# Lobster: A GPU-Accelerated Framework for Neurosymbolic Programming

Paul Biberstein
paulbib@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Ziyang Li
liby99@seas.upenn.edu
Johns Hopkins University
Baltimore, Maryland, USA

Joseph Devietti
devietti@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Mayur Naik
mhnaik@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

## Abstract

Neurosymbolic programs combine deep learning with symbolic reasoning to achieve better data efficiency, interpretability, and generalizability compared to standalone deep learning approaches. However, existing neurosymbolic learning frameworks implement an uneasy marriage between a highly scalable, GPU-accelerated neural component and a slower symbolic component that runs on CPUs.

We propose Lobster, a unified framework for harnessing GPUs in an end-to-end manner for neurosymbolic learning. Lobster maps a general neurosymbolic language based on Datalog to the GPU programming paradigm. This mapping is implemented via compilation to a new intermediate language called APM. The extra abstraction provided by apm allows Lobster to be both flexible, supporting discrete, probabilistic, and differentiable modes of reasoning on GPU hardware with a library of provenance semirings, and performant, implementing new optimization passes.

We demonstrate that Lobster programs can solve interesting problems spanning the domains of natural language processing, image processing, program reasoning, bioinformatics, and planning. On a suite of 9 applications, Lobster achieves an average speedup of 3.9x over Scallop, a state-of-the-art neurosymbolic framework, and enables scaling of neurosymbolic solutions to previously infeasible tasks.

***CCS Concepts:*** • **Theory of computation → Probabilistic computation**; • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Compilers**.

***Keywords:*** neurosymbolic programming, GPU acceleration, Datalog, compiler optimizations

**ACM Reference Format:**
Paul Biberstein, Ziyang Li, Joseph Devietti, and Mayur Naik. 2026. Lobster: A GPU-Accelerated Framework for Neurosymbolic Programming. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3760250.3762232

## 1 Introduction

Deep learning and classical algorithms represent two predominant paradigms of modern programming. Classical algorithms excel at problems with clearly defined rules and structured data, such as sorting a list of numbers or finding a shortest path in a graph. In contrast, deep learning is well suited to contexts where classical algorithmic approaches become intractable, particularly for problems involving noisy, complex, and high-dimensional data—such as detecting objects in an image or parsing natural language text.

Many machine learning problems in different domains demand the complementary capabilities of these two paradigms. Neurosymbolic programming [6] is an emerging approach to solve such problems by suitably decomposing the computation between a neural network and a symbolic program. The resulting *neurosymbolic programs* have been demonstrated to achieve better data efficiency, interpretability, and generalizability compared to standalone deep learning approaches. Such properties are crucial for various safety-critical domains such as system security [25], cyber-physical systems [54], and healthcare [48].

Recent frameworks such as DeepProbLog [28], Scallop [26], and ISED [41] have enhanced the programmability and accessibility of neurosymbolic applications. Figure 1 illustrates a neurosymbolic program for solving a binary image-classification problem [45]. The symbolic program is specified in Datalog [2], a declarative language. Crucially, by using a differentiable Datalog engine, gradients can be back-propagated through the program to train the neural network, thereby enabling automatic learning of relevant image features without manual engineering.

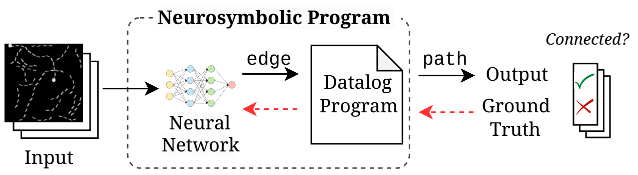


**Figure 1.** An example neurosymbolic program.

Despite their benefits, neurosymbolic programs using these frameworks incur significant computational overhead during training and inference. The scalability challenges stem primarily from managing probabilistic data and maintaining end-to-end differentiability. For instance, in the neurosymbolic program in Figure 1, a neural network identifies edges in an image. These edges are represented in an `edge` relation, where each tuple has an associated probability representing the network's confidence. The symbolic program computes the transitive closure to produce a `path` relation, where the probability of each path tuple must take into account all the possible ways to derive it from the edge tuples, and their associated probabilities. Since it is often intractable to perform exact probabilistic reasoning, approximated probabilistic inference is employed, though this has only limited scalability benefits. Differentiability further complicates the problem by requiring us to track each input's contribution to the output, increasing space and time complexity due to the extra book-keeping required for gradients.

In this paper, we propose Lobster, a GPU-accelerated framework designed to enhance the scalability of neurosymbolic programming. Lobster's core innovation is efficiently mapping Datalog—a logic programming language shown to be effective in neurosymbolic contexts [26]—onto GPU architectures, for different modes of reasoning: discrete, probabilistic, and differentiable. The key architectural choice that enables the efficiency of this mapping is compilation to a new intermediate language, called APM, which is expressive enough to support complex reasoning but at the same time restricted enough to ensure massively parallel execution.

Supporting both advanced reasoning modes and GPU acceleration makes Lobster the first system of its kind, as shown in Figure 2. While various engines exist for discrete [36], probabilistic [13], and differentiable [26] settings, they are limited to CPU runtimes with single- or multi-threading. Other work [38, 44] implements GPU-accelerated Datalog execution, but does not support differentiable and probabilistic reasoning. In contrast, Lobster handles general neurosymbolic queries with multiple reasoning modes within a unified, GPU-accelerated framework of *provenance semirings* [17]. This requires fundamental changes to the program semantics compared to discrete Datalog, enriching the runtime with semiring-based tags that propagate alongside data throughout the computation. To support this efficiently on modern hardware, Lobster introduces GPU-optimized operators specifically tailored for tagged computation. By supporting a library of 7 common semirings in the literature, Lobster allows employing reasoning in a particular mode (e.g. probabilistic) by simply selecting a suitable corresponding semiring (e.g. Top-$k$-Proofs).

In addition to describing the compilation to and execution of APM, we propose a number of optimizations unique to Lobster and discuss considerations for our implementation—a fully-fledged compiler and runtime written in Rust—that can execute existing Datalog-based neurosymbolic programs on GPUs without any modifications.

In summary, the core contributions of this paper are:

- We introduce Lobster, the first GPU-accelerated neurosymbolic programming framework.
- We propose the APM language and show how to compile Datalog to APM.
- We implement a compiler and runtime for Lobster using Rust and CUDA.
- We evaluate Lobster on an extensive set of discrete, probabilistic, and differentiable benchmarks, showing that Lobster consistently outperforms prior systems, including more specialized ones. Lobster achieves a speedup of 3.9x on average and upto >100× over Scallop [27], the closest existing state-of-the-art system.

The code for Lobster is publicly available at https://github.com/P-bibs/Lobster.

The rest of the paper is organized as follows. We first give an illustrative overview of Lobster in Section 2. Then we introduce Lobster's core language and compiler (Section 3), optimizations (Section 4), and implementation details (Section 5). Finally, we present experimental results (Section 6) and related work (Section 7).

## 2 Illustrative Overview

We illustrate Lobster using an example image-reasoning task Pathfinder [45] wherein the goal is to determine whether two dots in the input image are connected by dashed lines (Figure 3). Neurosymbolic methods have been shown to achieve greater accuracy than purely neural methods on this task [27], but the symbolic performance bottleneck quickly appears as the lengths and complexities of lines (and therefore reasoning chain size) increase.

### 2.1 A Neurosymbolic Solution

As with many visual reasoning tasks, the Pathfinder task presents an obvious opportunity for neural/symbolic decomposition: a neural network can extract a structured representation from the image, and a symbolic engine can be used to reason over this representation.

More formally, we choose a discretization factor $n$ and overlay the lattice graph $G_{n,n}$ on the image, where each vertex corresponds to a square region of pixels. A convolutional

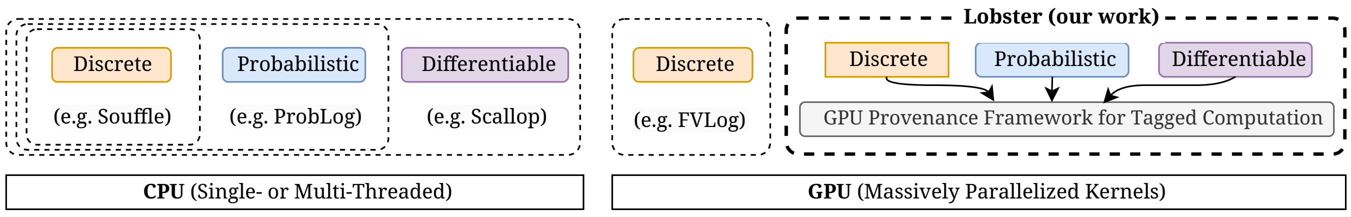
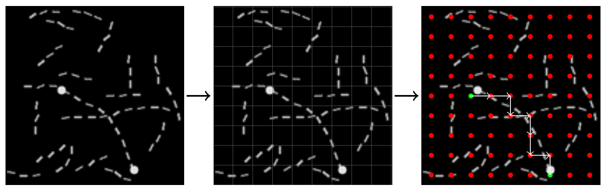
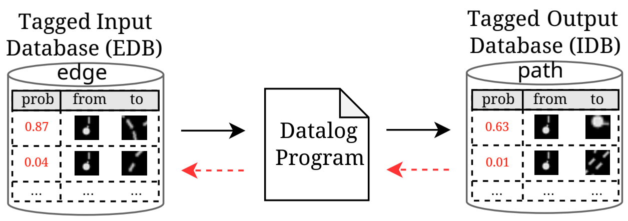


**Figure 2.** High-level comparison between Lobster and existing frameworks.



**(a)** An example graph a trained model extracts for the Pathfinder task. Green vertices are endpoints and white edges are predicted connectivity. Additional edges not relevant to endpoint connectivity are elided for clarity.



