Scallop: A Language for Neurosymbolic Programming

ZIYANG LI*, University of Pennsylvania, USA
JIANI HUANG*, University of Pennsylvania, USA
MAYUR NAIK, University of Pennsylvania, USA

We present Scallop, a language which combines the benefits of deep learning and logical reasoning. Scallop enables users to write a wide range of neurosymbolic applications and train them in a data- and compute-efficient manner. It achieves these goals through three key features: 1) a flexible symbolic representation that is based on the relational data model; 2) a declarative logic programming language that is based on Datalog and supports recursion, aggregation, and negation; and 3) a framework for automatic and efficient differentiable reasoning that is based on the theory of provenance semirings. We evaluate Scallop on a suite of eight neurosymbolic applications from the literature. Our evaluation demonstrates that Scallop is capable of expressing algorithmic reasoning in diverse and challenging AI tasks, provides a succinct interface for machine learning programmers to integrate logical domain knowledge, and yields solutions that are comparable or superior to state-of-the-art models in terms of accuracy. Furthermore, Scallop's solutions outperform these models in aspects such as runtime and data efficiency, interpretability, and generalizability.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Learning paradigms**; **Probabilistic reasoning**; **Logical and relational learning**.

Additional Key Words and Phrases: Neurosymbolic methods, Differentiable reasoning

ACM Reference Format:
Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proc. ACM Program. Lang.* 7, PLDI, Article 166 (June 2023), 25 pages. https://doi.org/10.1145/3591280

1 INTRODUCTION

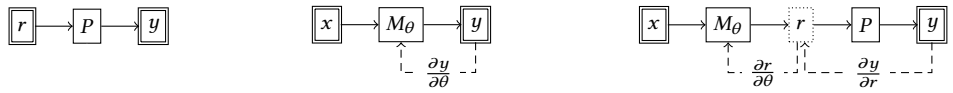
Classical algorithms and deep learning embody two prevalent paradigms of modern programming. Classical algorithms are well suited for exactly-defined tasks, such as sorting a list of numbers or finding a shortest path in a graph. Deep learning, on the other hand, is well suited for tasks that are not tractable or feasible to perform using classical algorithms, such as detecting objects in an image or parsing natural language text. These tasks are typically specified using a set of input-output training data, and solving them involves learning the parameters of a deep neural network to fit the data using gradient-based methods.

The two paradigms are complementary in nature. For instance, a classical algorithm such as the logic program P depicted in Figure 1a is interpretable but operates on limited (e.g., structured) input r. In contrast, a deep neural network such as M_θ depicted in Figure 1b can operate on rich (e.g., unstructured) input x but is not interpretable. Modern applications demand the capabilities of both paradigms. Examples include question answering [Rajpurkar et al. 2016], code completion [Chen et al. 2021b], and mathematical problem solving [Lewkowycz et al. 2022], among many others.

*These two authors contributed equally.

Authors' addresses: Ziyang Li, University of Pennsylvania, USA, liby99@seas.upenn.edu; Jiani Huang, University of Pennsylvania, USA, jianih@seas.upenn.edu; Mayur Naik, University of Pennsylvania, USA, mnaik@seas.upenn.edu.

2475-1421/2023/6-ART166
https://doi.org/10.1145/3591280



(a) Logic program. (b) Neural model. (c) A basic neurosymbolic program.

Fig. 1. Comparison of different paradigms. Logic program P accepts only structured input r whereas neural model M_θ with parameter θ can operate on unstructured input x. Supervision is provided on data indicated in double boxes. Under *algorithmic supervision*, a neurosymbolic program must learn θ without supervision on r.

For instance, code completion requires deep learning to comprehend programmer intent from the code context, and classical algorithms to ensure that the generated code is correct. A natural and fundamental question then is how to program such applications by integrating the two paradigms.

Neurosymbolic programming is an emerging paradigm that aims to fulfill this goal [Chaudhuri et al. 2021]. It seeks to integrate symbolic knowledge and reasoning with neural architectures for better efficiency, interpretability, and generalizability than the neural or symbolic counterparts alone. Consider the task of handwritten formula evaluation [Li et al. 2020], which takes as input a formula as an image, and outputs a number corresponding to the result of evaluating it. An input-output example for this task is $\langle x = \text{<image>}, y = 1.6 \rangle$. A neurosymbolic program for this task, such as the one depicted in Figure 1c, might first apply a convolutional neural network M_θ to the input image to obtain a structured intermediate form r as a sequence of symbols ['1', '+', '3', '/', '5'], followed by a classical algorithm P to parse the sequence, evaluate the parsed formula, and output the final result 1.6.

Despite significant strides in individual neurosymbolic applications [Chen et al. 2020; Li et al. 2020; Mao et al. 2019; Minervini et al. 2020; Wang et al. 2019; Yi et al. 2018], there is a lack of a language with compiler support to make the benefits of the neurosymbolic paradigm more widely accessible. We set out to develop such a language and identified five key criteria that it should satisfy in order to be practical. These criteria, annotated by the components of the neurosymbolic program in Figure 1c, are as follows:

(1) A symbolic data representation for r that supports diverse kinds of data, such as image, video, natural language text, tabular data, and their combinations.
(2) A symbolic reasoning language for P that allows to express common reasoning patterns such as recursion, negation, and aggregation.
(3) An automatic and efficient differentiable reasoning engine for learning ($\frac{\partial y}{\partial r}$) under *algorithmic supervision*, i.e., supervision on observable input-output data (x, y) but not r.
(4) The ability to tailor learning ($\frac{\partial y}{\partial r}$) to individual applications' characteristics, since non-continuous loss landscapes of logic programs hinder learning using a one-size-fits-all method.
(5) A mechanism to leverage and integrate with existing training pipelines ($\frac{\partial r}{\partial \theta}$), implementations of neural architectures and models M_θ, and hardware (e.g., GPU) optimizations.

In this paper, we present Scallop, a language which satisfies the above criteria. The key insight underlying Scallop is its choice of three inter-dependent design decisions: a relational model for symbolic data representation, a declarative language for symbolic reasoning, and a provenance framework for differentiable reasoning. We elaborate upon each of these decisions.

Relations can represent arbitrary graphs and are therefore well suited for representing symbolic data in Scallop applications. For instance, they can represent *scene graphs*, a canonical symbolic representation of images [Johnson et al. 2015], or abstract syntax trees, a symbolic representation of natural language text. Further, they can be combined in a *relational database* to represent multi-modal data. Relations are also a natural fit for probabilistic reasoning [Dries et al. 2015], which is necessary since symbols in r produced by neural model M_θ can have associated probabilities.

Next, the symbolic reasoning program P in Scallop is specified in a declarative logic programming language. The language extends Datalog [Abiteboul et al. 1995] and is expressive enough for programmers to specify complex domain knowledge patterns that the neural model M_θ would struggle with. Datalog implementations can take advantage of optimizations from the literature on relational database systems. This in turn enables efficient inference and learning since the logical domain knowledge specifications of the task at hand help reduce the burden of M_θ, whose responsibilities are now less complex and more modular. Finally, Datalog is rule-based, which makes programs easier to write, debug, and verify. It also facilitates inferring them by leveraging techniques from program synthesis [Gulwani et al. 2017] and ILP [Cropper and Dumančić 2022].

While symbolic reasoning offers many aforementioned benefits, it poses a fundamental challenge for learning parameter θ. Deep learning relies on gradient-based methods, enabled by the differentiable nature of the low-level activation functions that comprise M_θ, to obtain $\frac{\partial r}{\partial \theta}$. The key challenge then concerns how to support automatic and efficient differentiation of the high-level logic program P to obtain $\frac{\partial y}{\partial r}$, which can be used in conjunction with $\frac{\partial r}{\partial \theta}$ to compute $\frac{\partial y}{\partial \theta}$. Scallop addresses this problem by leveraging the framework of *provenance semirings* [Green et al. 2007]. The framework proposes a common algebraic structure for applications that define annotations (i.e., tags) for tuples and propagate the annotations from inputs to outputs of relational algebra (RA) queries. One of our primary contributions is a novel adaptation of the framework for differentiable reasoning for an extended fragment of RA that includes recursion, negation, and aggregation. Scallop implements an extensible library of provenance structures including the extended max-min semiring and the top-k proofs semiring [Huang et al. 2021]. We further demonstrate that different provenance structures enable different heuristics for the gradient calculations, providing an effective mechanism to tailor the learning process to individual applications' characteristics.

We have implemented a comprehensive and open-source toolchain for Scallop in 45K LoC of Rust. It includes a compiler, an interpreter, and PyTorch bindings to integrate Scallop programs with existing machine learning pipelines. We evaluate Scallop using a suite of eight neurosymbolic applications that span the domains of image and video processing, natural language processing, planning, and knowledge graph querying, in a variety of learning settings such as supervised learning, reinforcement learning, rule learning, and contrastive learning. Our evaluation demonstrates that Scallop is expressive and yields solutions of comparable, and often times superior, accuracy than state-of-the-art models. We show additional benefits of Scallop's solutions in terms of runtime and data efficiency, interpretability, and generalizability.

