac{a \otimes ⓪}{⓪}\textsc{Annhil}$$

These operators can be lifted to matrices to form *path algebras*:

$$\delta_{st} = \overbrace{\bigoplus_{P \in P^*_{st}} \underbrace{\bigotimes_{e \in P}}_{\text{Aggregate}} W_e}^{\text{Update}}$$

$\oplus$	$\otimes$	⓪	①	Path
min	$+$	∞	0	Shortest
max	$+$	$-\infty$	0	Longest
max	min	0	∞	Widest
$\veebar$	$\wedge$	$\circ$	$\top$	Random

Recap: Classical logic in a nutshell

$$\frac{a \veebar b}{(p \vee q) \wedge \neg(p \wedge q)}\,\text{XOR} \qquad \frac{a \rightarrow b}{\neg a \vee b}\,\text{Impl}$$

$$\frac{a \leftrightarrow b}{(\neg a \vee b) \wedge (\neg b \vee a)}\,\text{Iff}$$

$$\frac{\neg\neg a}{a}\,\text{2Neg} \qquad \frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c}\,\text{Assoc}_{\wedge\vee} \qquad \frac{a \bullet b}{b \bullet a}\,\text{Comm}_{\wedge\vee}$$

$$\frac{a \wedge (b \vee c)}{(a \wedge b) \vee (a \wedge c)}\,\text{Dist}_{\wedge} \qquad \frac{a \vee (b \wedge c)}{(a \vee b) \wedge (a \vee c)}\,\text{Dist}_{\vee}$$

$$\frac{\neg(a \vee b)}{\neg a \wedge \neg b}\,\text{DeMorgan}_{\vee} \qquad \frac{\neg(a \wedge b)}{\neg a \vee \neg b}\,\text{DeMorgan}_{\wedge}$$

Normalization in classical logic

Conjunctive Normal Form

$$\textsc{Conj} \to (\textsc{Disj}) \mid \textsc{Conj} \land (\textsc{Disj})$$
$$\textsc{Unit} \to \textsc{Var} \mid \neg\textsc{Var} \mid \bot \mid \top$$
$$\textsc{Disj} \to \textsc{Unit} \mid \textsc{Disj} \lor \textsc{Disj}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\neg(x \lor \neg y) \lor \neg\neg z}{\neg(x \lor \neg y) \lor z}\text{2Neg}}{(\neg x \land \neg\neg y) \lor z}\text{DeMorgan}}{(\neg x \land y) \lor z}\text{2Neg}}{(\neg x \lor z) \land (y \lor z)}\text{Dist}$$

Zhegalkin Normal Form

$$f(x_1, \ldots x_n) = \bigoplus_{i \subseteq \{1,\ldots,n\}} a_i x^i$$

i.e., a_i's filter the powerset.

$$\cfrac{\cfrac{\cfrac{\cfrac{x + (y \land \neg z)}{x + y(1 \oplus z)}}{x + (y \oplus yz)}}{x \oplus (y \oplus yz) \oplus x(y \oplus yz)}}{x \oplus y \oplus xy \oplus yz \oplus xyz}$$

Some common algebraic and logical forms

a_1	a_2	a_3	a_4	ZNF	Logical	CNF
0	0	0	0	0	$\bot$	$x \land \neg x$
1	0	0	0	1	$\top$	$x \lor \neg x$
0	1	0	0	x	x	x
1	1	0	0	$1 + \mathrm{x}$	$\neg x$	$\neg x$
0	0	1	0	y	y	y
1	0	1	0	$1 + \mathrm{y}$	$\neg y$	$\neg y$
0	1	1	0	$\mathrm{x} + \mathrm{y}$	$x \oplus y$	$(x \lor y) \land (\neg x \lor \neg y)$
1	1	1	0	$1 + \mathrm{x} + \mathrm{y}$	$x \iff y$	$(x \lor \neg y) \land (\neg x \lor y)$
0	0	0	1	xy	$x \land y$	$x \land y$
1	0	0	1	$1 + \mathrm{xy}$	$\neg(x \land y)$	$(\neg x) \lor (\neg y)$
0	1	0	1	$\mathrm{x} + \mathrm{xy}$	$x \land (\neg y)$	$x \land (\neg y)$
1	1	0	1	$1 + \mathrm{x} + \mathrm{xy}$	$x \implies y$	$(\neg x) \lor y$
0	0	1	1	$\mathrm{y} + \mathrm{xy}$	$(\neg x) \land y$	$(\neg x) \land y$
1	0	1	1	$1 + \mathrm{y} + \mathrm{xy}$	$x \impliedby y$	$x \lor (\neg y)$
0	1	1	1	$\mathrm{x} + \mathrm{y} + \mathrm{xy}$	$x \lor y$	$x \lor y$
1	1	1	1	$1 + \mathrm{x} + \mathrm{y} + \mathrm{xy}$	$\neg(x \lor y)$	$(\neg x) \land (\neg y)$