**(b)** The input and output for the Pathfinder Datalog program.

```
1  type Cell = u32
2  type edge(x: Cell, y: Cell)
3  type is_endpoint(x: Cell)
4
5  rel path(x, y) :-
6    edge(x,y) or (path(x, z) and edge(z, y)).
7  rel endpoints_connected() :- is_endpoint(x),
8    is_endpoint(y), path(x, y), x != y.
```

**(c)** A Lobster program that computes reachability information for a graph extracted by a neural network.

| Accuracy | Neural | Neurosymbolic |
|---|---|---|
| Pathfinder | 71.40 | 87.42 |
| Pathfinder-x | 49.80 | 89.46 |

**(d)** Accuracy of neural methods vs. neurosymbolic methods [27]. "Pathfinder-x" is a more challenging dataset.

| | Scallop | Lobster |
|---|---|---|
| Training Time | 41 hr. | 32 hr |

**(e)** Training time of Scallop versus Lobster.

**Figure 3.** The overall pipeline, symbolic program, and the acceleration result of using Lobster for the Pathfinder task.

neural network model [22] predicts probabilities for each edge in $G_{n,n}$, with high probability indicating confidence that two grid cells are connected by a dashed line. Additionally, the model predicts a probability corresponding to the presence of a dot in each grid cell. The symbolic component then computes the transitive closure of the graph and queries the result to determine the probability two dots are connected.

### 2.2 A Scallop Implementation

We present the high-level pipeline of implementing a neurosmbolic solution using Scallop, a state-of-the-art neurosymbolic framework, in Figure 3a. We specify the symbolic component as a Datalog program which is shown in Figure 3c. Datalog offers an intuitive interface for defining data types (line 1) and relation types (lines 2-3) which we use to specify the inputs. Notice that the graph is encoded as a binary relation between `Cell`s, which we represent with unsigned integers. Our program contains recursive rules, which are key for modeling reachability in a graph. The actual reachability rule is defined succinctly in line 6.

Notably, the probabilistic reasoning semantics is abstracted away. The `edge` relation in Figure 3c (line 2) contains probabilistic facts extracted by the underlying neural network. For example, a fact `0.97::edge(0, 1)` represents a prediction that a dotted line connects cell 0 and cell 1 with the probability of 0.97. All the derived facts, such as `path(i, j)`, will carry probabilities computed from a concrete set of edges that are used to derive them. A proof for `path(1,5)`, for example, could be the conjunction `is_endpoint(1)` $\wedge$ `edge(1, 2)` $\wedge \cdots \wedge$ `edge(4, 5)` $\wedge$ `is_endpoint(5)`, representing a concrete path formed by a subset of the probabilistic input facts. This computation is manifested by the underlying provenance framework within Lobster. In our solution, we use a probabilistic provenance called `diff-top-1-proofs` which instruments the program to carry the most-likely path between any two cells.

Since the symbolic engine is differentiable, the entire neurosymbolic system can be trained end-to-end, resulting in a trained neural network despite only having "yes"/"no" supervision for each image. The neurosymbolic solution achieves

87.42% accuracy on the Pathfinder task, surpassing the 71.40% accuracy of purely neural methods (Figure 3d).

### 2.3 Scalability and Programmability Challenges

The neurosymbolic solution poses a significant scalability challenge. While the neural component can utilize modern hardware accelerators like GPUs and TPUs, the symbolic component runs on CPUs alone. This presents a performance bottleneck, as in order to calculate derivatives, the symbolic engine must consider all possible predicted graphs and their associated probabilities. As the size of the input image or difficulty of line curvature increases, the number of possible structures and the size of their associated weights also grows exponentially, leading to a combinatorial explosion in the number of required computations.

An expert could write custom GPU kernels to accelerate this specific task, but this requires specialized knowledge of CUDA performance tuning. Instead, we seek to build a general, GPU-accelerated framework that accelerates any logic program used as part of a neurosymbolic pipeline, in order to make GPU acceleration widely accessible.

### 2.4 Our Results

Figure 3e shows the speedup of Lobster over the CPU baseline, Scallop, on the Pathfinder task. Scallop spends significant time on symbolic computation, while Lobster can perform the requisite combinatorial graph processing much quicker. Importantly, users do not need to change existing neurosymbolic programs to benefit from the acceleration. Lobster's efficiency gains are enabled by mapping a significant fragment of Datalog to the GPU programming paradigm, and making judicious decisions for representing relations, parallelizing relational operators, and scheduling computation which we describe next.

***How to Represent Relations?*** Lobster uses a flat, column-oriented layout in order to make optimal use of the GPU memory hierarchy. While performant CPU Datalog engines such as Soufflé make use of multi-layer data structures such as B-trees [21], our simple layout makes more sense in the context of GPU acceleration as column-oriented layouts are cache-friendly, and GPU programs are often memory-bound rather than compute-bound. Concretely, this means some of the simpler operations in the Pathfinder task, such as `union`ing the relation `edge` with the relation `path` to initially populate `path`, can reach close to 100% utilization of the GPU memory bus. This choice also suits the context of executing Datalog programs. For instance, the transitive closure operation is compiled to a series of query operations consisting of relational operations like *join* and *project* which operate on specific columns. As a result, columnar data allows more natural algorithm implementations. We discuss the memory layout further in Section 5.

***How to Parallelize Relational Operators?*** Beyond memory layout, efficient algorithms are necessary to improve the performance of symbolic computations. In Lobster, the key insight is that Datalog programs are compiled to a core set of relational queries, and each of these queries can be individually parallelized to improve their performance on potentially massive inputs. For example, the rule on line 6 of Figure 3c is compiled to a query that includes joining the entire set of currently discovered paths against the base edge set. The size of the input to this join grows exponentially as the program iterates, so executing the join with data-parallelism with respect to its input is critical. We describe how our compiler exposes this parallelism in Section 3.

***How to Handle Provenance Tags?*** Computing derivatives of the symbolic computation is necessary for training the neural network via gradient descent. This necessitates tracking the provenance of each fact in the Datalog program which involves computing over large and potentially complicated semiring tags. To adapt powerful but complex semirings like diff-top-k-proofs [26]—which tracks up to $k$ proofs of arbitrary size for each fact—to the GPU, we leverage two insights. First, we observe that the max proof size can be statically determined at compile time. Second, we find that the `diff-top-1-proofs` semiring that tracks just one proof is sufficient for many programs; Lobster could also easily be extended to track larger $k$ as well.

## 3 Language and Compiler

Lobster focuses on accelerating the Datalog back-end with GPU hardware. This poses a challenge, as the process of supporting rich reasoning is at odds with ensuring high performance. To achieve both of these goals, Lobster introduces a new intermediate language, APM (Abstract Parallel Machine), which is designed to simplify the process of compiling and optimizing Datalog programs for GPU execution. We assume an existing Datalog compiler is capable of converting a user-level program to a mid-level program based on relational algebra. From there, Lobster compiles the relational algebra down to an APM program that can be executed on the GPU. In this section, we describe the low-level sequential language APM and present the compilation process from the mid-level relational algebra language to APM.

### 3.1 Background

***Relational Algebra Machine.*** We start by describing our compiler's source language, the Relational Algebra Machine (RAM), which is based on the familiar language of *Relational Algebra* for expressing database queries [2]. The abstract syntax of RAM is shown in Figure 4. At a high level, executing a RAM program $\bar{\phi}$ means sequentially executing each stratum $\phi_1, \ldots, \phi_n$. Within each stratum, rules are iteratively applied to the extensional database (EDB), which contains the input facts, until a fix-point is reached. The newly derived

**Table 1.** A summary of instructions in the APM language. Lowercase latin characters represent registers, while capital latin characters represent scalar integers. We write $\overline{r_n}$ to denote a sequence of $n$ registers and $r_t$ to denote a register containing semiring tags. $\alpha_{n,m}$ denotes a function from $n$-tuples to $m$-tuples, $\sigma$ a function $\tau \times \tau \longrightarrow \tau$ for some type $\tau$, and $\rho$ a relation in the database.