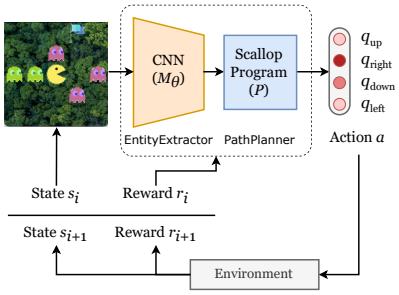
Any programming language treatise would be remiss without acknowledging the language's lineage. TensorLog [Cohen et al. 2017] and DeepProbLog (DPL) [Manhaeve et al. 2021] pioneered the idea of extending probabilistic logic programming languages (e.g., ProbLog [Dries et al. 2015]) with differentiable reasoning. Scallop was originally proposed in [Huang et al. 2021] to improve the scalability of DPL by using Datalog instead of Prolog and relaxing its exact probabilistic semantics. We build upon [Huang et al. 2021] by extending its expressiveness, formalizing the semantics, developing a customizable provenance framework, and providing a practical toolchain.

The rest of the paper is organized as follows. Section 2 presents an illustrative overview of Scallop. Section 3 describes Scallop's language for symbolic reasoning. Section 4 presents the differentiable reasoning framework. Section 5 describes our implementation of Scallop. Section 6 empirically evaluates Scallop on a benchmark suite. Section 7 surveys related work and Section 8 concludes.

2 ILLUSTRATIVE OVERVIEW

We illustrate Scallop using an reinforcement learning (RL) based planning application which we call PacMan-Maze. The application, depicted in Fig. 2a, concerns an intelligent agent realizing a sequence of actions in a simplified version of the PacMan maze game. The maze is an implicit 5×5

(a) Three states of one gameplay session.



(b) Architecture of application with Scallop.

```
class PacManAgent(torch.nn.Module):
 def __init__(self, grid_dim, cell_size):
  # initializations...
  self.extract_entities =
   EntityExtractor(grid_dim, cell_size)
  self.path_planner = ScallopModule(
   file="path_planner.scl",
   provenance="diff-top-k-proofs", k=1,
   input_mappings={"actor": cells,
    "goal": cells, "enemy": cells},
   output_mappings={"next_action": actions})

 def forward(self, game_state_image):
  actor, goal, enemy =
   self.extract_entities(game_state_image)
  next_action = self.path_planner(
   actor=actor, goal=goal, enemy=enemy)
  return next_action
```

(c) Snippet of implementation in Python.

Fig. 2. Illustration of a planning application PacMan-Maze in Scallop.

grid of cells. Each cell is either empty or has an entity, which can be either the *actor* (PacMan), the *goal* (flag), or an *enemy* (ghost). At each step, the agent moves the actor in one of four directions: up, down, right, or left. The game ends when the actor reaches the goal or hits an enemy. The maze is provided to the agent as a raw image that is updated at each step, requiring the agent to process sensory inputs, extract relevant features, and logically plan the path to take. Additionally, each session of the game has randomized initial positions of the actor, the goal, and the enemies.

Concretely, the game is modeled as a sequence of interactions between the agent and an environment, as depicted in Fig. 2b. The game state $s_i \in S$ at step i is a 200×200 colored image ($S = \mathbb{R}^{200\times200\times3}$). The agent proposes an action $a_i \in A = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$ to the environment, which generates a new image s_{i+1} as the next state. The environment also returns a reward r_i to the agent: 1 upon reaching the goal, and 0 otherwise. This procedure repeats until the game ends or reaches a predefined limit on the number of steps.

A popular RL method to realize our application is Q-Learning. Its goal is to learn a function $Q : S \times A \to \mathbb{R}$ that returns the expected reward of taking action a_i in state s_i.[1] Since the game states are images, we employ Deep Q-Learning [Mnih et al. 2015], which approximates the Q function using a convolutional neural network (CNN) model with learned parameter θ. An end-to-end deep learning based approach for our application involves training the model to predict the Q-value of each action for a given game state. This approach takes 50K training episodes to achieve a 84.9% test success rate, where a single episode is one gameplay session from start to end.

In contrast, a neurosymbolic solution using Scallop only needs 50 training episodes to attain a 99.4% test success rate. Scallop enables to realize these benefits of the neurosymbolic paradigm by decomposing the agent's task into separate neural and symbolic components, as shown in Fig. 2b. These components perform sub-tasks that are ideally suited for their respective paradigms: the neural component perceives pixels of individual cells of the image at each step to identify the entities in them, while the symbolic component reasons about enemy-free paths from the actor

[1]We elide the Q-Learning algorithm as it is not needed to illustrate the neurosymbolic programming aspects of our example.

```
1  // File path_planner.scl
2  type actor(x: i32, y: i32), goal(x: i32, y: i32), enemy(x: i32, y: i32)
3
4  const UP = 0, DOWN = 1, RIGHT = 2, LEFT = 3
5  rel safe_cell(x, y) = range(0, 5, x), range(0, 5, y), not enemy(x, y)
6  rel edge(x, y, x, yp, UP) = safe_cell(x, y), safe_cell(x, yp), yp == y + 1
7  // Rules for DOWN, RIGHT, and LEFT edges are omitted...
8
9  rel next_pos(p, q, a) = actor(x, y), edge(x, y, p, q, a)
10 rel path(x, y, x, y) = next_pos(x, y, _)
11 rel path(x1, y1, x3, y3) = path(x1, y1, x2, y2), edge(x2, y2, x3, y3, _)
12 rel next_action(a) = next_pos(p, q, a), path(p, q, r, s), goal(r, s)
```

Fig. 3. The logic program of the PacMan-Maze application in Scallop.

to the goal to determine the optimal next action. Fig. 2c shows an outline of this architecture's implementation using the popular PyTorch framework.

Concretely, the neural component is still a CNN, but it now takes the pixels of a single cell in the input image at a time, and classifies the entity in it. The implementation of the neural component (`EntityExtractor`) is standard and elided for brevity. It is invoked on lines 14-15 with input `game_state_image`, a tensor in $\mathbb{R}^{200\times200\times3}$, and returns three $\mathbb{R}^{5\times5}$ tensors of entities. For example, `actor` is an $\mathbb{R}^{5\times5}$ tensor and $\texttt{actor}_{ij}$ is the probability of the actor being in cell (i, j). A representation of the entities is then passed to the symbolic component on lines 16-17, which derives the Q-value of each action. The symbolic component, which is configured on lines 6-11, comprises the Scallop program shown in Fig. 3. We next illustrate the three key design decisions of Scallop outlined in Section 1 with respect to this program.

Relational Model. In Scallop, the primary data structure for representing symbols is a *relation*. In our example, the game state can be symbolically described by the kinds of entities that occur in the discrete cells of a 5×5 grid. We can therefore represent the input to the symbolic component using binary relations for the three kinds of entities: `actor`, `goal`, and `enemy`. For instance, the fact `actor(2,3)` indicates that the actor is in cell (2,3). Likewise, since there are four possible actions, the output of the symbolic component is represented by a unary relation `next_action`.

Symbols extracted from unstructured inputs by neural networks have associated probabilities, such as the $\mathbb{R}^{5\times5}$ tensor `actor` produced by the neural component in our example (line 14 of Fig. 2c). Scallop therefore allows to associate tuples with probabilities, e.g. `0.96 :: actor(2,3)`, to indicate that the actor is in cell (2,3) with probability 0.96. More generally, Scallop enables the conversion of tensors in the neural component to and from relations in the symbolic component via input-output mappings (lines 9-11 in Fig. 2c), allowing the two components to exchange information seamlessly.

Declarative Language. Another key consideration in a neurosymbolic language concerns what constructs to provide for symbolic reasoning. Scallop uses a declarative language based on Datalog, which we present in Section 3 and illustrate here using the program in Fig. 3. The program realizes the symbolic component of our example using a set of logic rules. Following Datalog syntax, they are "if-then" rules, read right to left, with commas denoting conjunction.

Recall that we wish to determine an action a (up, down, right, or left) to a cell (p, q) that is adjacent to the actor's cell (x, y) such that there is an enemy-free path from (p, q) to the goal's cell (r, s). The nine depicted rules succinctly compute this sophisticated reasoning pattern by building successively complex relations, with the final rule (on line 14) computing all such actions.[2]

[2]We elide showing an auxiliary relation of all grid cells tagged with probability 0.99 which serves as the penalty for taking a step. Thus, longer paths are penalized more, driving the symbolic program to prioritize moving closer to the goal.

The arguably most complex concept is the `path` relation which is recursively defined (on lines 10-11). Recursion allows to define the pattern succinctly, enables the trained application to generalize to grids arbitrarily larger than 5×5 unlike the purely neural version, and makes the pattern more amenable to synthesis from input-output examples. Besides recursion, Scallop also supports negation and aggregation; together, these features render the language adequate for specifying common high-level reasoning patterns in practice.

Differentiable Reasoning. With the neural and symbolic components defined, the last major consideration concerns how to train the neural component using only end-to-end supervision. In our example, supervision is provided in the form of a reward of 1 or 0 per gameplay session, depending upon whether or not the sequence of actions by the agent successfully led the actor to the goal without hitting any enemy. This form of supervision, called algorithmic or weak supervision, alleviates the need to label intermediate relations at the interface of the neural and symbolic components, such as the `actor`, `goal`, and `enemy` relations. However, this also makes it challenging to learn the gradients for the tensors of these relations, which in turn are needed to train the neural component using gradient-descent techniques.

