Probabilistic Array Programming on Galois Fields

Breandan Considine, Jin Guo, Xujie Si

McGill University, Mila IQIA

breandan.considine@mail.mcgill.ca

June 13, 2022

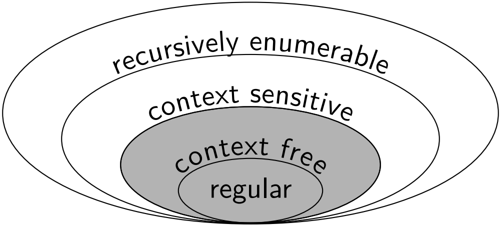
Overview

Recap: Context Free Grammars

Suppose we have a context free grammar (CFG) $G = \langle V, \Sigma, P, S \rangle$ where V is the set of nonterminals, Σ is the terminals, $P : V \times (V \cup \Sigma)^+$ are the productions, $S \in V$ is the start symbol and $+$ is the Kleene plus.

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., $\underline{S \to SS \mid (S) \mid ()} \Leftrightarrow^* \underline{S \to XR \mid SS \mid LR, L \to (, R \to), X \to LS}$



Algebraic parsing, distilled

Given a CFG $\mathcal{G} \coloneqq \langle V, \Sigma, P, S \rangle$ 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 $\mathcal{P}(V)$ be our domain, 0 be $\varnothing$, $\oplus$ be $\cup$, and $\otimes$ be defined as:

$$a \otimes b \coloneqq \{C \mid \langle A, B \rangle \in a \times b, (C \to AB) \in P\}$$

We initialize $\mathbf{M}_0[i,j](\mathcal{G}, \sigma) \coloneqq \{A \mid i + 1 = j, (A \to \sigma_i) \in P\}$ and search for a matrix $\mathbf{M}^*$ via fixpoint iteration,

$$\mathbf{M}^* = \begin{pmatrix} \varnothing & \{V\}_{\sigma_1} & \ldots & \ldots & \mathcal{T} \\ \varnothing & \varnothing & \{V\}_{\sigma_2} & \ldots & \ldots \\ \varnothing & \varnothing & \varnothing & \{V\}_{\sigma_3} & \ldots \\ \varnothing & \varnothing & \varnothing & \varnothing & \{V\}_{\sigma_4} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

where $\mathbf{M}^*$ is the least solution to $\mathbf{M} = \mathbf{M} + \mathbf{M}^2$. We can then define the recognizer as $R \coloneqq \mathbb{1}_{\mathcal{T}}(S) \iff \mathbb{1}_{\mathcal{L}(\mathcal{G})}(\sigma)$.

Kotlin implementation: CFG definition

```
typealias Production = Pair<String, List<String>>
typealias CFG = Set<Production>
val Production.LHS: String get() = first
val Production.RHS: List<String> get() = second
val CFG.nonterminals: Set<String> by cache { map { it.LHS }.toSet() }
val CFG.words: Set<String> by cache { nonterminals + flatMap { it.RHS } }
val CFG.terminals: Set<String> by cache { words - nonterminals }
// Many-to-many mapping of nonterminals to RHS expansions
val CFG.bimap: BidirectionalMap by cache { BidirectionalMap(this) }

fun CFG.makeAlgebra(): Ring<Set<String>> =
 Ring.of(
  // 0 = ∅
  nil = setOf(),
  // x + y = x ∪ y
  plus = { x, y → x union y },
  // x · y = { A0 | A1 ∈ x, A2 ∈ y, (A0 → A1 A2) ∈ P }
  times = { x, y → join(x, y) }
 )

fun CFG.join(ls: Set<String>, rs: Set<String>): Set<String> =
 (ls * rs).flatMap { (l, r) → bimap[listOf(l, r)] }.toSet()
```

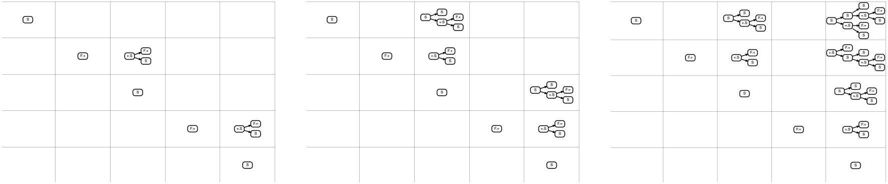
Kotlin implementation: the recognizer

```
// Constructs initial matrix according to: M_{i+1=j} = { A | (A → σ_i) ∈ P }
fun CFG.initialMatrix(str: List<String>): Matrix<Set<String>> =
 Matrix(makeAlgebra(), str.size + 1) { i, j ->
   // Aligns nonterminals matching each terminal along superdiagonal
   if (i + 1 != j) emptySet() else bimap[listOf(str[j - 1])].toSet()
 }

// Computes the fixpoint of an abstract matrix function
tailrec fun <T: Matrix<S>, S> T.seekFixpoint(op: (T) -> T): T {
 val next = op(this)
 return if (this == next) next else next.seekFixpoint(op)
}

// Checks whether start symbol is contained in the northeasternmost entry
fun CFG.check(s: String): Boolean = START in parse(tokenize(s))[0].last()

// Since matrix is strictly UT, this converges in at most |tokens| steps
fun CFG.parse(tokens: List<String>): Matrix<Set<String>> =
   initialMatrix(tokens).seekFixpoint { it + it * it }
```

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

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



- 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 $\mathbb{B}^{n\times n\times|V|}$ (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. $\mathbb{1}_{\subseteq V}: V \to \mathbb{B}^{|V|}$
- $\oplus, \otimes$ are defined as $\boxplus, \boxtimes$, so that the following diagram commutes:

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

- 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!