| **Signature** | **Description** |
|---|---|
| $\texttt{alloc}\langle\tau_1,\ldots,\tau_n\rangle(\overline{r_n}, S)$ | Allocate $n$ registers each of size $S$ and types $\tau_1,\ldots,\tau_n$. In practice, the type can be inferred based on usage and is elided. |
| $\overline{d_m} \leftarrow \texttt{eval}\langle\alpha_{n,m}\rangle(\overline{s_n})$ | Evaluate $\alpha$ on each row of $\overline{s_n}$. |
| $\overline{d_n} \leftarrow \texttt{gather}(i, \overline{s_n})$ | Gather rows of $\overline{s_n}$ based on indices $i$. |
| $d \leftarrow \texttt{gather}\langle\alpha_{n,1}\rangle(\overline{i_n}, \overline{s_n})$ | Gather rows of $\overline{s_n}$ based on indices $\overline{i_n}$ and reduce the resulting tuple with $\alpha$. |
| $\texttt{store}\langle\rho\rangle(\overline{s_n}, s_t)$ | Store registers $\overline{s_n}$ and $s_t$ as the columns and tags for relation $\rho$ with arity $n$ in the database. |
| $[\overline{s_n}, s_t] = \texttt{load}\langle\rho\rangle()$ | Loads the columns and tags of relation $\rho$ with arity $n$ from the database into registers $\overline{s_n}$ and $s_t$. |
| $d \leftarrow \texttt{build}(\overline{s_n})$ | Builds a hash index for the table with columns $\overline{s_n}$. |
| $\overline{d_n} \leftarrow \texttt{count}(\overline{b_n}, h, \overline{a_n})$ | Count the number of occurrences of each tuple in the table with columns $\overline{b_n}$ in the table with columns $\overline{a_n}$ via the hash index $h$. |
| $\overline{d_n} \leftarrow \texttt{scan}(s)$ | Computes the exclusive prefix sum of register $s$. |
| $[d_l, d_r] \leftarrow \texttt{join}\langle W\rangle(\overline{b_m}, \overline{a_n}, h, c, o)$ | Produces the resulting indices from a $W$ column join of two tables with columns $\overline{b_m}$ and $\overline{a_n}$ via the hash index $h$, histogram $c$, and histogram prefix sum $o$. |
| $\overline{d_n} \leftarrow \texttt{copy}(\overline{s_n})$ | Copies from register $\overline{s_n}$, truncating if the destination is smaller than the source. |
| $\overline{d_n} \leftarrow \texttt{sort}(\overline{s_n})$ | Lexicographically sorts the table with columns $\overline{s_n}$. |
| $\left[\overline{d_n}, s\right] \leftarrow \texttt{unique}\langle\sigma\rangle(\overline{s_n})$ | Merges adjacent duplicate rows via $\sigma$ from the table with columns $\overline{s_n}$, returning the number of unique elements $s$. |
| $\overline{d_n} \leftarrow \texttt{merge}(\overline{a_n}, \overline{b_n})$ | Merges two lexicographically sorted tables with columns $\overline{a_n}$ and $\overline{b_n}$. |

$$\begin{array}{rlll}
\text{(Predicate)} & \rho & & \\
\text{(Projection Fn.)} & \alpha & & \\
\text{(Selection Fn.)} & \beta & & \\
\text{(Expression)} & \epsilon & ::= & \rho \mid \pi_\alpha(\epsilon) \mid \sigma_\beta(\epsilon) \mid \epsilon_1 \bowtie_n \epsilon_2 \\
& & \mid & \epsilon_1 \cup \epsilon_2 \mid \epsilon_1 \times \epsilon_2 \mid \epsilon_1 \cap \epsilon_2 \\
\text{(Rule)} & \psi & ::= & \rho \leftarrow \epsilon \\
\text{(Stratum)} & \phi & ::= & \{ \psi_1, \ldots, \psi_n \} \\
\text{(Program)} & \overline{\phi} & ::= & \phi_1; \ldots; \phi_n
\end{array}$$

**Figure 4.** The RAM language.

facts form the intensional database (IDB), which accumulates intermediate results produced by the program. Each rule $\rho \leftarrow \epsilon$ consists of a target relation $\rho$ and a query $\epsilon$. This query is a dataflow graph with many sources but only one sink. The operators in the graph are a core fragment of relational algebra operators, comprising project ($\pi$), select ($\sigma$), and join ($\bowtie$) as well as three set operators, union ($\cup$), product ($\times$), and intersect ($\cap$). Note that $\pi$ and $\sigma$ allow taking arbitrary projection or selection functions, while the join operation $\bowtie$ accepts the number of columns to perform join on. For this section, we focus on accelerating a single recursive stratum.

***Provenance Semirings.*** Relational algebra programs can incorporate differentiable or probabilistic reasoning by tagging each fact with additional information such as probabilities or boolean formulas, as shown in prior work [26, 28]. More generally, *provenance semirings* [17] enable programmable semantics that allow tags from an arbitrary semiring. Formally speaking, a provenance semiring $T$ is a 5-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ where $T$ is the space of tags (Figure 5a). $\oplus$ and $\otimes$ dictate how tags are combined through disjunction and conjunction operations. In Figure 5b, we show a few provenance semirings used in the literature [13, 17, 20] for discrete reasoning and approximated probabilistic reasoning. As an example, a tag can be a boolean formula $\phi \in \Phi$ represented in disjunctive normal form (DNF) under set notation. Here, the boolean variables $v$ will be references to facts in the input databases. With probability $\Pr(v)$ attached, one might perform top-$k$ filtering on proofs to avoid blow-up of the boolean formulas. In order to support the discrete, probabilistic, and differentiable modes of reasoning, Lobster implements 7 commonly used provenance semirings, which we elaborate in Section 3.5.

### 3.2 APM: A Language for Parallel Machines

APM is a low-level, assembly-style procedural language that explicitly exposes allocations and is composed exclusively of instructions which permit massively parallel execution. An overview of the instructions is provided in Table 1. APM seeks to solve the problem that, traditionally, GPU programming is much like C programming: an unbounded set of programs can be expressed, even ones that map poorly to

| | | | |
|---|---|---|---|
| (Tag) | $t$ | $\in$ | $T$ |
| (False) | $\mathbf{0}$ | $\in$ | $T$ |
| (True) | $\mathbf{1}$ | $\in$ | $T$ |
| (Disjunction) | $\oplus$ | : | $T \times T \to T$ |
| (Conjunction) | $\otimes$ | : | $T \times T \to T$ |

**(a)** The provenance semiring structure.

| Provenance | $T$ | $\mathbf{0}$ | $\mathbf{1}$ | $\oplus$ | $\otimes$ |
|---|---|---|---|---|---|
| Bool | $\{\bot, \top\}$ | $\bot$ | $\top$ | $\vee$ | $\wedge$ |
| Max-Min-Prob | $[0, 1]$ | 0 | 1 | max | min |
| Top-$k$-Proofs | $\Phi$ | $\{\}$ | $\{\emptyset\}$ | $\vee_k$ | $\wedge_k$ |

**(b)** Common examples of provenance semirings.

**Figure 5.** Provenance semiring structure and examples.

the underlying hardware. APM alleviates this problem by taking the implicit guidelines of the GPU programming model and making them explicit in the design of APM. This results in a desirable property: once a program is compiled to APM, efficient GPU execution is assured. We consider a number of core limitations of the GPU programming paradigm and make a corresponding design decision in APM:

1. **Lockstep Execution** While GPUs have thousands of cores available for parallel computation, these cores are not as flexible as CPU cores. Specifically, GPU cores implement a single instruction, multiple data (SIMD) paradigm, in which a set of 32 threads (known as a *warp*) must execute the same set of instructions while operating over separate thread registers. This informs the **lack of control flow** in APM, ensuring minimal thread divergence.
2. **Allocation** Allocating GPU memory while GPU code is executing has negative performance implications. Therefore, data structures commonly used in database systems that rely on pointer chasing like B-Trees and Tries are non-starters in programs wishing to execute on GPUs. Instead, data structures like sorted arrays, which use large contiguous blocks of memory and can pre-allocate enough memory for their use up-front, are preferred. This restriction is respected by requiring APM programs be in **static single assignment** (SSA) form [12], and by requiring **all registers be explicitly allocated** with a size before use. As a result, compiling to APM requires statically determining memory allocation points, and determining register lifetimes is trivial.
3. **Coalesced Memory** In GPUs, memory accesses are fastest when threads within a warp access consecutive memory locations, a pattern known as coalesced memory access. As such, a columnar representation for relational tables helps ensure maximum utilization of GPU memory bandwidth by ensuring the common path of per-column memory operations results in coalesced accesses. Correspondingly, all registers in APM are **vector registers** that store a non-resizable buffer of identically-typed values.

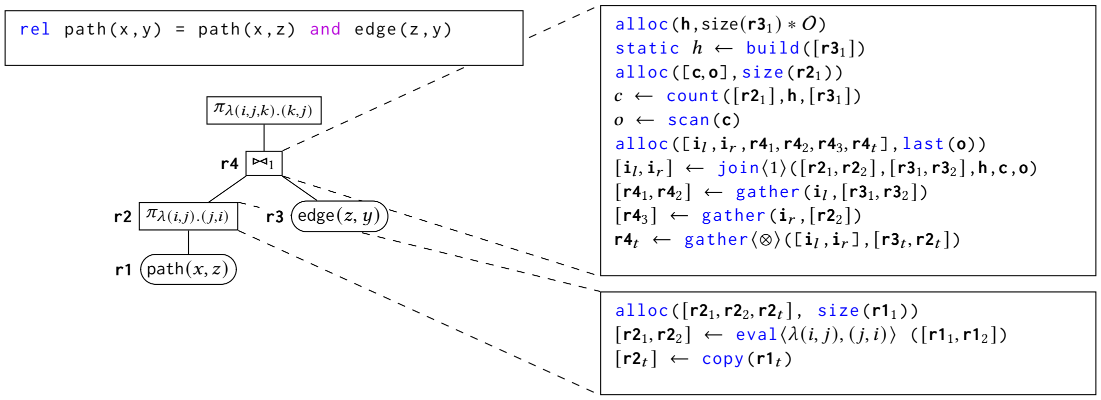
## 3.3 Compiling RAM to APM

Given the design considerations of APM, we now must determine how to compile a RAM program to APM. The most important decision is determining how to represent tables in APM, as they are the base unit of data in relational algebra. As tables consist of a fixed schema of columns of identical size, it is straightforward to represent an arity $n$ relation as $n$ registers of equal size, adding an additional register for provenance semiring tags. It is then sensible to discuss sets of registers in APM as tables, just as we discuss sets of facts in RAM as relations.