The key insight in Scallop is to exploit the structure of the logic program to guide the gradient calculations. The best heuristic for such calculations depends on several application characteristics such as the amount of available data, reasoning patterns, and the learning setup. Scallop provides a convenient interface for the user to select from a library of built-in heuristics. Furthermore, since all of these heuristics follow the structure of the logic program, Scallop implements them uniformly as instances of a general and extensible *provenance framework*, described in Section 4. For our example, line 8 in Fig. 2c specifies `diff-top-k-proofs` with `k=1` as the heuristic to use, which is the default in Scallop that works best for many applications, as we demonstrate in Section 6.

3 LANGUAGE

We provide an overview of Scallop's language for symbolic reasoning which we previously illustrated in the program shown in Fig. 3. Here, we illustrate each of the key constructs using examples of inferring kinship relations.

3.1 Data Types

The fundamental data type in Scallop is set-valued relations comprising tuples of statically-typed primitive values. The primitive data types include signed and unsigned integers of various sizes (e.g. `i32`, `usize`), single- and double-precision floating point numbers (`f32`, `f64`), boolean (`bool`), character (`char`), and string (`String`). The following example declares two binary relations, `mother` and `father`:

```
type mother(c: String, m: String), father(c: String, f: String)
```

Values of relations can be specified via individual tuples or a set of tuples of constant literals:

```
rel mother("Bob", "Christine")                    // Christine is Bob's mother
rel father = {("Alice", "Bob"), ("John", "Bob")} // Bob is father of two kids
```

As a shorthand, primitive values can be named and used as constant variables:

```
const FATHER = 0, MOTHER = 1, GRANDMOTHER = 2, ... // other relationships
rel composition(FATHER, MOTHER, GRANDMOTHER) // father's mother is grandmother
```

Type declarations are optional since Scallop supports type inference. The type of the `composition` relation is inferred as `(usize, usize, usize)` since the default type of unsigned integers is `usize`.

3.2 (Horn) Rules

Since Scallop's language is based on Datalog, it supports "if-then" rule-like Horn clauses. Each rule is composed of a head atom and a body, connected by the symbol `:-` or `=`. The following code shows two rules defining the `grandmother` relation. Conjunction is specified using "`,`"-separated atoms within the rule body whereas disjunction is specified by multiple rules with the same head predicate. Each variable appearing in the head must also appear in some positive atom in the body (we introduce negative atoms below).

```
rel grandmother(a, c) :- father(a, b), mother(b, c) // father's mother
rel grandmother(a, c) :- mother(a, b), mother(b, c) // mother's mother
```

Conjunctions and disjunctions can also be expressed using logical connectives like `and`, `or`, and `implies`. For instance, the following rule is equivalent to the above two rules combined.

```
rel grandmother(a, c) = (mother(a, b) or father(a, b)) and mother(b, c)
```

Scallop supports value creation by means of foreign functions (FFs). FFs are polymorphic and include arithmetic operators such as `+` and `-`, comparison operators such as `!=` and `>=`, type conversions such as `[i32] as String`, and built-in functions like `$hash` and `$string_concat`. They only operate on primitive values but not relational tuples or atoms. The following example shows how strings are concatenated together using FF, producing the result `full_name("Alice Lee")`.

```
rel first_name("Alice"), last_name("Lee")
rel full_name($string_concat(x, "␣", y)) = first_name(x), last_name(y)
```

Note that FFs can fail due to runtime errors such as division-by-zero and integer overflow, in which case the computation for that single fact is omitted. In the example below, when dividing `6` by `denominator`, the result is not computed for denominator `0` since it causes a FF failure:

```
rel denominator = {0, 1, 2}        // there are 3 denominators
rel result(6 / x) = denominator(x) // results contain only integers 3 and 6
```

The purpose of this semantics is to support probabilistic extensions (Section 3.3) rather than silent suppression of runtime errors. When dealing with floating-point numbers, tuples with `NaN` (not-a-number) are also discarded.

Recursion. A relation r is dependent on s if an atom s appears in the body of a rule with head atom r. A *recursive* relation is one that depends on itself, directly or transitively. The following rule derives additional `kinship` facts by composing existing `kinship` facts using the `composition` relation.

```
rel kinship(r3,a,c) = kinship(r1,a,b), kinship(r2,b,c), composition(r1,r2,r3)
```

Negation. Scallop supports stratified negation using the `not` operator on atoms in the rule body. The following example shows a rule defining the `has_no_children` relation as any person `p` who is neither a father nor a mother. Note that we need to bound `p` by a positive atom `person` in order for the rule to be well-formed.

```
rel person = {"Alice", "Bob", "Christine"} // can omit () since arity is 1
rel has_no_children(p) = person(p) and not father(_, p) and not mother(_, p)
```

A relation r is *negatively* dependent on s if a negated atom s appears in the body of a rule with head atom r. In the above example, `has_no_children` negatively depends on `father`. A relation cannot be negatively dependent on itself, directly or transitively, as Scallop supports only stratified negation. The following rule is rejected by the compiler, as the negation is not stratified:

```
rel something_is_true() = not something_is_true() // compilation error!
```

Aggregation. Scallop also supports stratified aggregation. The set of built-in aggregators include common ones such as `count`, `sum`, `max`, and first-order quantifiers `forall` and `exists`. Besides the operator, the aggregation specifies the binding variables, the aggregation body to bound those variables, and the result variable(s) to assign the result. The aggregation in the example below reads, "variable `n` is assigned the count of `p`, such that `p` is a person":

```
rel num_people(n) = n := count(p: person(p))
```

In the rule, `p` is the binding variable and `n` is the result variable. Depending on the aggregator, there could be multiple binding variables or multiple result variables. Further, Scallop supports SQL-style group-by using a `where` clause in the aggregation. In the following example, we compute the number of children of each person `p`, which serves as the group-by variable.

```
rel parent(a, b) = father(a, b) or mother(a, b)
rel num_child(p, n) = n := count(c: parent(c, p) where p: person(p))
```

Finally, quantifier aggregators such as `forall` and `exists` return one boolean variable. For instance, in the aggregation below, variable `sat` is assigned the truthfulness (`true` or `false`) of the following statement: "for all `a` and `b`, if `b` is `a`'s father, then `a` is `b`'s son or daughter".

```
rel integrity_constraint(sat) =
    sat := forall(a, b: father(a, b) implies (son(b, a) or daughter(b, a)))
```

There are a couple of syntactic checks on aggregations. First, similar to negation, aggregation also needs to be stratified—a relation cannot be dependent on itself through an aggregation. Second, the binding variables must be bounded by a positive atom in the body of the aggregation.

3.3 Probabilistic Extensions

Although Scallop is designed primarily for neurosymbolic programming, its syntax also supports probabilistic programming. This is especially useful when debugging Scallop code before integrating it with a neural network. Consider a machine learning programmer who wishes to extract structured relations from a natural language sentence "Bob takes his daughter Alice to the beach". The programmer could imitate a neural network producing a probability distribution of kinship relations between Alice (`A`) and Bob (`B`) as follows:

```
rel kinship = {0.95::(FATHER, A, B); 0.01::(MOTHER, A, B); ... }
```

Here, we list out all possible kinship relations between Alice and Bob. For each of them, we use the syntax `[PROB]::[TUPLE]` to tag the kinship tuples with probabilities. The semicolon ";" separating them specifies that they are mutually exclusive—Bob cannot be both the mother and father of Alice.

Scallop also supports operators to sample from probability distributions. They share the same surface syntax as aggregations, allowing sampling with group-by. The following rule deterministically picks the most likely kinship relation between a given pair of people `a` and `b`, which are implicit group-by variables in this aggregation. As the end, only one fact, `0.95::top_1_kinship(FATHER, A, B)`, will be derived according to the above probabilities.

```
rel top_1_kinship(r, a, b) = r := top<1>(rp: kinship(rp, a, b))
```

Other types of sampling are also supported, including categorical sampling (`categorical<K>`) and uniform sampling (`uniform<K>`), where a static constant `K` denotes the number of trials.

Finally, rules can also be tagged by probabilities which can reflect their confidence. The following rule states that a grandmother's daughter is one's mother with 90% confidence.

```
rel 0.9::mother(a, c) = grandmother(a, b) and daughter(b, c)
```

Probabilistic rules are syntactic sugar. They are implemented by introducing in the rule's body an auxiliary 0-ary (i.e., boolean) fact that is regarded true with the tagged probability.

(Tag)	t	$\in$	T
(False)	$\mathbf{0}$	$\in$	T
(True)	$\mathbf{1}$	$\in$	T
(Disjunction)	$\oplus$	:	$T \times T \to T$
(Conjunction)	$\otimes$	:	$T \times T \to T$
(Negation)	$\ominus$	:	$T \to T$
(Saturation)	$\Subset$	:	$T \times T \to \text{Bool}$

Fig. 4. Algebraic interface for provenance.

(Predicate)	p		
(Aggregator)	g	::=	`count` | `sum` | `max` | `exists` | ...
(Expression)	e	::=	$p \mid \gamma_g(e) \mid \pi_\alpha(e) \mid \sigma_\beta(e)$
		|	$e_1 \cup e_2 \mid e_1 \times e_2 \mid e_1 - e_2$
(Rule)	r	::=	$p \leftarrow e$
(Stratum)	s	::=	$\{r_1, \ldots, r_n\}$
(Program)	$\overline{s}$	::=	$s_1; \ldots; s_n$

Fig. 5. Abstract syntax of core fragment of SclRam.