Facts about finite fields

- For every prime number p and positive integer n, there exists a finite field with p^n elements, denoted $GF(p^n)$, $\mathbb{Z}/p^n$ or $\mathbb{F}_p^n$.
- The following instruction sets have identical expressivity:
  - Pairs: $\{\vee, \neg\}, \{\wedge, \neg\}, \{\rightarrow, \neg\}, \{\rightarrow, \bot\}, \{\rightarrow, \veebar\}, \{\wedge, \veebar\}, \ldots$
  - Triples: $\{\vee, =, \veebar\}, \{\vee, \veebar, \top\}, \{\wedge, =, \bot\}, \{\wedge, =, \veebar\}, \{\wedge, \veebar, \top\}, \ldots$
- In other words, we can compute any Boolean function $\mathbb{B}^n \rightarrow \mathbb{B}$ by composing any one of the above operator sets in an orderly fashion.
- $\mathbb{F}_2$ corresponds to arithmetic modulo 2, i.e., $\oplus := \veebar, \otimes := \wedge$.
- There are (at least) two schools of thought about Boolean circuits:
  - Logical: Conjunctive Normal Form (CNF). May not be unique.
  - Algebra: Zhegalkin Normal Form (ZNF). Always unique.
- The type $\mathbb{F}_2^n \rightarrow \mathbb{F}_2$ possesses 2^{2^n} inhabitants.

Preface to "Two Memoirs on Pure Analysis"

"*Long algebraic calculations were at first hardly necessary for mathematical progress... It was only since Euler that concision has become indispensable to continuing the work this great geometer has given to science. Since Euler, calculation has become more and more necessary and... the algorithms so complicated that progress would be nearly impossible without the elegance that modern geometers have brought to bear on their research, and by which means the mind can promptly and with a glance grasp a large number of operations.*

...

It is clear that elegance, so admirably and aptly named, has no other purpose.

...

Jump headlong into the calculations! Group the operations, classify them by their difficulties and not their appearances. This, I believe, is the mission of future geometers. This is the road on which I am embarking in this work."

Évariste Galois, 1811-1832

What's the point?

- Algebraists have developed a powerful language for rootfinding
- Tradition handed down from Euler, Galois, Borel, Kleene, Chomsky
- We know closed forms for exponentials of structured matrices
- Solving these forms can be much faster than power iteration
- Unifies many problems in PL, probability and graph theory
- Context-free parsing is just rootfinding on a semiring algebra
- Type checking sans recursive types is just graph reachability
- Unification/simplification is lazy hypergraph search
- Bounded program synthesis is matrix factorization/completion
- By doing so, we can leverage well-known algebraic techniques

Future work

Parsing

- The line between parsing and computation is blurry
- Investigate connection between dynamical and term rewrite systems
- Extend Valiant's parser to tensors/context-sensitive languages
- Recover the original parse tree or eliminate Chomsky Normal Form
- What is the connection to Leibnizian differentiability?
- What is the meaning of abstract algebraic eigenvalues?

Probability

- Look into Markov chains (detailed balance, stationarity, reversibility)
- Fuse Valiant parser and probabilistic context-free grammar
- Message passing and graph diffusion processes
- Look into constrained optimization (e.g., L/QP) to rank feasible set

Special thanks

Nghi D. Q. Bui
Zhixin Xiong
Brigitte Pientka
David Yu-Tung Hui
Ori Roth
Justine Gehring

Learn more at:

https://github.com/breandan/tidyparse