Tidyparse IDE plugin

arithmetic_left_recursive/arithmetic.tidy | mini_ocaml/ocaml.tidy | arithmetic_checked/arithmetic.tidy | arithmetic.cfg | simple.tidy | simple.cfg | test.tidy | ocaml/ocaml.tidy | SATValiantTest.kt

```
let rec a = _ _ _ _ _

let rec filter p l = m                                l -> if p x then x :: _ _ _ els

let curry f = ( fun x
```

```
let rec a = ( <X> , <X> )
let rec a = ( <X> , [] )
let rec a = ( <X> , filter )
let rec a = ( [] , [] )
let rec a = ( [] , filter )
```

Press ↩ to insert, ⇥ to replace

ocaml.cfg

```
S -> X
X -> A | V | ( X , X ) | X X | ( X )
A -> FUN | F | LI | M | L
FUN -> fun V `->` X
F -> if X then X else X
M -> match V with Branch
Branch -> `|` X `->` X | Branch Branch
L -> let V = X
L -> let rec V = X
LI -> L in X
```

ocaml.cfg

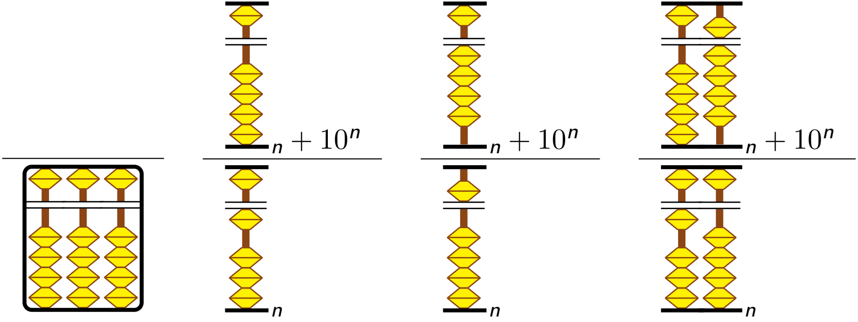
```
V -> Vexp | ( Vexp ) | List | Vexp Vexp
Vexp -> Vname | FunName | Vexp VO Vexp | B
Vexp -> ( Vname , Vname ) | Vexp Vexp | I
List -> [] | V :: V
Vname -> a | b | c | d | e | f | g | h | i
Vname -> j | k | l | m | n | o | p | q | r
Vname -> s | t | u | v | w | x | y | z
FunName -> foldright | map | filter
FunName -> curry | uncurry | ( VO )
VO ->  + | - | * | / | >
VO -> = | < | `||` | `&&`
I -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
B ->  true | false
```

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



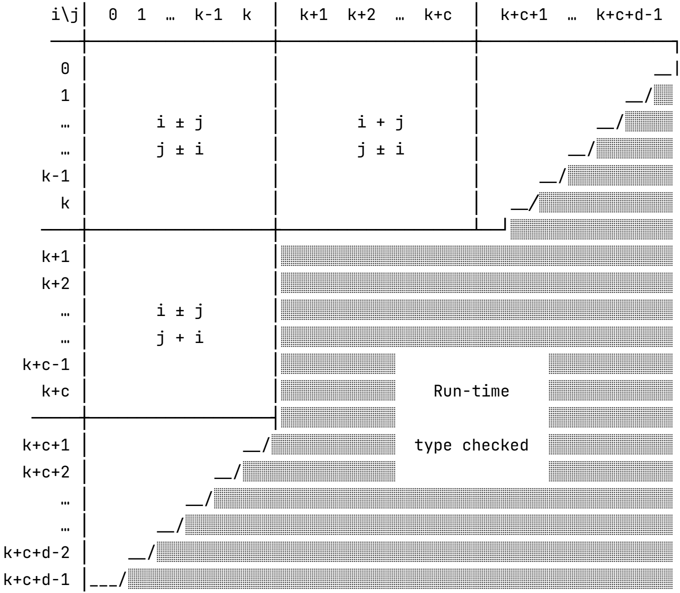
Abacus dependent types

```
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
- 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>
```

Linear Finite State Registers

Let $\mathbf{M} : \mathrm{GF}(2^{n \times n})$ be a square matrix $\mathbf{M}^0_{r,c} = P_c$ if $r = 0$ else $\mathbb{1}[c = r - 1]$, where P is a feedback polynomial over $GF(2^n)$ with coefficients $P_{1\ldots n}$ and semiring operators $\otimes := \veebar, \oplus := \vee$:

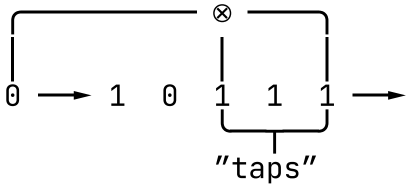
$$\mathbf{M}^t V = \begin{pmatrix} P_1 & P_2 & P_3 & P_4 & P_5 \\ \top & \circ & \circ & \circ & \circ \\ \circ & \top & \circ & \circ & \circ \\ \circ & \circ & \top & \circ & \circ \\ \circ & \circ & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{pmatrix}$$

Selecting any $V \neq \mathbf{0}$ and coefficients P_j from a known *primitive polynomial* generates an ergodic sequence over $\mathrm{GF}(2^n)$ with full periodicity:

$$\mathbf{S} = \begin{pmatrix} V & \mathbf{M}V & \mathbf{M}^2 V & \cdots & \mathbf{M}^{2^n-1} V \end{pmatrix}$$

Linear finite state registers

$$P = 18 = \begin{array}{ccccc} 1 & + & x^3 & + & x^5 \\ 0 & 0 & 1 & 0 & 1 \\ \| & \| & \| & \| & \| \\ a & b & c & d & e \end{array}$$



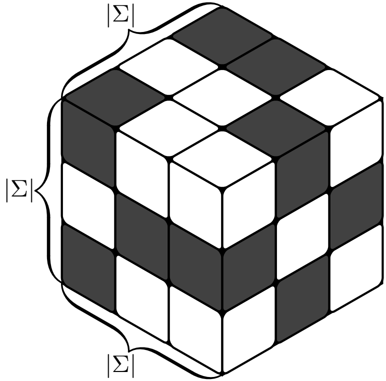
S_0 =	1	0	1	1	1
S_1 =	0	1	0	1	1
S_2 =	1	0	1	0	1
S_3 =	0	1	0	1	0
S_4 =	0	0	1	0	1

$$\begin{array}{ccccc} a & b & c & d & e \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array} \quad * \quad \begin{array}{c} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{array}$$

$$M \quad * \quad V$$

$$\begin{array}{ccccc} S_4 & = M \ldots & S_1 & = M * & V \\ \| & & \| & & \| \\ 0 & & 0 & & 1 \\ 0 & & 1 & & 0 \\ 1 & & 0 & & 1 \\ 0 & & 1 & & 1 \\ 1 & & 0 & & 1 \end{array}$$

Multidimensional sampling: the hasty pudding trick



To uniformly sample $\boldsymbol{\sigma} \sim \Sigma^n$ without replacement, we could track historical samples, or, we can form an injection $GF(2^n) \rightharpoonup \Sigma^d$, cycle a primitive polynomial over $GF(2^n)$, then discard samples that do not identify an element in any indexed dimension. This procedure rejects $(1 - |\Sigma| 2^{-\lceil \log_2 |\Sigma| \rceil})^d$ samples on average and requires $\sim \mathcal{O}(1)$.

e.g., $\Sigma^2 = \{A, B, C\}^2$, $x^4 + x^3 + 1$

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
||	||	||	||	||	||	||	||
[1000]	[0100]	[0010]	[1001]	[1100]	[0110]	[1011]	[0101]
C A	B A	A C	C B		B C		B B

Multidimensional No-Replacement Sampler

```kotlin
fun List<Int>.bitLens() = map { ceil(log2(it.toDouble())).toInt() }

// Splits a bitvector into designated chunks and returns indices
// (10101011, [3, 2, 3]) → [101, 01, 011] → [4, 1, 3]
fun List<Boolean>.toIndexes(bitLens: List<Int>): List<Int> =
 bitLens.fold(listOf<List<Boolean>>() to this) { (a, b), i →
   (a + listOf(b.take(i))) to b.drop(i)
 }.first.map { it.toInt() }

fun Sequence<List<Boolean>>.hastyPudding(lengths: List<Int>) =
 map { it.toIndexes(lengths.bitLens()) }
  .filter { it.zip(lengths).all { (a, b) → a < b } }

fun <T> List<Set<T>>.sampleWithoutReplacement(
 lengths: List<Int> = map { it.size },
 bitLens: List<Int> = map(Set<T>::size).bitLens(),
 degree: Int = bitLens.sum().also { println("LFSR(GF(2^$it))") }
): Sequence<List<T>> =
 LFSR(degree).hastyPudding(lengths)
   .map { zip(it).map { (dims, idx) → dims[idx] } }
```

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?

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

Learn more at:

http://array22.ndan.co

Special thanks

Nghi D. Q. Bui

Zhixin Xiong

Jaylene Zhang

David Yu-Tung Hui

Fabian Muehlboeck

Ben Greenman

McGill UNIVERSITY

FPT® Fpt Software

Mila