4 REASONING FRAMEWORK

The preceding section presented Scallop's surface language for use by programmers to express discrete reasoning. However, the language must also support differentiable reasoning to enable end-to-end training. In this section, we formally define the semantics of the language by means of a provenance framework. We show how Scallop uniformly supports different reasoning modes—discrete, probabilistic, and differentiable—simply by defining different provenances.

We start by presenting the basics of our provenance framework (Section 4.1). We then present a low-level representation SclRam, its operational semantics, and its interface to the rest of a Scallop application (Sections 4.2-4.4). We next present differentiation and three different provenances for differentiable reasoning (Section 4.5). Lastly, we discuss practical considerations (Section 4.6).

4.1 Provenance Framework

Scallop's provenance framework enables to tag and propagate additional information alongside relational tuples in the logic program's execution. The framework is based on the theory of *provenance semirings* [Green et al. 2007]. Fig. 4 defines Scallop's algebraic interface for provenance. We call the additional information a *tag* t from a *tag space* T. There are two distinguished tags, $\mathbf{0}$ and $\mathbf{1}$, representing unconditionally *false* and *true* tags. Tags are propagated through operations of binary *disjunction* $\oplus$, binary *conjunction* $\otimes$, and unary *negation* $\ominus$ resembling logical *or*, *and*, and *not*. Lastly, a *saturation* check $\Subset$ serves as a customizable stopping mechanism for fixed-point iteration.

All of the above components combined form a 7-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes, \ominus, \Subset)$ which we call a *provenance* T. Scallop provides a built-in library of provenances and users can add custom provenances simply by implementing this interface.

Example 4.1. `max-min-prob` (`mmp`) $\triangleq$ $([0, 1], 0, 1, \max, \min, \lambda x.(1 - x), ==)$, is a built-in *probabilistic provenance*, where tags are numbers between 0 and 1 that are propagated with max and min. The tags do not represent true probabilities but are merely an approximation. We discuss richer provenances for more accurate probability calculations later in this section.

A provenance must satisfy a few properties. First, the 5-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ should form a semiring. That is, $\mathbf{0}$ is the additive identity and annihilates under multiplication, $\mathbf{1}$ is the multiplicative identity, $\oplus$ and $\otimes$ are associative and commutative, and $\otimes$ distributes over $\oplus$. To guarantee the existence of fixed points (which are discussed in Section 4.3), it must also be *absorptive*, i.e., $t_1 \oplus (t_1 \otimes t_2) = t_1$ [Dannert et al. 2021]. Moreover, we need $\ominus \mathbf{0} = \mathbf{1}$, $\ominus \mathbf{1} = \mathbf{0}$, $\mathbf{0} \Subset \mathbf{1}$, $\mathbf{0} \Subset \mathbf{0}$, and $\mathbf{1} \Subset \mathbf{1}$. A provenance which violates an individual property (e.g. absorptive) is still useful to applications that do not use the affected features (e.g. recursion) or if the user simply wishes to bypass the restrictions.

4.2 SclRam Intermediate Language

Scallop programs are compiled to a low-level representation called SclRam. Fig. 5 shows the abstract syntax of a core fragment of SclRam. Expressions resemble queries in an extended relational algebra. They operate over relational predicates (p) using unary operations for aggregation (γ_g

$$
\begin{array}{rlllll}
\text{(Constant)} & \mathbb{C} & \ni & c & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{str} \mid \ldots \\
\text{(Tuple)} & \mathbb{U} & \ni & u & ::= & c \mid (u_1, \ldots, u_n) \\
\text{(Tagged-Tuple)} & \mathbb{U}_T & \ni & u_t & ::= & t :: u \\
\text{(Fact)} & \mathbb{F} & \ni & f & ::= & p(u) \\
\text{(Tagged-Fact)} & \mathbb{F}_T & \ni & f_t & ::= & t :: p(u)
\end{array}
$$

$$
\begin{array}{rlllll}
\text{(Tuples)} & U & \in & \mathcal{U} & \triangleq & \mathcal{P}(\mathbb{U}) \\
\text{(Tagged-Tuples)} & U_T & \in & \mathcal{U}_T & \triangleq & \mathcal{P}(\mathbb{U}_T) \\
\text{(Facts)} & F & \in & \mathcal{F} & \triangleq & \mathcal{P}(\mathbb{F}) \\
\text{(Database)} & F_T & \in & \mathcal{F}_T & \triangleq & \mathcal{P}(\mathbb{F}_T)
\end{array}
$$

Fig. 6. Semantic domains for ScLRam.

with aggregator g), projection (π_α with mapping α), and selection (σ_β with condition β), and binary operations union ($\cup$), product ($\times$), and difference ($-$).

A rule r in ScLRam is denoted $p \leftarrow e$, where predicate p is the rule head and expression e is the rule body. An unordered set of rules combined form a stratum s, and a sequence of strata $s_1; \ldots; s_n$ constitutes a ScLRam program. Rules in the same stratum have distinct head predicates. Denoting the set of head predicates in stratum s by P_s, we also require $P_{s_i} \cap P_{s_j} = \emptyset$ forall $i \neq j$ in a program. Stratified negation and aggregation from the surface language is enforced as syntax restrictions in ScLRam: within a rule in stratum s_i, if a relational predicate p is used under aggregation (γ) or right-hand-side of difference ($-$), that predicate p cannot appear in P_{s_j} if $j \geq i$.

We next define the semantic domains in Fig. 6 which are used subsequently to define the semantics of ScLRam. A tuple u is either a constant or a sequence of tuples. A fact $p(u) \in \mathbb{F}$ is a tuple u named under a relational predicate p. Tuples and facts can be tagged, forming *tagged tuples* ($t :: u$) and *tagged facts* ($t :: p(u)$). Given a set of tagged tuples U_T, we say $U_T \vDash u$ iff there exists a t such that $t :: u \in U_T$. A set of tagged facts form a database F_T. We use bracket notation $F_T[p]$ to denote the set of tagged facts in F_T under predicate p.

4.3 Operational Semantics of ScLRam

We now present the operational semantics for our core fragment of ScLRam in Fig. 7. A ScLRam program $\bar{s}$ takes as input an *extensional database* (EDB) F_T, and returns an *intentional database* (IDB) $F'_T = [\![\bar{s}]\!](F_T)$. The semantics is conditioned on the underlying provenance T. We call this *tagged semantics*, as opposed to the *untagged semantics* found in traditional Datalog.

Basic Relational Algebra. Evaluating an expression in ScLRam yields a set of tagged tuples according to the rules defined at the top of Fig. 7. A predicate p evaluates to all facts under that predicate in the database. Selection filters tuples that satisfy condition β, and projection transforms tuples according to mapping α. The mapping function α is partial: it may fail since it can apply foreign functions to values. A tuple in a union $e_1 \cup e_2$ can come from either e_1 or e_2. In (cartesian) product, each pair of incoming tuples combine and we use $\otimes$ to compute the tag conjunction.

Difference and Negation. To evaluate a difference expression $e_1 - e_2$, there are two cases depending on whether a tuple u evaluated from e_1 appears in the result of e_2. If it does not, we simply propagate the tuple and its tag to the result (Diff-1); otherwise, we get $t_1 :: u$ from e_1 and $t_2 :: u$ from e_2. Instead of erasing the tuple u from the result as in untagged semantics, we propagate a tag $t_1 \otimes (\ominus\, t_2)$ with u (Diff-2). In this manner, information is not lost during negation. Fig. 8 compares the evaluations of an example difference expression under different semantics. While the tuple $(2, 3)$ is removed in the outcome with untagged semantics, it remains with the tagged semantics.

Aggregation. Aggregators in ScLRam are discrete functions g operating on set of (untagged) tuples $U \in \mathcal{U}$. They return a *set* of aggregated tuples to account for aggregators like `argmax` which can produce multiple outcomes. For example, we have $\texttt{count}(U) = \{|U|\}$. However, in the probabilistic domain, discrete symbols do not suffice. Given n tagged tuples to aggregate over, each tagged tuple can be turned on or off, resulting in 2^n distinct *worlds*. Each world is a partition of the input set U_T ($|U_T| = n$). Denoting the positive part as X_T and the negative part as $\overline{X}_T = U_T - X_T$, the tag associated with this world is a conjunction of tags in X_T and negated tags in $\overline{X}_T$. Aggregating on

Expression semantics $\boxed{\alpha : \mathbb{U} \to \mathbb{U}, \quad \beta : \mathbb{U} \to \text{Bool}, \quad g : \mathcal{U} \to \mathcal{U}, \quad [\![e]\!] : \mathcal{F}_T \to \mathcal{U}_T}$

$$\frac{t :: p(u) \in F_T}{t :: u \in [\![p]\!](F_T)} \text{ (Predicate)} \quad \frac{t :: u \in [\![e]\!](F_T) \qquad \beta(u) = \text{true}}{t :: u \in [\![\sigma_\beta(e)]\!](F_T)} \text{ (Select)} \quad \frac{t :: u \in [\![e]\!](F_T) \qquad u' = \alpha(u)}{t :: u' \in [\![\pi_\alpha(e)]\!](F_T)} \text{ (Project)}$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \cup [\![e_2]\!](F_T)}{t :: u \in [\![e_1 \cup e_2]\!](F_T)} \text{ (Union)} \quad \frac{t_1 :: u_1 \in [\![e_1]\!](F_T) \qquad t_2 :: u_2 \in [\![e_2]\!](F_T)}{(t_1 \otimes t_2) :: (u_1, u_2) \in [\![e_1 \times e_2]\!](F_T)} \text{ (Product)}$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \qquad [\![e_2]\!](F_T) \not\models u}{t :: u \in [\![e_1 - e_2]\!](F_T)} \text{ (Diff-1)} \quad \frac{t_1 :: u \in [\![e_1]\!](F_T) \qquad t_2 :: u \in [\![e_2]\!](F_T)}{(t_1 \otimes (\ominus\, t_2)) :: u \in [\![e_1 - e_2]\!](F_T)} \text{ (Diff-2)}$$

$$\frac{X_T \subseteq [\![e]\!](F_T) \qquad \{t_i :: u_i\}_{i=1}^n = X_T \qquad \{\bar{t}_j :: \bar{u}_j\}_{j=1}^m = [\![e]\!](F_T) - X_T \qquad u \in g(\{u_i\}_{i=1}^n)}{(\bigotimes_{i=1}^n t_i) \otimes (\bigotimes_{j=1}^m (\ominus\, \bar{t}_j)) :: u \in [\![\gamma_g(e)]\!](F_T)} \text{ (Aggregate)}$$

