Probabilistic Array Programming on Galois Fields

Anonymous Author(s)

Abstract

We present a framework for probabilistic program synthesis based on Galois theory. This framework is capable of modeling discrete distributions, algebraic parsing, graph representation learning and sketch-based program synthesis, all using sparse matrix multiplication on $GF(p^n)$. This elegant representation allows us to leverage complexity-theoretic lower bounds and easily access various compiler targets, including low level languages like VHDL and netlist as well as higher level IRs such as JVM, LLVM, and JS using the same codebase. We discuss its theory, implementation and a few use cases, which include learning and sketching syntax in context-free languages via SAT/SMT encoding.

1 Introduction

A Galois field is a field containing a finite set of elements, e.g., $\mathbb{Z}/n$ where n is prime. We are primarily interested in $\text{GF}(2^n)$, taking values on $\{0,1\}^n$, due to its amenability to SAT/SMT encoding, well-studied theoretical properties, circuit synthesizability and broad applicability to signal processing and computational linguistics. For example, the theory of context-free grammars are a special case [2, 22]. Given a CFG $\mathcal{G} \coloneqq \langle V, \Sigma, P, S\rangle$ in Chomsky Normal Form, we can construct a recognizer $R_{\mathcal{G}} : \Sigma^n \rightarrow \mathbb{B}$ 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:

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

We initialize $\mathbf{M}^0_{r,c}(\mathcal{G},\sigma) \coloneqq \{V \mid c = r+1, (V \rightarrow \sigma_r) \in P\}$ 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 $R \coloneqq \mathbb{1}_{\mathcal{T}}(S) \iff \mathbb{1}_{\mathcal{L}(\mathcal{G})}(\sigma)$.

This decision procedure can be lowered to binary matrices by noting $\bigoplus_{k=1}^{n} \mathbf{M}_{ik} \otimes \mathbf{M}_{kj}$ has cardinality bounded by $|V|$ and is thus representable as a fixed-length vector. Full details of this bisimilarity can be found in Valiant [38] and Lee [25], who proves its time complexity to be $\mathcal{O}(n^{\omega})$ where ω is the matrix multiplication bound (currently $\omega < 2.763$ [19] as of writing this manuscript). Assuming sparsity, this technique can typically be reduced to linearithmic time, and is currently the best asymptotic bound for CFL recognition to date.

ARRAY'22, June 13, 2022, San Diego, CA, USA

2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

A similar procedure may be used to learn, parse and sample from probabilistic grammars such as PCFGs [16] and probabilistic circuits [30] using *semirings*, many of which have carrier sets that can be compactly represented as elements of $\text{GF}(2^n)$. Known as *propagation* or *message passing*, this procedure consists of two steps: *aggregate* and *update*. Let δ_{st} denote some distance metric on a path between vertices s and t in a graph. To obtain δ_{st}, one may run the following procedure using a desired path algebra until convergence:

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

$\oplus$	$\otimes$	⓪	①	Path
min	+	∞	0	Shortest
max	+	$-\infty$	0	Longest
max	min	0	∞	Widest
$\vee$	$\underline{\vee}$	$\circ$	$\top$	Random

Many dynamic programming algorithms, including Bellman-Ford, Floyd-Warshall, Dijkstra's shortest path, as well as belief, constraint, error, expectation and backpropagation can be neatly expressed as semiring algebras. We refer the curious reader to Gondran [15] and Baras [3] for an extensive survey of the algebraic path problem and its many wonderful applications throughout statistics and computer science.

$\text{GF}(2^n)$ can also be used to search over highly complex state spaces and sample without replacement from arbitrarily large sets. Let $\mathbf{M} : \text{GF}(2^{n \times n})$ be a square matrix defined as $\mathbf{M}^0_{r,c} = P_c$ 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 $\otimes \coloneqq \underline{\vee}, \oplus \coloneqq \vee$ lifted to matrices in the usual way:

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

Selecting any $V \neq \mathbf{0}$ and coefficients P_j from a primitive polynomial [35] produces a fixpoint operator generating an ergodic sequence over $\text{GF}(2^n)$ with full periodicity. That is, the sequence $\mathbf{V} = \begin{pmatrix} V & \mathbf{M}V & \mathbf{M}^2V & \cdots & \mathbf{M}^{2^n-1}V \end{pmatrix}$ forms a space-filling curve whose trajectory tours the full state space in pseudorandom order without repetition. Known as a linear finite state register (LFSR) [24], this circuit is one of the fastest known pseudorandom number generators, drawing samples without replacement from indexed families S_V in $\mathcal{O}(\log |S|)$ space and $\mathcal{O}(1)$ time – useful for weighted search and density estimation via inverse transform sampling.