With table representation fixed, compiling RAM to APM involves flattening the RAM program (represented as a DAG) into the APM program (represented as a sequential list of instructions). This flattening is implemented via a recursive RAM-to-APM function `compile` :: RAM $\to$ [instr] $\times$ [reg], a function that compiles a RAM expression into a sequence of instructions and returns the registers the result table is stored in. For the complete definition of `compile`, see the Appendix. Translation proceeds in the presence of a translation context $F_T$, also known as the EDB, which contains schema necessary for applying the translations and the provenance for using the proper tag operations. We now examine two of these translation rules in detail to give examples of why the translation to APM is challenging but makes the resulting programs amenable to GPU execution.

***Project.*** Projection is an example of the simplest parallelism Lobster exploits: row-level parallelism. A projection expression $\pi_\alpha(\epsilon)$ consists of a projection expression $\alpha$ evaluated on an input table. Critically, the expression is evaluated identically and without coordination across each fact of the input relation: this is a perfect fit for the SIMD paradigm employed by GPUs. Propagating semiring tags is also straightforward: the provenance of each fact in the output is tied to exactly one fact in the input, so tags can be copied through to the result without modification. Finally, performing allocation of the input facts is also straightforward, as the size of the output relation is identical to that of the input relation. A concrete usage of lowering `project` to APM can be seen in Figure 6, which features the compilation of a simple permutation projection.

***Join.*** Potentially the most important relational operator, join ($\bowtie_n$) forms the computational core of most Lobster programs, so it is important to find an efficient implementation. Unfortunately, it is also more challenging than `project` for two reasons: (1) whereas each input fact in project produces exactly one output fact, with join each input fact can compute zero or more output facts and (2) rather than evaluating an expression against each row, join requires a membership test against the table being joined against. To overcome these obstacles, Lobster takes inspiration from GPU hash-join algorithms [38]. Lobster has an additional requirement, however,


```
rel path(x,y) = path(x,z) and edge(z,y)
```

```
alloc(h,size(r3_1) * O)
static h ← build([r3_1])
alloc([c,o],size(r2_1))
c ← count([r2_1],h,[r3_1])
o ← scan(c)
alloc([i_l,i_r,r4_1,r4_2,r4_3,r4_t],last(o))
[i_l,i_r] ← join⟨1⟩([r2_1,r2_2],[r3_1,r3_2],h,c,o)
[r4_1,r4_2] ← gather(i_l,[r3_1,r3_2])
[r4_3] ← gather(i_r,[r2_2])
r4_t ← gather⟨⊗⟩([i_l,i_r],[r3_t,r2_t])
```

```
alloc([r2_1,r2_2,r2_t], size(r1_1))
[r2_1,r2_2] ← eval⟨λ(i,j),(j,i)⟩ ([r1_1,r1_2])
[r2_t] ← copy(r1_t)
```

**Figure 6.** In this example, we compile a part of the rule shown in Figure 3c (line 6). The code block on the top shows the Datalog rule, while bottom-left illustrates the abstract syntax tree of the RAM program compiled from it. We expand the nodes **r2** and **r4** on the right to show their low-level APM code. We denote with $O$ a configurable parameter that determines the size of the hash table used in the join implementation.

to track provenance correctly in joins, where the provenance of each output fact is the product of the provenance of the input facts. A concrete usage of lowering `join` to APM can be seen in Figure 6, which features the compilation of a join over two binary relations.

### 3.4 Evaluating APM

Once an APM program is compiled, it is executed in a least fix-point iteration manner. Note that the program executes continuously, updating the database each time, until no new facts are discovered. To make this process efficient, Lobster employs a semi-naive evaluation strategy, which is a variant of the traditional naive evaluation strategy that avoids redundant computation.

Succinctly, semi-naive evaluation involves tracking a frontier of recently discovered facts, and only applying rules to frontier facts. This avoids the redundant computation of applying rules to stale facts that are known a priori to not produce new facts. Concretely, the database is partitioned into three sets of facts: delta facts (those that are computed during the current iteration), recent facts (those that were computed in the previous iteration), and stable facts (all other facts). After each iteration, the recent facts are merged with the stable facts and the delta facts become the recent facts. Importantly, these semantics are codified in translation rules (see Appendix), which express semi-naive evaluation in terms of APM instructions. This means the deduplication of facts and the tracking of recent and stable facts is parallelized on the GPU just like the rest of the computation.

### 3.5 Provenance Semiring Framework

As discussed prior, Lobster employs a GPU-accelerated provenance semiring framework with 7 implemented semirings covering discrete, probabilistic, and differentiable reasoning. Specifically, Lobster supports unit, max-min-prob, add-mult-prob, top-1-proof, and the differentiable versions of the probabilistic semirings. Tags in APM are stored as an additional register alongside the column registers. Since the tags may store boolean, floating point, and even complex data structures like dual-numbers and boolean formulae, we need each provenance to specify a fixed size for each tag.

***Limitations.*** Notably, Lobster departs from prior work by not supporting the fully general top-$k$-proof provenance [20], but just the special case of top-1-proof. This special case tracks just one conjunction of boolean variables for each fact, encoding that fact's proof set. During disjunction, the provenance picks the more likely of the two proofs by computing the probabilities for each. For conjunction, the provenance merges the two proofs and ensures that no conflict is present. We find that this special case is sufficient for most practical applications and is much more efficient to compute than the general case. Note that in this formulation, the size limit for a proof needs to be specified ahead of time. We set the limit to 300 which is sufficient for all evaluated benchmarks.

## 4 Optimizations

Section 3 describes the APM language and its value as a principled way to handle Datalog with provenance on GPUs; however, APM shows additional utility as a platform for optimizations. Here, we discuss optimizations from prior works that are easily implementable as APM transformations, as well as novel optimizations enabled by the APM runtime.

### 4.1 Buffer Reuse and Management

Regardless of evaluation strategy, query evaluation produces lots of temporary data, making allocation performance important. Accordingly, every allocation in an APM program is

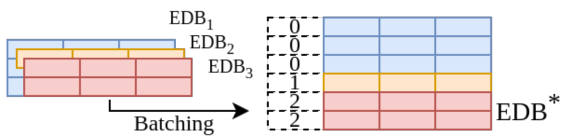


**Figure 7.** How Lobster incorporates batched input data via sample tagging. Distinct colors represent distinct samples within a batch.

identified by an `alloc` instruction. Due to the lack of branching and looping constructs, each iteration through an APM program produces a fixed number of allocations. This enables two optimizations: arena allocation and buffer reuse.

***Arena Allocation.*** All data in registers is discarded after each iteration of an APM program. This fixed lifetime guarantee makes arena allocation [18] a good fit for APM. This observation makes allocations and deallocations in APM essentially zero-cost, as allocation is reduced to bumping a pointer and deallocation is a no-op.

***Buffer Reuse.*** When arena allocation is impossible due to memory limits, allocation cost can still be amortized via buffer reuse. Prior work on GPU Datalog execution made a similar discovery [44], noting that it was expensive to allocate a specific buffer required for merging relations after each iteration. To address this, the authors over-allocate that buffer initially and reuse it across iterations. Since each APM buffer is identified by an `alloc` instruction, it is trivial to implement this optimization in APM not just for a specific buffer, but for each buffer in the program. This optimization is surprisingly effective, since the size of each register is strongly correlated with its size on the previous iteration.

### 4.2 Hash Index Reuse via Static Registers

Using a hash-based join is a boon for GPU acceleration, but building a hash index during each iteration of the fix-point loop negatively impacts performance. To resolve this, we observe that frequently at least one input to a join is an EDB relation, which is constant across iterations. Formally, Datalog programs are said to be "linear recursive" if each join has at most one IDB input, and we find that nearly all programs we consider are linear recursive. In these cases, we can build a hash index during the first iteration of the fix-point loop and reuse it during successive iterations.

To realize this, we introduce the concept of static registers, modeled after the `static` keyword in C. Static registers are initialized once and retain their values across iterations. Marking the result of `build` as `static` when compiling `join` ensures that the hash index is reused across iterations.

### 4.3 Batched Evaluation

A key component of deep learning is grouping samples into *batches* of samples that can be processed by the model in a single pass. To truly integrate Lobster with deep learning as an end-to-end neurosymbolic tool, it should be aware of batching and able to process batches effectively. Surprisingly, batching is a straightforward extension of the existing semantics. Given a program $\overline{\phi}$ evaluated against a batch of three databases $F_1$, $F_2$, and $F_3$, we seek a database $F^*$ such that evaluating $\overline{\phi}$ against $F^*$ provides equivalent information to evaluating $\overline{\phi}$ against $F_1$, $F_2$, and $F_3$ separately.

We can construct $F^*$ without adding new constructs to APM or RAM by taking our APM columnar representation of a table and adding a new register $r_s$ to the front of the table that records the sample id. Tables are now represented as a register pack $[r_s, r_1, \ldots, r_n, r_t]$, including the sample id, the columns, and the provenance semiring tags. After execution, each IDB fact will have a sample tag which can be used to disambiguate results into per-sample databases that are returned to the user. This is illustrated in Figure 7.