Rule semantics $\boxed{\langle . \rangle : \mathcal{U}_T \to \mathcal{U}_T, \quad [\![r]\!] : \mathcal{F}_T \to \mathcal{F}_T}$

(Normalize) $\langle U_T \rangle = \{(\bigoplus_{i=1}^n t_i) :: u \mid t_1 :: u, \ldots, t_n :: u \text{ are all tagged-tuples in } U_T \text{ with the same tuple } u\}$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad \langle [\![e]\!](F_T) \rangle \not\models u}{t^{\text{old}} :: p(u) \in [\![p \leftarrow e]\!](F_T)} \text{ (Rule-1)} \quad \frac{t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle \qquad [\![p]\!](F_T) \not\models u}{t^{\text{new}} :: p(u) \in [\![p \leftarrow e]\!](F_T)} \text{ (Rule-2)}$$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle}{(t^{\text{old}} \oplus t^{\text{new}}) :: p(u) \in [\![p \leftarrow e]\!](F_T)} \text{ (Rule-3)}$$

Stratum and Program semantics $\boxed{\text{lfp}^\circ : (\mathcal{F}_T \to \mathcal{F}_T) \to (\mathcal{F}_T \to \mathcal{F}_T), \quad [\![s]\!], [\![\bar{s}]\!] : \mathcal{F}_T \to \mathcal{F}_T}$

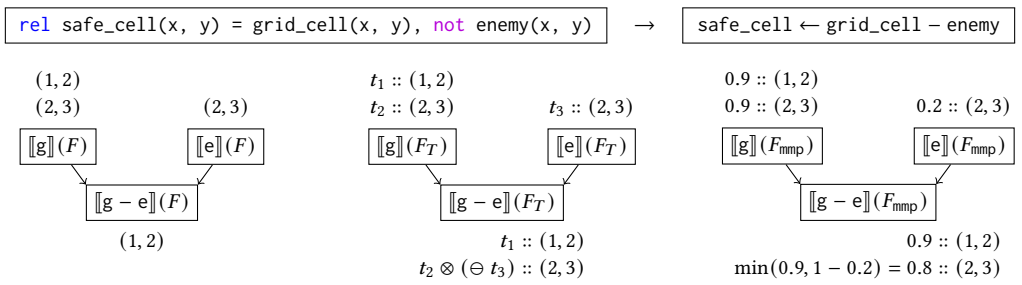
(Saturation) $F_T^{\text{old}} \doteq F_T^{\text{new}}$ iff $\forall t^{\text{new}} :: p(u) \in F_T^{\text{new}}, \exists t^{\text{old}} :: p(u) \in F_T^{\text{old}}$ such that $t^{\text{old}} \ominus t^{\text{new}}$

(Fixpoint) $\text{lfp}^\circ(h) = h \circ \cdots \circ h = h^n$ if there exists a minimum $n > 0$, such that $h^n(F_T) \doteq h^{n+1}(F_T)$

(Stratum) $[\![s]\!] = \text{lfp}^\circ(\lambda F_T . (F_T - \bigcup_{p \in P_s} F_T[p]) \cup (\bigcup_{r \in s} [\![r]\!](F_T)))$

(Program) $[\![\bar{s}]\!] = [\![s_n]\!] \circ \cdots \circ [\![s_1]\!]$, where $\bar{s} = s_1; \ldots; s_n$.

Fig. 7. Operational semantics of core fragment of ScLRam.

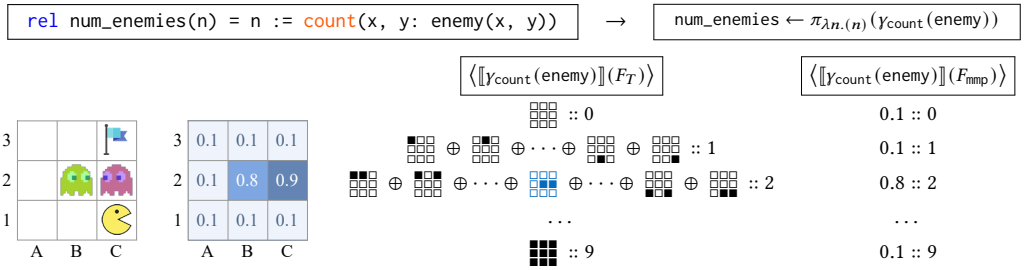


(a) Untagged semantics (b) ScLRam tagged semantics (c) ScLRam with `max-min-prob`

Fig. 8. An example rule adapted from Section 2 is compiled to a ScLRam rule with difference. g and e are abbreviated relation names. The graphs illustrate the evaluation of expression g – e under different semantics.

this world then involves applying aggregator g on tuples in the positive part X_T. This is inherently exponential if we enumerate all worlds. However, we can optimize over each aggregator and each provenance to achieve better performance. For instance, counting over `max-min-prob` tagged tuples can be implemented by an $O(n \log(n))$ algorithm, much faster than exponential. Fig. 9 demonstrates a running example and an evaluation of a counting expression under `max-min-prob` provenance. The resulting count can be 0-9, each derivable by multiple worlds.

Rules and Fixed-Point Iteration. Evaluating a rule $p \leftarrow e$ on database F_T concerns evaluating the expression e and merging the result with the existing facts under predicate p in F_T. The result of



(a) Maze image (b) enemy (c) Evaluation of the aggregation expression

Fig. 9. An example counting enemies in a PacMan maze. Shown above are the Scallop rule and compiled ScLRam rule with aggregation. (a) and (b) visualize the maze and the content of a probabilistic enemy relation. For example, we have t_{B2} :: enemy$(B, 2)$ where $t_{B2} = 0.8$. In (c), we show two normalized ($\langle . \rangle$ defined in Fig. 7) evaluation results under abstract tagged semantics and with `max-min-prob` provenance. Each symbol such as ▦ represents a world corresponding to our arena (■: enemy; □: no enemy). A world is a conjunction of 9 tags, e.g., ▦ $= t_{A3} \otimes (\ominus t_{A2}) \otimes \cdots \otimes (\ominus t_{C1})$. We mark the correct world ▦ which yields the answer 2.

evaluating e may contain duplicated tuples tagged by distinct tags, owing to expressions such as union, projection, or aggregation. Therefore, we perform *normalization* on the set to join ($\oplus$) the distinct tags. From here, there are three cases to merge the newly derived tuples ($\langle\langle [\![e]\!](F_T) \rangle\rangle$) with the previously derived tuples ($[\![p]\!](F_T)$). If a fact is present only in the old or the new, we simply propagate the fact to the output. When a tuple u appears in both the old and the new, we propagate the disjunction of the old and new tag ($t^{\text{old}} \oplus t^{\text{new}}$). Combining all cases, we obtain a set of newly tagged facts under predicate p.

Recursion in ScLRam is realized similar to least fixed point iteration in Datalog [Abiteboul et al. 1995]. The iteration happens on a per-stratum basis to enforce stratified negation and aggregation. Evaluating a single step of stratum s means evaluating all the rules in s and returning the updated database. Note that we define a specialized least fixed point operator $\mathbf{lfp}^{\circ}$, which stops the iteration once the whole database is *saturated*. Fig. 10 illustrates an evaluation involving recursion and database saturation. The whole database saturates on the 7th iteration, and finds the tag representing the optimal path for the PacMan to reach the goal. Termination is not universally guaranteed in ScLRam due to the presence features such as value creation. But its existence can be proven in a per-provenance basis. For example, it is easy to show that if a program terminates under untagged semantics, then it terminates under tagged semantics with `max-min-prob` provenance.