Together, these relatively simple array primitives may be composed to build an expressive family of probability distributions over non-trivial kinds of algebraic data types such as bounded-width regular and context-free languages.

2 From array programs to graphs and back

Graphs are algebraic structures [41] capable of representing many procedural and relational phenomena. A graph can be represented as a matrix $\{\mathbb{B}, \Sigma^k, \mathbb{N}/n\}^{|G|\times|G|}$, whose entries describe the presence, label or type signature of an edge between two vertices. Not only can graphs be represented as arrays, but also as CFGs. Algebra provides a unifying language for studying many graph algorithms and program analysis tasks [23]. Just like CFGs, graphs can also be defined algebraically, using an algebraic data type [28]. Considering Erwig's [13] inductive graph definition, we notice that,

$$\begin{array}{lcl} \text{vertex} & \rightarrow & \text{int} \\ \text{neighbors} & \rightarrow & [\text{vertex}] \\ \text{context} & \rightarrow & (\text{neighbors, vertex, adj}) \\ \text{graph} & \rightarrow & \text{empty} \mid \text{context \& graph} \end{array}$$

bears a striking resemblance to a CFG! In fact, we use this correspondence to parse graphs, visualize CFGs and type check array programs. Many paths can be taken to translate between languages, graphs, types, arrays and algebras. Depicted in the transition matrix below are a few possibilities:

	Graphs	Types	CFGs	Arrays	Algebras
Graphs		ATG [28]	CFGG [29]	Laplacian	CP [36]
Types	Lattice [12]		STM [34]	TLP [32]	ADT [26]
CFGs	Reachability [33]	HOAS [31]		Valiant [38]	GF(2^n)
Arrays	Knowledge [21]	NF [14]	SQL [7]		Codd [10]
Algebras	Circuits [39, 40]	TCAH [37]	CFGL [1]	Cayley [6]	

Table 1. Where ATG are algebraically typed graphs, CFGG are context-free graph grammars, CP is a characteristic polynomial, STM is the subtyping machine, TLP is tabled logic programming, ADT is an algebraic data type, HOAS is higher order abstract syntax, KGs are knowledge graphs, NF is the Naperian functor, SQL is structured query language, TCAH is type-class algebra hierarchy and CFGL are context-free group languages. Entries highlighted in gray have been concretized by our DSL.

Although we have only explored a subset of this design space, our experience has already shed light on the rich theoretical connections between programming languages, graphs and linear algebra. We have recently discovered a novel application for algebraic parsing, developed an algorithm for sketch-based CFL synthesis and lowered it onto SAT/SMT solver. We believe further exploration of this space promises to yield yet-undiscovered applications for array programming and program synthesis in general.

3 Implementation

Among the core features which our DSL provides include:

- Type and shape inference for multidimensional arrays
- Compilation of array programs to SAT/SMT solvers
- Array-based graph representation and manipulation
- Tools for spectral and algebraic graph theory
- Backpropagation and other message passing schemes
- Lazily-evaluated sparse multidimensional arrays
- Multiplatform compilation: JS/JVM/Native/VHDL
- Notebook- and browser-based visualizations

Our DSL benefits from the following design patterns:

- Abacus-based dependent types simulating GF(2^n)
- Typeclass based algebras inspired by Spitters et al. [37]
- Algebraic graph-constructors inspired by Mokhov [28]
- Type-family for graphs inspired by Greenman et al. [17]
- Multidimensional arrays inspired by Gibbons [14]
- Nested datatypes inspired Bird and Meertens [4]

We implemented some of our favorite algorithms in the DSL:

- Valiant's algebraic context-free language recognizer [38]
- Embedded DSL for CFL normalization/sketching [9]
- Matrix completion with SAT/SMT solving
- Regular language parsing and induction
- Algebraic and probabilistic circuits [8]
- Algorithmic/automatic differentiation [11]
- Graph embedding and dimensionality reduction [18]
- Persistent homology embeddings of source code
- Monoidal counting tensors with mergable summaries
- Weighted and unweighted sampling with LFSRs [24]
- Full-factorial multivariate analysis of variance
- Sparse, dense and higher-order Markov chains
- Preconditioning and multistochastic tensor balancing
- Property-based testing with top-down tree synthesis