Some desirable ramifications that naturally arise from this framing of batching are (1) facts from separate batches cannot be joined together, so long as the width of each join operator is extended by one to include the batch tag; (2) parallelizing over each element of the batch is implicit, as the runtime already parallelizes over the rows of a relation; and (3) the additional memory footprint is minimal, since batches are seldom larger than 256 samples, meaning sample tags only take one byte of memory per row.

## 5 Implementation

We build Lobster with a mixture of Rust, C++, and CUDA, reusing the front-end and query planner of Scallop to limit the scope of implementation. Lobster comprises approximately 2,000 new lines of Rust code and 9,000 new lines of CUDA/C++. We now discuss implementation details that fall outside the scope of the theory of Lobster's core compiler and runtime, yet are of practical interest and importance for implementing Lobster.

### 5.1 Hash Table Design

Crucial to Lobster's GPU-accelerated join algorithm is the existence of a lock-free, GPU hash table supporting parallel insertions and lookups. While many implementations are possible, ours is inspired by previous work in [38]. Namely, we adopt open-addressing with linear probing for collision resolution, enabling a contiguous memory representation with no indirection. Unlike prior work, Lobster supports reasoning over relations with arbitrary width, so rather than storing fact data directly in the hash table, we build hash *indices* that map back to a row of the source table. While this necessitates an additional random memory access to resolve collisions, it decouples the time and space complexity of the join from the width of the input relations.

### 5.2 Bytecode Interpreter for Expression Evaluation

Projection operations are pervasive in Datalog programs, yet generally account for a small portion of the runtime compared to operators like `join` due to projection's algorithmic simplicity. Nonetheless, Lobster's optimizes the handling of `project` operations for GPU execution. Specifically, there are two code paths for the implementation of `eval` in APM. Project expressions that permute or subset the columns of the relation can be evaluated as a series of columnar memory copies. Project expressions that contain arithmetic or comparison of tuple elements are compiled to bytecode for a simple stack machine, and each GPU thread executes this bytecode program against one fact with a small fixed-size stack residing in thread-local memory.

### 5.3 Scheduling Stratum Offloading

Lobster relations start their life in CPU memory and once in GPU memory it is advantageous to continue operating on them with the GPU. To avoid the impact of high-latency CPU-GPU memory transfers, Lobster adopts an intelligent strategy for scheduling data transfers.

Lobster's begins by transferring facts to the GPU before the longest-running stratum (identified via a heuristic based on counting recursive joins) and back to the CPU after that stratum. From that longest-running stratum, we expand forwards and backwards in the static data-dependency graph to encompass adjacent strata as well, until the size of the stratum's inputs and outputs is small. Adopting a min-cut-like approach to GPU scheduling avoids spending excessive time in CPU-GPU transfers.

### 5.4 Other Forms of Parallelism

The primary parallelism exploited by Lobster is within each relational operator, and there is no parallelism in the execution of separate relational operators (they are executed sequentially). While the idea of parallelizing or pipelining operators is appealing, since Lobster's design requires very little CPU-GPU data movement, there is no opportunity to overlap data transfer with computation, so no performance improvement is achieved.

## 6 Evaluation

We empirically evaluate Lobster with the goal of demonstrating how well it performs in both training and inference tasks, and for the latter we explore a range of benchmarks that require differentiable, probabilistic, or discrete reasoning.

In the following sections, we introduce the benchmark tasks (Section 6.1) and the chosen baselines (Section 6.2) and present results in Sections 6.3 and 6.4. All benchmarks are run on a machine with two 20-core Intel Xeon CPUs, a GeForce RTX 2080 Ti GPU, and 768 GB RAM, with the exception of the discrete benchmarks, which have higher VRAM requirements and were run on a machine with two 24-core Intel Xeon CPUs, 1.5 TB RAM, and an NVIDIA A100 GPU with 80 GB VRAM.

### 6.1 Benchmarks

We evaluate Lobster on a suite of ten benchmark tasks summarized in Table 2. Since Lobster is built on a flexible framework of provenance semirings, it supports differentiable, probabilistic, and discrete reasoning. Correspondingly, we pick tasks across each of these reasoning modes to better illustrate the tradeoffs inherent in providing this flexibility. The tasks span diverse application domains: natural language processing (CLUTRR), image processing (Pathfinder and HWF), program reasoning (Probabilistic Static Analysis), bioinformatics (RNA SSP), planning (PacMan-Maze), and graph databases (Transitive Closure, Same Generation, and CSPA).

The table describes each task's input, the functionality of the logic program, the kind of reasoning involved, and the number of rules. The tasks requiring differentiable reasoning are taken from Scallop's evaluation (although we omit some tasks that do not have an obvious notion of scalability), the probabilistic reasoning tasks are crafted by us inspired by problems from the literature [40, 42], and the discrete reasoning tasks mirror the evaluation of the latest work in this space, FVLog [43].

Notably, our results focus exclusively on performance, with no mention of correctness. This is because in all cases each system under test produces identical results: for differentiable tasks, the model reaches identical accuracy whether Scallop or Lobster is used, and for probabilistic and discrete tasks, the logic programs are identical and therefore produce identical results. We refer curious readers who desire a more thorough discussion of comparing accuracy between pure-neural and neurosymbolic models to the Scallop paper [27], the results of which we partially replicate here with Lobster.

We next briefly describe each of the tasks.

**Pathfinder** This task is discussed in-depth in Section 2 and requires reasoning over an image to determine if two dots are connected by a sequence of dashes.

**PacMan-Maze** In this task, a neurosymbolic reinforcement learning agent aims to solve a 2D maze given only an image of the maze. The neural portion executes a CNN to predict enemy locations in the maze, and the symbolic portion plans a safe path to the goal. Our formulation leverages curriculum learning: the agent first learns in a 5-by-5 maze and then moves to a 20-by-20 grid.

**HWF** The Handwritten Formula (HWF) [24] task requires parsing and evaluating a formula of handwritten digits and operators, given supervision only on the final value. The dataset consists of formulas of varying length, meaning naive parallelism strategies like processing each formula in a batch

**Table 2.** Characteristics of benchmark tasks.

| Task | Input | Logic Program | Kind | #Rules | Provenance |
|---|---|---|---|---|---|
| Pathfinder | Image | Check if two dots are connected by a sequence of dashes. | Diff. | 2 | diff-top-1-proofs |
| PacMan-Maze | Image | Plan optimal next step by finding safe path from actor to goal. | Diff. | 14 | diff-top-1-proofs |
| HWF | Images | Parse and evaluate formula over recognized symbols. | Diff. | 13 | diff-top-1-proofs |
| CLUTTR | Text | Deduce kinship by recursively applying composition rules. | Diff. | 3 | diff-top-1-proofs |
| Prob. Static Analysis | Code | Compute alarms with severity via probabilistic static analysis. | Prob. | 28 | minmaxprob |
| RNA SSP | RNA | Parse an RNA sequence according to a context-free grammar. | Prob. | 28 | prob-top-1-proofs |
| Transitive Closure | Graph | Compute transitive closure of a directed graph. | Disc. | 2 | unit |
| Same Generation | Graph | Compute graph vertices that are in the "same generation". | Disc. | 2 | unit |
| CSPA | Graph | A context sensitive pointer analysis. | Disc. | 10 | unit |

separately will fall short due to work imbalances. Further, the symbolic program requires support for floating-point data and floating-point arithmetic operations.

**CLUTRR** CLUTRR is a natural language reasoning task about family kinship relations [39]. The input contains a natural language passage about a family with each sentence in the passage hinting at kinship relations. The goal is to infer the relationship between a given pair of characters; however, the target relation is not stated explicitly in the passage and it must be deduced through a reasoning chain. The most difficult problem in the evaluation dataset requires reasoning through a chain of length 10.

**Probabilistic Static Analysis** This benchmark extends static program analysis with probabilistic inputs. Specifically, analysis inputs are annotated with probabilities to reflect the system's confidence. These probabilities are propagated to the output and used to rank results in order to decrease the visibility of false positives [42].

**RNA SSP** This task performs RNA Secondary Structure Prediction (SSP) using the ArchiveII [40] dataset. RNA SSP discovery is of widespread interest in the medical community, as the secondary structure of RNA molecules is crucial for understanding their function. Our neurosymbolic solution uses a Datalog program to parse an RNA sequence according to a context-free grammar, given probabilistic input from a transformer model. The dataset consists of a set of 475 RNA sequences of length 28 to 175.

**Transitive Closure** This benchmark computes the reachability of nodes in a graph using discrete reasoning. We use graphs from the SNAP graph repository [23] that take at least 1 second to process.

**Same Generation** This benchmark computes which nodes in a directed graph are the same distance from a common ancestor, i.e. which nodes are in the "same generation". We use the same graphs as in the Transitive Closure benchmark.

**Context Sensitive Pointer Analysis (CSPA)** This benchmark computes a context sensitive pointer analysis for a program. We mirror the evaluation of GDLog [44] both in the Datalog program that we use and the input graphs.

## 6.2 Baselines

We compare Lobster to several other systems, as shown previously in Figure 2. While no prior system matches Lobster's feature set, comparisons to more limited systems help us gauge Lobster's performance across a range of use-cases.