4.4 External Interface and Execution Pipeline

Thus far, we have only illustrated the `max-min-prob` provenance, in which the tags are approximated probabilities. There are other probabilistic provenances with more complex tags such as proof trees or boolean formulae. We therefore introduce, for each provenance T, an *input tag* space I, an *output tag* space O, a *tagging function* $\tau : I \to T$, and a *recover function* $\rho : T \to O$. For instance, all probabilistic provenances share the same input and output tag spaces $I = O = [0, 1]$ for a unified interface, while the internal tag spaces T could be different. We call the 4-tuple (I, O, τ, ρ) the *external interface* for a provenance T. The whole execution pipeline is then illustrated below:

In the context of a Scallop application, an EDB is provided in the form $F_{\texttt{option<}I\texttt{>}}$. During the *tagging phase*, τ is applied to each input tag to obtain F_T, following which the ScLRam program operates on F_T. For convenience, not all input facts need to be tagged—untagged input facts are simply

```
rel path(x1,y1,x3,y3) = (edge(x1,y1,x3,y3) or path(x1,y1,x2,y2) and edge(x2,y2,x3,y3)) and not enemy(x3,y3)
```

Iteration count i	1	2	3	4	5	6	7
$t^{(i)}_{\texttt{C1-C3}}$ in $F^{(i)}_T$	–	↑	(same)	↱ ⊕ ↱ ⊕ ⋯ ⊕ ↑	(same)	↱ ⊕ ↱ ⊕ ⋯ ⊕ ↑	(same)
$t^{(i)}_{\texttt{C1-C3}}$ in $F^{(i)}_{\texttt{mmp}}$	–	0.1	0.1	0.2	0.2	0.9	0.9
$t^{(i)}_{\texttt{C1-C3}}$ saturated?	–	false	true	false	true	false	true
$F^{(i)}_{\texttt{mmp}}$ saturated?	false	false	false	false	false	false	true

Fig. 10. A demonstration of the fixed-point iteration to check whether actor at `C1` can reach `C3` without hitting an enemy (Fig. 9a). The Scallop rule to derive this is defined on the top, and we assume bidirectional edges are populated and tagged by **1**. Let $t_{\texttt{C1-C3}}$ be the tag associated with `path(C,1,C,3)`. We use a symbol like ↑ to represent a conjunction of negated tags of `enemy` along the illustrated path, e.g. ↑ = $(\ominus t_{\texttt{C2}}) \otimes (\ominus t_{\texttt{C3}})$. 2nd iter is the first time $t_{\texttt{C1-C3}}$ is derived, but the path ↑ is blocked by an enemy. On 6th iter, the best path ↱ is derived in the tag. After that, under the `max-min-prob` provenance, both the tag $t_{\texttt{C1-C3}}$ and the database $F_{\texttt{mmp}}$ are saturated, causing the iteration to stop. Compared to untagged semantics in Datalog which will stop after 4 iterations, ScIRam with `mmp` saturates slower but allowing to explore better reasoning chains.

associated by the tag **1** in F_T. In the *recovery phase*, ρ is applied to obtain F_O, the IDB that the whole pipeline returns. Scallop allows the user to specify a set of *output relations* and ρ is only applied to tags under such relations to avoid redundant computations.

4.5 Differentiable Reasoning with Provenance

We now elucidate how provenance also supports differentiable reasoning. Let all the probabilities in the EDB form a vector $\vec{r} \in \mathbb{R}^n$, and the probabilities in the resulting IDB form a vector $\vec{y} \in \mathbb{R}^m$. Differentiation concerns deriving output probabilities $\vec{y}$ as well as the derivative $\nabla\vec{y} = \frac{\partial \vec{y}}{\partial \vec{r}} \in \mathbb{R}^{m \times n}$.

In Scallop, one can obtain these using a *differentiable provenance* (DP). DPs share the same external interface—let the input tag space $I = [0, 1]$ and output tag space O be the space of *dual-numbers* $\mathbb{D}$ (Fig. 12). Now, each input tag $r_i \in [0, 1]$ is a probability, and each output tag $\hat{y}_j = (y_j, \nabla y_j)$ encapsulates the output probability y_j and its derivative w.r.t. inputs, ∇y_j. From here, we can obtain our expected output $\vec{y}$ and $\nabla\vec{y}$ by stacking together y_j-s and ∇y_j-s respectively.

Scallop provides 8 configurable built-in DPs with different empirical advantages in terms of runtime efficiency, reasoning granularity, and performance. In this section, we elaborate upon 3 simple but versatile DPs, whose definitions are shown in Fig. 11. In the following discussion, we use r_i to denote the i-th element of $\vec{r}$, where i is called a *variable* (ID). Vector $\vec{\mathbf{e}}_i \in \mathbb{R}^n$ is the standard basis vector where all entries are 0 except the i-th entry.

4.5.1 `diff-max-min-prob` *(*`dmmp`*).* This provenance is the differentiable version of `mmp`. When obtaining r_i from an input tag, we transform it into a dual-number by attaching $\vec{\mathbf{e}}_i$ as its derivative. Note that throughout the execution, the derivative will always have at most one entry being non-zero and, specifically, 1 or -1. The saturation check is based on equality of the probability part only, so that the derivative does not affect termination. All of its operations can be implemented by algorithms with time complexity $O(1)$, making it extremely runtime-efficient.

4.5.2 `diff-add-mult-prob` *(*`damp`*).* This provenance has the same internal tag space, tagging function, and recover function as `dmmp`. As suggested by its name, its disjunction and conjunction operations are just + and · for dual-numbers. When performing disjunction, we clamp the real part of the dual-number obtained from performing +, while keeping the derivative the same. The saturation function for `damp` is designed to always returns true to avoid non-termination. But this decision makes it less suitable for complex recursive programs. The time complexity of operations in `damp` is $O(n)$, which is slower than `dmmp` is but still very efficient in practice.

Provenance	T	$\mathbf{0}$	$\mathbf{1}$	$t_1 \oplus t_2$	$t_1 \otimes t_2$	$\ominus\, t$	$t_1 \ominus t_2$	$\tau(r_i)$	$\rho(t)$
`diff-max-min-prob`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\max(t_1, t_2)$	$\min(t_1, t_2)$	$\hat{1} - t$	$t_1^{\text{fst}} == t_2^{\text{fst}}$	$(r_i, \vec{\mathbf{e}}_i)$	t
`diff-add-mult-prob`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\text{clamp}(t_1 + t_2)$	$t_1 \cdot t_2$	$\hat{1} - t$	true	$(r_i, \vec{\mathbf{e}}_i)$	t
`diff-top-k-proofs`	Φ	$\emptyset$	$\{\emptyset\}$	$t_1 \vee_k t_2$	$t_1 \wedge_k t_2$	$\neg_k\, t$	$t_1 == t_2$	$\{\{\text{pos}(i)\}\}$	$\text{WMC}(t, \Gamma)$

Fig. 11. Definitions of three differentiable provenances.

$$\hat{a}_i = (a_i, \nabla a_i) \in \mathbb{D} \qquad \hat{a}_1 + \hat{a}_2 = (a_1 + a_2, \nabla a_1 + \nabla a_2) \qquad \min(\hat{a}_1, \hat{a}_2) = \hat{a}_i, \text{ where } i = \text{argmin}_i(a_i)$$
$$\hat{0} = (0, \vec{0}) \qquad \hat{a}_1 \cdot \hat{a}_2 = (a_1 \cdot a_2, a_2 \cdot \nabla a_1 + a_1 \cdot \nabla a_2) \qquad \max(\hat{a}_1, \hat{a}_2) = \hat{a}_i, \text{ where } i = \text{argmax}_i(a_i)$$
$$\hat{1} = (1, \vec{0}) \qquad -\hat{a}_1 = (-a_1, -\nabla a_1) \qquad \text{clamp}(\hat{a}_1) = (\text{clamp}_0^1(a_1), \nabla a_1)$$

Fig. 12. Operations on dual-number $\mathbb{D} \triangleq [0, 1] \times \mathbb{R}^n$, where n is the number of input probabilities.

$$\text{(Variable)} \quad i \in 1 \ldots n \qquad \varphi_1 \vee_k \varphi_2 = \text{top}_k(\varphi_1 \cup \varphi_2)$$
$$\text{(Literal)} \quad \nu ::= \text{pos}(i) \mid \text{neg}(i) \qquad \varphi_1 \wedge_k \varphi_2 = \text{top}_k(\{\, \eta \mid (\eta_1, \eta_2) \in \varphi_1 \times \varphi_2, \eta = \eta_1 \cup \eta_2, \eta \text{ no conflict} \,\})$$
$$\text{(Proof)} \quad \eta ::= \{\nu_1, \nu_2, \ldots\} \qquad \neg_k \varphi = \text{top}_k(\text{cnf2dnf}(\{\{\neg \nu \mid \nu \in \eta\} \mid \eta \in \varphi\}))$$
$$\text{(Formula)} \quad \Phi \ni \varphi ::= \{\eta_1, \eta_2, \ldots\} \qquad \Gamma(i) = (r_i, \vec{\mathbf{e}}_i)$$

Fig. 13. Definitions used for `diff-top-k-proofs` provenance.