4 Conclusion

Programs are graphical structures with a rich denotational and operational semantics [20]. Many useful graph representations have been proposed, including call graphs, dataflow graphs, computation graphs [5], e-Graphs [42], down to arithmetic [27] and probabilistic circuits [8]. Our DSL celebrates the duality between arrays and programs, supporting both programmatically-generated arrays and array-based representations of programs via the Valiant correspondence (it is possible to interpret either or both as labeled graphs for visualization). Although we have not yet implemented a self-interpreter, this would be a promising avenue for future work. Primarily, we use our DSL to generate counterfactuals for evaluating machine learning models on source code.

Our framework provides various animations which have been developed to facilitate visual pattern matching. Users can pause, play and rewind a graph program trace to see message passing in slow motion. This feature is invaluable for inspecting and debugging graph dynamical processes.

In our ARRAY presentation, we will describe some of the applications we have developed using this framework. We will explore the idea of generic array programming with abstract algebras, define an algebraic type family for graphs, then show how our DSL can be used to compose and evaluate graphs representing probabilistic programs. We will show a concrete application for sketch-based program synthesis with applications to robust parsing and rewriting. Our DSL has direct applicability to learning and reasoning about source code and inductive programming.

References

[1] Anatoly Anisimov. 1971. Group languages. *Cybernetics* (1971), 594–601. https://link.springer.com/content/pdf/10.1007/BF01071030.pdf

[2] Ekaterina Bakinova, Artem Basharin, Igor Batmanov, Konstantin Lyubort, Alexander Okhotin, and Elizaveta Sazhneva. 2020. Formal languages over GF(2). *Information and Computation* (2020), 104672. https://users.math-cs.spbu.ru/~okhotin/papers/formal_languages_gf2.pdf

[3] John S Baras and George Theodorakopoulos. 2010. Path problems in networks. *Synthesis Lectures on Communication Networks* 3, 1 (2010).

[4] Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *International Conference on Mathematics of Program Construction*. Springer, 52–67. http://www.cs.ox.ac.uk/richard.bird/online/BirdMeertens98Nested.pdf

[5] Olivier Breuleux and Bart van Merriënboer. 2017. Automatic differentiation in Myia. (2017). https://mlsys.org/Conferences/doc/2018/39.pdf

[6] Arthur Cayley. 1854. On the theory of groups, as depending on the symbolic equation $\theta^n = 1$. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 7, 42 (1854), 40–47.

[7] Donald D Chamberlin. 2012. Early history of SQL. *IEEE Annals of the History of Computing* 34, 4 (2012), 78–82.

[8] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. 2020. Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models. (2020). http://starai.cs.ucla.edu/papers/ProbCirc20.pdf

[9] Noam Chomsky and Marcel P Schützenberger. 1959. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 118–161.

[10] Edgar F Codd. 2002. A relational model of data for large shared data banks. In *Software pioneers*. Springer, 263–294. https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

[11] Breandan Considine. 2019. *Programming Tools for Intelligent Systems*. Master's thesis. Université de Montréal. https://ndan.co/public/masters_thesis.pdf

[12] Stephen Dolan. 2016. *Algebraic subtyping*. https://www.cs.tufts.edu/~nr/cs257/archive/stephen-dolan/thesis.pdf

[13] Martin Erwig. 2001. Inductive graphs and functional graph algorithms. *Journal of Functional Programming* 11, 5 (2001), 467–492. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.9377&rep=rep1&type=pdf

[14] Jeremy Gibbons. 2017. Aplicative programming with naperian functors. In *European Symposium on Programming*. Springer, 556–583.

[15] Michel Gondran and Michel Minoux. 2008. *Graphs, dioids and semirings: new models and algorithms*. Springer Science & Business Media.