**Scallop** supports differentiable reasoning with provenance semirings like Lobster, but supports only batch-level CPU multicore parallelism and therefore struggles to scale with problem and data complexity. We do not evaluate on DeepProbLog [29], a similar CPU-only system supporting differentiable and probabilistic reasoning, as previous work has shown that Scallop's performance is uniformly superior [27].

**ProbLog** [13] provides discrete and probabilistic reasoning, but does not support GPU acceleration or CPU multithreading. Notably, ProbLog executes logic programs via stable model semantics, indicating it may find certain programs easier or harder than other baselines, which are all based on bottom-up search. Additionally, while Problog has been used to study approximate probabilistic reasoning [35], approximate inference is not implemented in the publicly released Problog. Therefore, in our experiments Problog performs exact inference, as opposed to the more scalable approximate inference that Lobster and Scallop perform.

**FVLog** [43] supports only discrete reasoning, but does leverage GPU acceleration. FVLog is specialized for a different sort of workload than Lobster: it targets large batch analysis jobs that may span minutes, whereas Lobster emphasizes running the same program multiple times as a component of a neurosymbolic model. Notably, FVLog does not offer a Datalog front-end and query planner, meaning that all FVLog programs are human-written, low-level, relational algebra programs. As FVLog is faster than prior GPU discrete Datalog systems (like GDLog [44]), we compare only to FVLog for brevity.

**Soufflé** [36] is a state-of-the-art, multicore, CPU Datalog engine. While it does not support differentiable or probabilistic reasoning, comparing with Soufflé helps reveal the benefits of GPU acceleration versus CPU optimizations.

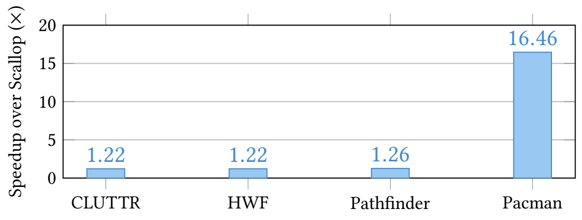


**Figure 8.** Lobster's speedup over Scallop on training tasks.

## 6.3 Lobster for Training

To evaluate the extent to which Lobster improves performance in the training pipeline, we compare the total training time of Lobster against Scallop, the only other system that supports neurosymbolic training. For each task, training is run until convergence rather than a pre-determined number of epochs, and therefore takes a task-specific number of epochs. However, for a given task, both Lobster and Scallop take the same number of epochs, as they are executing the same Datalog program and produce identical results.

The results in Figure 8 reveal that Lobster can achieve significant speedups in end-to-end training time compared to Scallop, ranging from 1.2x to 16x. Figure 8 includes the cost of neural computations, which are already heavily optimized on GPU hardware via Pytorch [33] and unaffected by Lobster, so Amdahl's Law limits the potential end-to-end speedup. Pacman is an exception as it performs extensive symbolic computation which Lobster can greatly accelerate.

## 6.4 Lobster for Inference

Beyond training of neurosymbolic models, we also evaluate Lobster's performance on a range of neurosymbolic inference tasks. In neurosymbolic inference, the neural component is pre-trained. For example, with Pacman a pre-trained neural classifier identifies in-game objects, but the symbolic program still needs to be run to determine the path to the goal for each game board. We report average times across a set of samples: CLUTRR processes relationship graphs from 13 text passages, HWF evaluates 160 formulas of length 13, Pathfinder is evaluated on a set of 1216 images, and PacMan-Maze solves 50 mazes on a 15x15 grid.

Figure 9 shows that Lobster can obtain significant speedups over Scallop. Compared to results from training (Figure 8), during inference benchmarks like CLUTTR and Pathfinder spend significantly less time in neural computation which leads to larger speedups with Lobster. Notably, the speedups for Pacman is less for inference than for training. We believe this is because the trained model solves each maze in fewer steps, making the symbolic computation a smaller fraction of the total runtime.

Next we explore how well Lobster scales to larger problem sizes. We choose the benchmarks that can be scaled most naturally: for Pathfinder we increase the resolution of the analysis and for Pacman the maze size. We choose

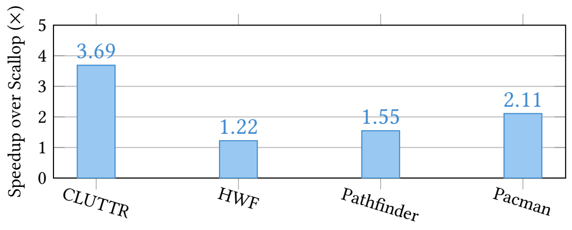
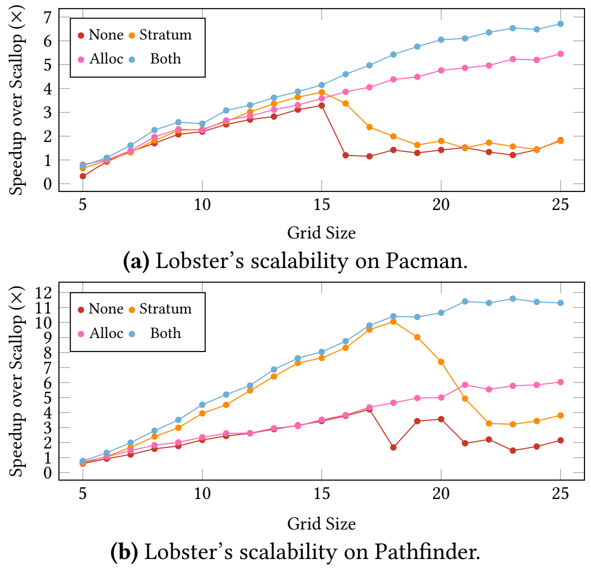


**Figure 9.** Lobster's speedup over Scallop on neurosymbolic inference tasks.



**(a)** Lobster's scalability on Pacman.

**(b)** Lobster's scalability on Pathfinder.

**Figure 10.** Lobster's scalability on Pathfinder and Pacman in the presence of various optimizations. "None" indicates no optimizations, "Stratum" includes the stratum scheduling heuristic of Section 5.3, "Alloc" includes the allocation optimizations of Section 4.1, and "Both" includes both optimizations.

Scallop as the baseline as it is the only system that supports these neurosymbolic workloads. Further, we only consider the symbolic computation time to more precisely measure Lobster's improvement. In Figure 10 we see that Lobster handles large problem sizes better than Scallop, although the speedup plateaus as the problem size becomes large enough to saturate GPU memory bandwidth. Figure 10 also shows an ablation study that justifies some of the optimizations in Lobster—without allocation optimization and stratum scheduling, performance rapidly degrades to near equal with Scallop on problem sizes greater than 20.

Beyond neurosymbolic workloads, we also run two **probabilistic** inference workloads: Probabilistic Static Analysis (PSA) and RNA Secondary Structure Prediction (RNA SSP). As these two workloads require only probabilistic (not differentiable) reasoning, we attempted to run them with ProbLog

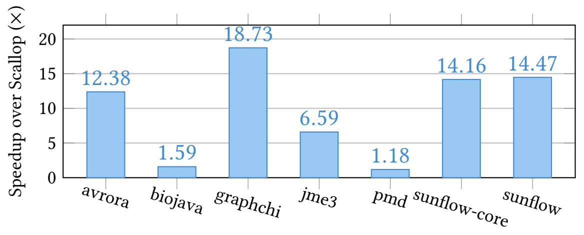


**Figure 11.** Lobster's speedup over Scallop on Probabilistic Static Analysis.

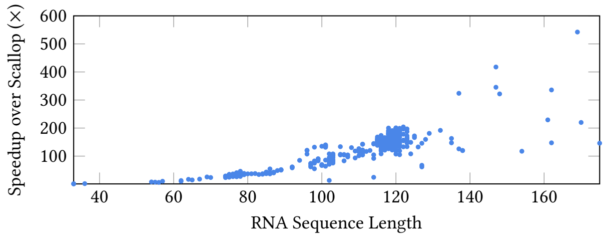


**Figure 12.** Lobster speedup over Scallop on RNA SSP.

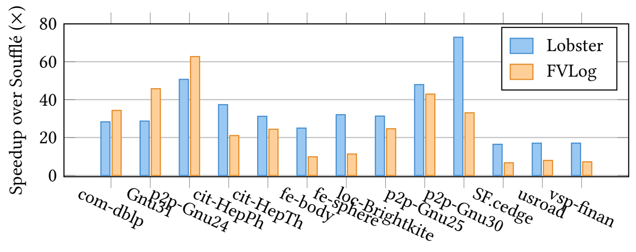


**Figure 13.** Speedup over Soufflé on Transitive Closure.

[13]. However, all of the PSA and RNA SSP runs hit our 2-hour timeout, except for PSA on sunflow-core which was 60% slower than Scallop and 30x slower than Lobster. As stated in Section 6.2, we believe this is explained by ProbLog's exact probabilistic inference.

Figure 11 shows that, on the PSA benchmark, Lobster again offers significant speedups over Scallop when performing static analysis across a range of source programs. With RNA SSP (Figure 12), on the very shortest sequence (28 base pairs) Lobster is 40% slower than Scallop. However, on all of the other sequences Lobster achieves a speedup, frequently by two orders of magnitude. The speedup correlates with sequence length, making Lobster even more valuable on longer sequences which are, generally, of greater biological interest.