4.5.3 `diff-top-k-proofs` *(*`dtkp`*).* This provenance extends the *top-k proofs* semiring originally proposed in [Huang et al. 2021] to additionally support negation and aggregation. Shown in Fig. 11 and 13, the tags of `dtkp` are boolean formulas $\varphi \in \Phi$ in *disjunctive normal form* (DNF). Each conjunctive clause in the DNF is called a *proof* η. A formula can contain at most k proofs, where k is a tunable hyper-parameter. Throughout execution, boolean formulas are propagated with $\vee_k$, $\wedge_k$, and $\neg_k$, which resemble *or*, *and*, and *not* on DNF formulas. At the end of these computations, top_k is applied to keep only k proofs with the highest *proof probability*:

$$\Pr(\eta) = \textstyle\prod_{\nu \in \eta} \Pr(\nu), \quad \Pr(\text{pos}(i)) = r_i, \quad \Pr(\text{neg}(i)) = 1 - r_i. \tag{1}$$

When merging two proofs during $\wedge_k$, there might be conflicting literals, e.g. $\text{pos}(i)$ and $\text{neg}(i)$, in which case we remove the whole proof. To take negation $\neg_k$ on φ, we first negate all the literals to obtain a *conjunctive normal form* (CNF) equivalent to $\neg\varphi$. Then we perform cnf2dnf operation (conflict check included) to convert it back to a DNF. To obtain the output dual-number $\hat{y}_j$ from a DNF formula φ_j, the tag for j-th output tuple, we adopt a differentiable *weighted-model-counting* (WMC) procedure [Manhaeve et al. 2021]. WMC computes the weight of a boolean formula φ_j given the weights of individual variables. Concretely, $\hat{y}_j = \text{WMC}(\varphi_j, \Gamma)$ where $\Gamma(i) = (r_i, \vec{\mathbf{e}}_i)$ is the mapping from variables to their dual-numbers. Note that WMC is #P-complete, and is the main contributor to the runtime when using this provenance. The tunable k enables the user to balance between runtime and reasoning granularity.

4.6 Practical Considerations

We finally discuss some practical aspects concerning ScLRam extensions and provenance selection.

Additional Features. We only presented the syntax and semantics of the core fragment of ScLRam. ScLRam additionally supports the following useful features: 1) sampling operations, and the provenance extension supporting them, 2) group-by aggregations, 3) tag-based early removal and its extension in provenance, and 4) mutually exclusive tags in `dtkp`.

Provenance Selection. A natural question is how to select a differentiable provenance for a given Scallop application. Based on our empirical evaluation in Section 6, `dtkp` is often the best performing one, and setting $k = 3$ is usually a good choice for both runtime efficiency and learning performance. This suggests that a user can start with `dtkp` as the default. In general, the provenance selection process is similar to the process of hyperparameter tuning common in machine learning.

5 IMPLEMENTATION

We implemented the core Scallop system in 45K LoC of Rust. The LoC of individual modules is shown in Table 1. Within the compiler, there are two levels of intermediate representations, front-IR and back-IR, between the surface language and the ScIRam language. In front-IR, we perform analyses such as type inference and transformations such as desugaring. In back-IR, we generate query plans and apply optimizations. The runtime operates directly on ScIRam and is based on semi-naive evaluation specialized for tagged semantics. There are *dynamic* and *static* runtimes for interpreted and compiled ScIRam programs. Currently, all computation is on CPU only, but can be parallelized per-batch for machine learning.

Table 1. LoC of core Scallop modules.

Module	LoC
Compiler	19K
Runtime	16K
Interpreter	2K
`scallopy`	4K

Scallop can be used through different interfaces such as interpreter, compiler, and interactive terminal. It also provides language bindings such as `scallopy` for Python and `scallop-wasm` for WebAssembly and JavaScript. With `scallopy`, Scallop can be seamlessly integrated with machine learning frameworks such as PyTorch, wherein the `scallopy` module is treated just like any other PyTorch module. When `jit` is specified in `scallopy`, Scallop programs can be *just-in-time* (JIT) compiled to Rust, turned into binaries, and dynamically loaded into the Python environment.

Scallop provides 18 built-in provenances (4 for discrete reasoning, 6 for probabilistic, and 8 for differentiable). The WMC algorithm is implemented using *sentential decision diagram* (SDD) [Darwiche 2011] with naive bottom-up compilation. We allow each provenance to provide its own implementation for operations such as aggregation and sampling. This gives substantial freedom and enables optimizations for complex operations. Our provenance framework is also interfaced with `scallopy`, allowing to quickly create and test new provenances in Python.

6 EVALUATION

We evaluate the Scallop language and framework on a benchmark suite comprising eight neurosymbolic applications. Our evaluation aims to answer the following research questions:

RQ1 How expressive is Scallop for solving diverse neurosymbolic tasks?
RQ2 How do Scallop's solutions compare to state-of-the-art baselines in terms of accuracy?
RQ3 Is the differentiable reasoning module of Scallop runtime-efficient?
RQ4 Is Scallop effective at improving generalizability, interpretability, and data-efficiency?
RQ5 What are the failure modes of Scallop solutions and how can we mitigate them?

In the following sections, we first introduce the benchmark tasks and the chosen baselines for each task (Section 6.1). Then, we answer **RQ1** to **RQ5** in Section 6.2 to Section 6.6 respectively. All Scallop related and runtime related experiments were conducted on a machine with two 20-core Intel Xeon CPUs, four GeForce RTX 2080 Ti GPUs, and 768 GB RAM.

6.1 Benchmarks and Baselines

We present an overview of our benchmarks in Fig. 14. They cover a wide spectrum of tasks involving perception and reasoning. The input data modality ranges from images and videos to natural language texts and knowledge bases (KB). The size of the training dataset is also presented in the figure. We next elaborate on the benchmark tasks and their corresponding baselines.

MNIST-R. : *A Synthetic MNIST Test Suite.* This benchmark is designed to test various features of Scallop such as negation and aggregation. Each task takes as input one or more images of handwritten digits from the MNIST dataset [Lecun et al. 1998] and performs simple arithmetic (sum2, sum3, sum4), comparison (less-than), negation (not-3-or-4), or counting (count-3, count-3-or-4) over the depicted digits. For count-3 and count-3-or-4, we count digits from a set of 8 images. For

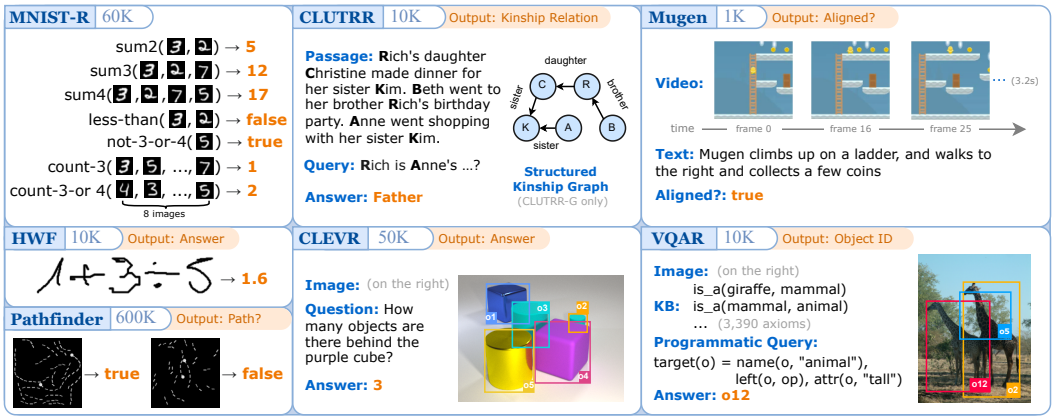


Fig. 14. Visualization of benchmark tasks. Beside the name of each task we specify the size of the training dataset and the output domain. PacMan-Maze is omitted since it has been shown in Section 2.

this test suite, we use a CNN-based model, DeepProbLog (DPL) [Manhaeve et al. 2021], and our prior work [Huang et al. 2021] (Prior) as the baselines.

HWF. : *Hand-Written Formula Parsing and Evaluation.* HWF, proposed in [Li et al. 2020], concerns parsing and evaluating hand-written formulas. The formula is provided in the form of a sequence of images, where each image represents either a digit (0-9) or an arithmetic symbol ($+, -, \times, \div$). Formulas are well-formed according to a grammar and do not divide by zero. The size of the formulas ranges from 1 to 7 and is indicated as part of the input. The goal is to evaluate the formula to obtain a rational number as the result. We choose from [Li et al. 2020] the baselines NGS-m-BS, NGS-RL, and NGS-MAPO, which are *neurosymbolic methods* designed specifically for this task.

Pathfinder. : *Image Classification with Long-Range Dependency.* In this task from [Tay et al. 2020], the input is an image containing two dots that are possibly connected by curved and dashed lines. The goal is to tell whether the dots are connected. There are two subtasks, Path and Path-X, where Path contains 32×32 images and Path-X contains 128×128 ones. We pick as baselines standard CNN and Transformer based models, as well as the state-of-the-art neural models S4 [Gu et al. 2021], S4* [Gu et al. 2022], and SGConv [Li et al. 2022].

PacMan-Maze. : *Playing PacMan Maze Game.* As presented in Section 2, this task tests an agent's ability to recognize entities in an image and plan the path for the PacMan to reach the goal. An RL environment provides the game state image as input and the agent must plan the optimal action {`up`, `down`, `left`, `right`} to take at each step. There is no "training dataset" as the environment is randomized for every session. We pick as baseline a CNN based Deep-Q-Network (DQN). Unlike other tasks, we use the "success rate" metric for evaluation, i.e., among 1000 game sessions, we measure the number of times the PacMan reaches the goal within a certain time-budget.

CLUTRR. : *Kinship Reasoning from Natural Language Context.* In this task from [Sinha et al. 2019], the input contains a natural language (NL) passage about a set of characters. Each sentence in the passage hints at kinship relations. The goal is to infer the relationship between a given pair of characters. The target relation is not stated explicitly in the passage and it must be deduced through a reasoning chain. Our baseline models include RoBERTa [Liu et al. 2019], BiLSTM [Graves et al. 2013], GPT-3-FT (fine-tuned), GPT-3-ZS (zero-shot), and GPT-3-FS (5-shot) [Brown et al. 2020]. In an alternative setup, CLUTRR-G, instead of the NL passage, the structured kinship graph

corresponding to the NL passage is provided, making it a *Knowledge Graph Reasoning* problem. For CLUTRR-G, we pick GAT [Veličković et al. 2017] and CTP [Minervini et al. 2020] as baselines.