[16] Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics* 25, 4 (1999), 573–606. https://aclanthology.org/J99-4004.pdf

[17] Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded polymorphism into shape. *ACM SIGPLAN Notices* 49, 6 (2014), 89–99. https://www.cs.cornell.edu/~blg59/resources/doc/effing-bound-polymorphism.pdf

[18] William L Hamilton. 2020. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14, 3 (2020), 1–159. https://www.cs.mcgill.ca/~wlh/grl_book/files/GRL_Book.pdf

[19] David G Harris. 2021. Improved algorithms for Boolean matrix multiplication via opportunistic matrix multiplication. *arXiv preprint arXiv:2109.13335* (2021). https://arxiv.org/pdf/2109.13335.pdf

[20] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 163–174. https://arxiv.org/pdf/1803.06686.pdf

[21] Aidan Hogan, Eva Blomqvist, Michael Cochez, et al. 2021. Knowledge graphs. *Synthesis Lectures on Data, Semantics, and Knowledge* (2021). https://dl.acm.org/doi/pdf/10.1145/3447772

[22] Patrik Jansson and Jean-Philippe Bernardy. 2016. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *Logical Methods in Computer Science* 12 (2016). https://arxiv.org/pdf/1601.07724.pdf

[23] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[24] Andreas Klein. 2013. Linear feedback shift registers. In *Stream Ciphers*. Springer, 17–58. https://link.springer.com/content/pdf/10.1007/978-1-4471-5079-4.pdf

[25] Lillian Lee. 2002. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)* 49, 1 (2002), 1–15. https://arxiv.org/pdf/cs/0112018.pdf

[26] Grant Reynold Malcolm. 1990. *Algebraic data types and program transformation*. Ph.D. Dissertation. Rijksuniversiteit Groningen.

[27] Gary L Miller, Vijaya Ramachandran, and Erich Kaltofen. 1988. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput.* 17, 4 (1988), 687–695. https://users.cs.duke.edu/~elk27/bibliography/88/MRK88.pdf

[28] Andrey Mokhov. 2017. Algebraic graphs with class (functional pearl). *ACM SIGPLAN Notices* 52, 10 (2017), 2–13. https://eprints.ncl.ac.uk/file_store/production/239461/EF82F5FE-66E3-4F64-A1AC-A366D1961738.pdf

[29] Theodosios Pavlidis. 1972. Linear and context-free graph grammars. *Journal of the ACM (JACM)* 19, 1 (1972), 11–22.

[30] Robert Peharz. 2015. *Foundations of sum-product networks for probabilistic modeling*. Ph.D. Dissertation. PhD thesis, Medical University of Graz. https://diglib.tugraz.at/download.php?id=576a7b8cc939f&location=browse

[31] Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. *ACM sigplan notices* 23, 7 (1988), 199–208. https://www.cs.cmu.edu/~fp/papers/pldi88.pdf

[32] Brigitte Pientka. 2005. Tabling for higher-order logic programming. In *International Conference on Automated Deduction*. Springer, 54–68. https://www.cs.mcgill.ca/~bpientka/papers/eftab-long.pdf

[33] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11-12 (1998), 701–726.

[34] Ori Roth. 2021. Study of the Subtyping Machine of Nominal Subtyping with Variance (full version). *arXiv preprint arXiv:2109.03950* (2021). https://arxiv.org/pdf/2109.03950.pdf

[35] Nirmal R Saxena and Edward J Mccluskey. 2004. Primitive polynomial generation algorithms implementation and performance analysis. *CRC Technical Report 2004* (2004). http://crc.stanford.edu/crc_papers/CRC-TR-04-03.pdf

[36] Allen J Schwenk. 1974. Computing the characteristic polynomial of a graph. In *Graphs and combinatorics*. Springer, 153–172.

[37] Bas Spitters and Eelis Van der Weegen. 2011. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21, 4 (2011), 795–825. https://arxiv.org/pdf/1102.1323.pdf

[38] Leslie G Valiant. 1975. General context-free recognition in less than cubic time. *Journal of computer and system sciences* 10, 2 (1975), 308–315. http://people.csail.mit.edu/virgi/6.s078/papers/valiant.pdf

[39] Leslie G Valiant. 1979. Completeness classes in algebra. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*. 249–261.

[40] Leslie G Valiant. 1992. Why is Boolean complexity theory difficult. *Boolean Function Complexity* 169, 84-94 (1992), 4. http://people.seas.harvard.edu/~valiant/whybool.pdf

[41] B Weisfeiler and A Leman. 1968. The reduction of a graph to canonical form and the algebgra which appears therein. *NTI, Series* 2 (1968). https://www.iti.zcu.cz/wl2018/pdf/wl_paper_translation.pdf

[42] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2020. egg: Easy, Efficient, and Extensible E-graphs. *arXiv preprint arXiv:2004.03082* (2020). https://arxiv.org/pdf/2004.03082.pdf