Finally, we examine Lobster's performance on **discrete reasoning** tasks that require neither differentiable nor probabilistic reasoning. We first run the Transitive Closure benchmark from FVLog [43] on a range of input graphs [23]. For these inputs, ProbLog always hit our 2-hour timeout, and Scallop (when not timing out) has a 30-90x slowdown over Soufflé, so we omit results for those systems. Figure 13 shows

**Table 3.** The runtime of Lobster versus FVLog on the Same Generation task. "OOM" indicates that the system ran out of memory.

| Dataset | Lobster (s) | FVLog (s) |
|---|---|---|
| fe-sphere | **3.91** | 12.99 |
| CA-HepTH | **2.16** | 6.40 |
| ego-Facebook | **0.53** | OOM |
| Gnu31 | OOM | OOM |
| fe_body | **10.17** | 21.17 |
| loc-Brightkite | **1.45** | OOM |
| SF.cedge | **14.01** | 23.72 |
| com-dblp | OOM | OOM |
| usroad | OOM | OOM |
| fc_ocean | **2.17** | 4.67 |
| vsp_finan | OOM | **90.10** |

**Table 4.** The runtime of Lobster versus FVLog on the CSPA task.

| Dataset | Lobster (s) | FVLog (s) |
|---|---|---|
| httpd | 3.61 | **2.57** |
| linux | **1.81** | 3.91 |
| postgres | **3.32** | 4.39 |

results for Lobster and FVLog and demonstrates that, despite Lobster's generality, it offers competitive performance even against more specialized systems, consistently beating the CPU-only Soufflé and often surpassing the GPU-accelerated FVLog. We attribute this to Lobster's adoption of APM. FVLog in particular lacks any IR and thus forgoes the opportunities afforded by IR-level optimizations (Section 4).

Since FVLog is the sole competitive system, we only run the remaining two discrete tasks, Same Generation and CSPA, on Lobster and FVLog, displaying the results in Table 3 and Table 4 respectively. We observe that for the Same Generation task, Lobster is at least twice as fast on each dataset and that there are multiple datasets that Lobster processes which FVLog runs out of memory on. The one exception is the vsp_finan dataset, where Lobster runs out of memory while FVLog finishes in 90 seconds. We believe this is due to the fact that Lobster is more general and therefore requires more memory to store intermediate results. For the CSPA task, Lobster and FVLog are approximately matched, with Lobster exhibiting a geometric mean speed up of 1.27x over FVLog.

## 7 Related Work

While there is a wealth of work on GPU-acceleration for SQL in both research and industry (e.g., [19, 34]), we focus our related work discussion on systems for logic programming beyond SQL. We relate Lobster to works along three directions: accelerated Datalog engines, probabilistic and differentiable programming, and neurosymbolic methods.

**High-Performance Datalog** A variety of Datalog-based systems have been built for program analysis [14, 21, 36] and even enterprise database applications [3], though these systems run exclusively on the CPU. The FVLog system [38, 44] provides a Datalog engine implemented for GPUs, but it lacks support for the probabilistic and differentiable reasoning needed for deep learning integration. Moreover, FVLog focuses on the domain of large analytics queries, which emphasizes simpler queries executed against large databases, which is not a focus for Lobster. Notably, FVLog does not come with a query planner and user-facing frontend, meaning that users need to directly interact with low-level relational algebra operations supported by the system.

**Probabilistic and Differentiable Programming** *Probabilistic programming* allows programmers to model distributions and perform probabilistic sampling and inference [5, 13, 16, 46]. *Differentiable programming* systems allow programmers to write code that is differentiable and therefore amenable to use during neural network training. Symbolic and automatic differentiation [4] are commonly used in popular ML frameworks such as PyTorch and others [1, 15, 33].

Probabilistic programs are not in general differentiable and thus cannot be run during training. The differentiable programming systems described above are designed for real-valued functions and are not compatible with logic programming. Lobster, on the contrary, focuses on the differentiability of logic programs with probabilities.

**Neurosymbolic Methods** The emerging domain of neurosymbolic computation combines symbolic reasoning into existing data-driven learning systems. There have been a large number of successful neurosymbolic systems across a range of machine learning domains like computer vision and natural language processing [7–11, 24, 28–32, 37, 41, 47, 49–53]. Lobster builds upon the Scallop neurosymbolic programming language [20, 26], as Scallop is general enough to implement other neurosymbolic systems [8, 30, 49, 50]. However, Lobster improves upon the CPU-only Scallop by using GPU acceleration to provide higher performance and the ability to scale to larger datasets, as we demonstrated in Section 6.

## 8 Conclusion

We have described the design and implementation of the Lobster neurosymbolic engine. With existing engines, symbolic computation can quickly become the bottleneck when neural computations benefit from domain-specific hardware accelerators like GPUs. Lobster shows how Datalog programs can also take advantage of GPUs, providing large speedups and strong scalability over CPU-only engines like Scallop.

## Acknowledgments

We would like to thank our shepherd, Professor Michael O'Boyle. This paper is based upon work supported by the National Science Foundation under Grant No. 2313010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc.

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *ACM International Conference on Management of Data (SIGMOD).* doi:10.1145/2723372.2742796

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. 2015. Automatic Differentiation in Machine Learning: a Survey. (2015). arXiv:1502.05767

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2018).

[6] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. 2021. Neurosymbolic Programming. *Foundations and Trends in Programming Languages* 7, 3 (2021). doi:10.1561/2500000049

[7] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web Question Answering with Neurosymbolic Program Synthesis. In *ACM International Conference on Programming Language Design and Implementation (PLDI).* doi:10.1145/3453483.3454047

[8] Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V. Le. 2020. Neural Symbolic Reader: Scalable Integration of Distributed and Symbolic Representations for Reading Comprehension. In *International Conference on Learning Representations (ICLR).*

[9] Zeming Chen, Qiyue Gao, and Lawrence S Moss. 2021. NeuralLog: Natural language inference with joint neural and logical reasoning. *arXiv preprint arXiv:2105.14167* (2021).

[10] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding Language Models in Symbolic Languages. (2022). arXiv:2210.02875

[11] William W. Cohen, Fan Yang, and Kathryn Rivard Mazaitis. 2017. TensorLog: Deep Learning Meets Probabilistic DBs. arXiv:1707.05390

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320

[13] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. ProbLog2: Probabilistic Logic Programming. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD).* doi:10.1007/978-3-319-23461-8_37

[14] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-up in-memory datalog

processing: observations and techniques. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 695–708. doi:10.14778/3311880.3311886

[15] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. In *SysML*. https://mlsys.org/Conferences/doc/2018/146.pdf

[16] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics, (AISTATS)*.

[17] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *ACM Symposium on Principles of Database Systems (PODS)*. doi:10.1145/1265530.1265535

[18] David R. Hanson. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience* 20, 1 (1990), 5–12. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380200104 doi:10.1002/spe.4380200104

[19] heavydb 2024. Heavy.AI. https://www.heavy.ai.

[20] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. 2021. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[21] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. doi:10.1162/neco.1989.1.4.541

[23] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[24] Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. 2020. Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning. In *International Conference on Machine Learning (ICML)*. doi:10.48550/arXiv.2006.06649

[25] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv:2405.17238 [cs.CR] https://arxiv.org/abs/2405.17238

[26] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1463–1487.

[27] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proc. ACM Program. Lang.* 7, PLDI, Article 166 (jun 2023), 25 pages. doi:10.1145/3591280

[28] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. Deepproblog: Neural Probabilistic Logic Programming. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[29] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2021. Neural Probabilistic Logic Programming in DeepProbLog. *Artificial Intelligence* 298 (2021). doi:10.1016/j.artint.2021.103504

[30] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. (2019). arXiv:1904.12584

[31] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584* (2019).

[32] Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. 2020. Learning Reasoning Strategies in End-to-End Differentiable Proving. In *International Conference on Machine Learning (ICML)*. arXiv:2007.06477

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems (NeurIPS)*. arXiv:1912.01703

[34] pgstrom 2024. PG-Strom. https://github.com/heterodb/pg-strom.

[35] Joris Renkens, Guy Van den Broeck, and Siegfried Nijssen. 2012. k-Optimal: A Novel Approximate Inference Algorithm for ProbLog. In *Inductive Logic Programming*, Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–38.

[36] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *International Conference on Compiler Construction (CC)*. doi:10.1145/2892208.2892226

[37] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[38] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2023. Towards Iterative Relational Algebra on the GPU. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 1009–1016. https://www.usenix.org/conference/atc23/presentation/shovon

[39] Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. 2019. CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text. (2019). arXiv:1908.06177

[40] Michael F. Sloma and David H. Mathews. 2016. Exact calculation of loop formation probability identifies folding motifs in RNA secondary structures. *RNA* 22 (2016), 1808 – 1818. https://api.semanticscholar.org/CorpusID:365048

[41] Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. 2024. Data-Efficient Learning with Neural Programs. *arXiv preprint arXiv:2406.06246* (2024).

[42] Leo St. Amour and Eli Tilevich. 2024. Toward Declarative Auditing of Java Software for Graceful Exception Handling. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 90–97.

[43] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. 2025. Column-Oriented Datalog on the GPU. arXiv:2501.13051 [cs.DB] https://arxiv.org/abs/2501.13051

[44] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2024. Modern Datalog on the GPU. arXiv:2311.02206 [cs.DB] https://arxiv.org/abs/2311.02206

[45] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. (2020). arXiv:2011.04006

[46] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. arXiv:1809.10756

[47] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. 2019. SATNet: Bridging Deep Learning and Logical Reasoning Using a Differentiable Satisfiability Solver. In *International Conference on Machine Learning (ICML)*. arXiv:1905.12149

[48] Yinjun Wu, Mayank Keoliya, Kan Chen, Neelay Velingker, Ziyang Li, Emily J Getzen, Qi Long, Mayur Naik, Ravi B Parikh, and Eric Wong. 2024. DISCRET: Synthesizing Faithful Explanations For Treatment Effect Estimation. In *International Conference on Machine Learning (ICML)*.

[49] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. 2018. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. In *International Conference on Machine Learning (ICML)*. arXiv:1711.11157

**Algorithm 1:** How to execute a RAM program via compilation to and execution of APM.

```
Data: Database $F_T$, RAM program $\overline{\phi}$.
Result: $F_T$ updated to reflect the result of evaluating $\overline{\phi}$.
for $\phi$ in $\overline{\phi}$ do
    instructions ← compile($\phi$);
    $F_T^{\text{stable}}, F_T^{\text{recent}}, F_T^{\Delta} \leftarrow \emptyset, F_T, \emptyset$;
    $size \leftarrow |F_T^{\text{stable}}|$;
    while true do
        for $i$ in instructions do
            execute $i$
        $size_{\text{new}} \leftarrow |F_T|$;
        if $size_{\text{new}} = size$ then
            break;
        $size \leftarrow size_{\text{new}}$;
    $F_T \leftarrow F_T^{\text{stable}}$
```

[50] Ziwei Xu, Yogesh S Rawat, Yongkang Wong, Mohan Kankanhalli, and Mubarak Shah. 2022. Don't Pour Cereal into Coffee: Differentiable Temporal Logic for Temporal Action Segmentation. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[51] Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Neurasp: Embracing neural networks into answer set programming. *arXiv preprint arXiv:2307.07700* (2023).

[52] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. 2018. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[53] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2023. Parsel: A (De-)compositional Framework for Algorithmic Reasoning with Language Models. arXiv:2212.10561

[54] Xi Zheng, Ziyang Li, Ivan Ruchkin, Ruzica Piskac, and Miroslav Pajic. 2025. NeuroStrata: Harnessing Neurosymbolic Paradigms for Improved Design, Testability, and Verifiability of Autonomous CPS. arXiv:2502.12267 [cs.SE] https://arxiv.org/abs/2502.12267

## A RAM to APM Translation

We detail the `compile` function used to translate RAM to APM in Figure 14.

## B APM Evaluation

Details on executing a RAM program via compilation to APM are provided in Algorithm 1.

| **Compile Expression** | **Instructions** |
|---|---|
| compile($\pi_{\alpha_{n,m}}(\epsilon), D$) *"Project"* | **let** $([s_1, \ldots, s_n, s_t], c)$ = compile$(\epsilon, D)$ **in**<br>$(c \cdot \{$<br>$[d_1, \ldots, d_m, d_t] \leftarrow$ `alloc`(size$(s_1)$)<br>$[d_1, \ldots, d_m] \leftarrow$ `eval`$\langle \alpha_{n,m} \rangle([s_1, \ldots, s_n])$<br>$d_t \leftarrow$ `copy`$(s_t)$ }, $[\overline{d_n}, d_t]$) |
| compile($\rho(x_1, \ldots, x_n), D$) *"Relation"* | ({`alloc`($[s_1, \ldots, s_n, s_t]$, size$(D(\rho))$)<br>$[\overline{s_n}, s_t] \leftarrow$ `load`$\langle D(\rho) \rangle()$ }, $[\overline{s_n}, s_t]$) |
| compile($p \leftarrow \epsilon, D$) *"Update"* | **let** $([s_1, \ldots, s_n, s_t], c)$ = compile$(\epsilon)$ **in**<br>$(c \cdot \{$<br>`store`($p$, $[s_1, \ldots, s_n, s_t]$) }, $\emptyset$) |
| compile($p_1 \leftarrow \epsilon_1, \ldots, p_n \leftarrow \epsilon_n$) *"Stratum"* | **let** $(\_, c_1)$ = compile$(p_1 \leftarrow \epsilon_1)$ **in**<br>…<br>**let** $(\_, c_n)$ = compile$(p_n \leftarrow \epsilon_n)$ **in**<br>$(c_1 \cdot \ldots \cdot c_n \cdot \{$<br>`alloc`($\overline{s_n}$, size$(F_T^{\text{stable}})(\rho)$)<br>$\overline{s_n} \leftarrow$ `load`$\langle F_T^{\text{stable}}(\rho) \rangle()$<br>`alloc`($\overline{r_n}$, size$(F_T^{\text{recent}})(\rho)$)<br>$\overline{r_n} \leftarrow$ `load`$\langle F_T^{\text{recent}}(\rho) \rangle()$<br>`alloc`($\overline{d_n}$, size$(F_T^{\Delta}(\rho))$)<br>$\overline{d_n} \leftarrow$ `load`$\langle F_T^{\Delta}(\rho) \rangle()$<br>`alloc`($\overline{s_n^{\text{new}}}$, size$(\overline{s_n} + \overline{r_n})$)<br>$\overline{s_n^{\text{new}}} \leftarrow$ `merge`$(\overline{s_n}, \overline{r_n})$<br>`alloc`($[\overline{d_n^{\text{sorted}}}, \overline{d_n^{\text{unique}}}]$, size$(\overline{d_n})$)<br>$\overline{d_n^{\text{sorted}}} \leftarrow$ `sort`$(\overline{d_n})$<br>$\overline{d_n^{\text{unique}}} \leftarrow$ `unique`$(\overline{d_n^{\text{sorted}}})$<br>`store`$\langle F_T^{\text{stable}}(\rho) \rangle(\overline{s_n^{\text{new}}})$<br>`store`$\langle F_T^{\text{recent}}(\rho) \rangle(\overline{d_n^{\text{unique}}})$<br>`store`$\langle F_T^{\Delta}(\rho) \rangle(\emptyset)$ } **for** $\rho$ **in** unique$(\rho_1, \ldots, \rho_n)$<br>, $\emptyset$) |
| compile($\epsilon_1 \bowtie_w \epsilon_2, D$) *"Join"* | **let** $(\overline{w_n}, c_1)$ = join$_{\text{impl}}(\epsilon_1 \bowtie_w \epsilon_2, F_T^{\text{stable}}, F_T^{\text{recent}})$ in<br>**let** $(\overline{x_n}, c_2)$ = join$_{\text{impl}}(\epsilon_1 \bowtie_w \epsilon_2, F_T^{\text{recent}}, F_T^{\text{stable}})$ in<br>**let** $(\overline{y_n}, c_3)$ = join$_{\text{impl}}(\epsilon_1 \bowtie_w \epsilon_2, F_T^{\text{recent}}, F_T^{\text{recent}})$ in<br>({ `alloc`($\overline{z_n}$, size$(w_1)$ + size$(x_1)$ + size$(y_1)$)<br>$\overline{z_n} \leftarrow$ `append`$(\overline{w_n}, \overline{x_n}, \overline{y_n})$ }, $\overline{z_n}$) |
| join$_{\text{impl}}(\epsilon_1 \bowtie_w \epsilon_2, D_1, D_2)$ | **let** $([a_1, \ldots, a_n, a_t], c_1)$ = compile$(\epsilon_1, D_1)$ **in**<br>**let** $([b_1, \ldots, b_m, b_t], c_2)$ = compile$(\epsilon_2, D_2)$ **in**<br>$(c_1 \cdot c_2 \cdot \{$<br>`alloc`($h$, size$(a_1) * O$)<br>`static` $h \leftarrow$ `build`($[a_1, \ldots, a_w]$)<br>`alloc`($[c, o]$, size$(b_1)$)<br>$c \leftarrow$ `count`($[b_1, \ldots, b_w]$,h,$[a_1, \ldots, a_w]$)<br>$o \leftarrow$ `scan`$(c)$<br>`alloc`($[i_l, i_r, d_1, \ldots, d_{n+m-w}, d_t]$,last(o))<br>$[i_l, i_r] \leftarrow$ `join`$\langle w \rangle(\overline{b_m}, \overline{a_n}$,h,c,o)<br>$[d_1, \ldots, d_n] \leftarrow$ `gather`$(i_l, \overline{a_n})$<br>$[d_{n+1}, \ldots, d_{n+m-w}] \leftarrow$ `gather`$(i_r, \overline{b_m})$<br>$d_t \leftarrow$ `gather`$\langle \otimes \rangle([i_l, i_r], [a_t, b_t])$ }, $[\overline{d_n}, d_t]$) |

**Figure 14.** A subset of the function `compile` :: RAM $\longrightarrow$ [instr] $\times$ [reg] which translates a RAM program to APM via a per-RAM operator translation rules. We assume register names are created from fresh symbols and never conflict. We use $\cdot$ to denote sequential composition of instructions and $\leftarrow$ to denote assignment. Translation proceeds in the context of a database that is partitioned into three components: $F_T^{\text{stable}}$, $F_T^{\text{recent}}$, and $F_T^{\Delta}$. This enables semi-naive evaluation, as discussed in Section 3.4.