Mugen. : *Video-Text Alignment and Retrieval.* Mugen [Hayes et al. 2022] is based on a game called CoinRun [Cobbe et al. 2019]. In the video-text alignment task, the input contains a 3.2 second long video of gameplay footage and a short NL paragraph describing events happening in the video. The goal is to compute a similarity score representing how "aligned" they are. There are two subsequent tasks, Video-to-Text Retrieval (VTR) and Text-to-Video Retrieval (TVR). In TVR, the input is a piece of text and a set of 16 videos, and the goal is to retrieve the video that best aligns with the text. In VTR, the goal is to retrieve text from video. We compare our method with SDSC [Hayes et al. 2022].

CLEVR. : *Compositional Language and Elementary Visual Reasoning [Johnson et al. 2017].* In this visual question answering (VQA) task, the input contains a rendered image of geometric objects and a NL question that asks about counts, attributes, and relationships of objects. The goal is to answer the question based on the image. We pick as baselines NS-VQA [Yi et al. 2018] and NSCL [Mao et al. 2019], which are *neurosymbolic methods* designed specifically for this task.

VQAR. : *Visual-Question-Answering with Common-Sense Reasoning.* This task, like CLEVR, also concerns VQA but with three salient differences: it contains real-life images from the GQA dataset [Hudson and Manning 2019]; the queries are in a programmatic form, asking to retrieve objects in the image; and there is an additional input in the form of a common-sense knowledge base (KB) [Gao et al. 2019] containing triplets such as (giraffe, is-a, animal) for common-sense reasoning. The baselines for this task are NMNs [Andreas et al. 2016] and LXMERT [Tan and Bansal 2019].

6.2 RQ1: Our Solutions and Expressivity

To answer **RQ1**, we demonstrate our Scallop solutions to the benchmark tasks (Table 2). For each task, we specify the interface relations which serve as the bridge between the neural and symbolic components. The neural modules process the perceptual input and their outputs are mapped to (probabilistic) facts in the interface relations. Our Scallop programs subsequently take these facts as input and perform the described reasoning to produce the final output. As shown by the *features* column, our solutions use all of the core features provided by Scallop.

The complete Scallop program for each task is provided in [Li et al. 2023]. These programs are succinct, as indicated by the LoCs in the last column of Table 2. We highlight three tasks, HWF, Mugen, and CLEVR, to demonstrate Scallop's expressivity. For HWF, the Scallop program consists of a formula parser. It is capable of parsing probabilistic input symbols according to a context free grammar for simple arithmetic expressions. For Mugen, the Scallop program is a *temporal specification checker*, where the specification is extracted from NL text to match the sequential events excerpted from the video. For CLEVR, the Scallop program is an interpreter for CLEVR-DSL, a domain-specific functional language introduced in the CLEVR dataset [Johnson et al. 2017].

We note that our prior work [Huang et al. 2021], which only supports positive Datalog, cannot express 5 out of the 8 tasks since they need negation and aggregation, as indicated by columns 'N' and 'A'. Moreover, HWF requires floating point support which is also lacking in our prior work.

Besides diverse kinds of perceptual data and reasoning patterns, the Scallop programs are applied in a variety of learning settings. As shown in Section 2, the program for PacMan-Maze is used in a *online representation learning* setting. For CLUTRR, we write integrity constraints (similar to the one shown in Section 3.2) to derive *semantic loss* used for constraining the language model outputs. For CLUTRR-G, learnable weights are attached to `composition` facts such as `composition(FATHER, MOTHER, GRANDMOTHER)`, which enables to learn such facts from data akin to *rule learning* in ILP.

Table 2. Characteristics of Scallop solutions for each task. Structured input which is not learnt is denoted by *. Neural models used are RoBERTa [Liu et al. 2019], DistilBERT [Sanh et al. 2019], and BiLSTM [Graves et al. 2013] for natural language (NL), CNN and FastRCNN [Girshick 2015] for images, and S3D [Xie et al. 2018] for video. We show the three key features of Scallop used by each solution: (R)ecursion, (N)egation, and (A)ggregation. †: For MNIST-R, the LoC is 2 for every subtask.

Task	Input	Neural Net	Interface Relation(s)	Scallop Program	Features			LoC
					R	N	A	
MNIST-R	Images	CNN	`digit`(*id*, *digit*)	Arithmetic, comparison, negation, and counting.		✔	✔	2†
HWF	Images	CNN	`symbol`(*id*, *symbol*) `length`(*len*)	Parses and evaluates formula over recognized symbols.	✔			39
Pathfinder	Image	CNN	`dot`(*id*) `dash`(*from_id*, *to_id*)	Checks if the dots are connected by dashes.	✔			4
PacMan-Maze	Image	CNN	`actor`(x, y) `enemy`(x, y) `goal`(x, y)	Plans the optimal action by finding an enemy-free path from actor to goal.	✔	✔	✔	31
CLUTRR (-G)	NL	RoBERTa	`kinship`(*rela*, *sub*, *obj*)	Deduces queried relationship by recursively applying learnt composition rules.	✔	✔	✔	8
	Query*	–	`question`(*sub*, *obj*)					
	Rule	–	`composition`(r_1, r_2, r_3)					
Mugen	Video	S3D	`action`(*frame*, *action*, *mod*)	Checks if events specified in NL text match the actions recognized from the video.	✔	✔	✔	46
	NL	DistilBERT	`expr`(*expr_id*, *action*) `mod`(*expr_id*, *mod*)					
CLEVR	Image	FastRCNN	`obj_attr`(*obj_id*, *attr*, *val*) `obj_rela`(*rela*, o_1, o_2)	Interprets CLEVR-DSL program (extracted from question) on scene graph (extracted from image).	✔	✔	✔	51
	NL	BiLSTM	`filter_expr`(*e*, *ce*, *attr*, *val*) `count_expr`(*e*, *ce*), …					
VQAR	Image	FastRCNN	`obj_name`(*obj_id*, *name*) `obj_attr`(*obj_id*, *val*) `obj_rela`(*rela*, o_1, o_2)	Evaluates query over scene graphs (extracted from image) with the aid of common-sense knowledge base (KB).	✔			42
	KB*	–	`is_a`(*name1*, *name2*), …					

For Mugen, our program is trained in a *contrastive learning* setup, since it requires to maximize similarity scores between aligned video-text pairs but minimize that for un-aligned ones.

6.3 RQ2: Performance and Accuracy

To answer **RQ2**, we evaluate the performance and accuracy of our methods in terms of two aspects: 1) the best performance of our solutions compared to existing baselines, and 2) the performance of our solutions with different provenance structures (`dmmp`, `damp`, `dtkp` with different k).

We start with comparing our solutions against selected baselines on all the benchmark tasks, as shown in Fig. 15, Table 3, and Fig. 17. First, we highlight two applications, PacMan-Maze and CLUTRR, which benefit the most from our solution. For PacMan-Maze, compared to DQN, we obtain a 1,000× speed-up in terms of training episodes, and a near perfect success rate of 99.4%. Note that our solution encodes environment dynamics (i.e. game rules) which are unavailable and hard to incorporate in the DQN model. For CLUTRR, we obtain a 25% improvement over baselines, which includes GPT-3-FT, the state-of-the-art large language model fine-tuned on the CLUTRR dataset. Next, for tasks such as HWF and CLEVR, our solutions attain comparable performance, even compared to neurosymbolic baselines NGS-m-BS, NSCL, and NS-VQA specifically designed for each task. On Path and Path-X, our solution obtains a 4% accuracy gain over our underlying model CNN and even outperforms a carefully designed transformer based model S4.

The performance of the Scallop solution for each task depends on the chosen provenance structure. As can be seen from Table 3 and Figs. 15–17, although `dtkp` is generally the best-performing one, each presented provenance is useful, e.g., `dtkp` for PacMan-Maze and VQAR, `damp` for less-than

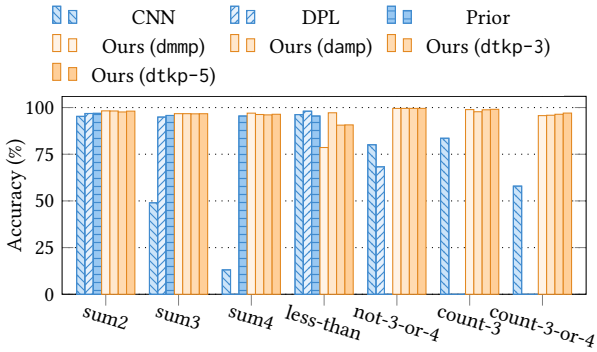


Fig. 15. MNIST-R suite accuracy comparison.

Table 3. PacMan-Maze performance.

Method	Scallop			DQN
	dmmp	damp	dtkp-1	
Succ Rate	8.80%	7.84%	**99.40%**	84.90%
#Episodes	50	50	**50**	50K

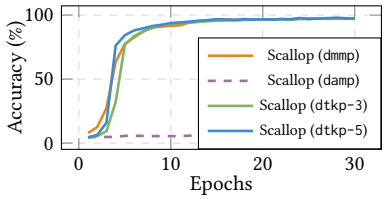


Fig. 16. HWF learning curve.

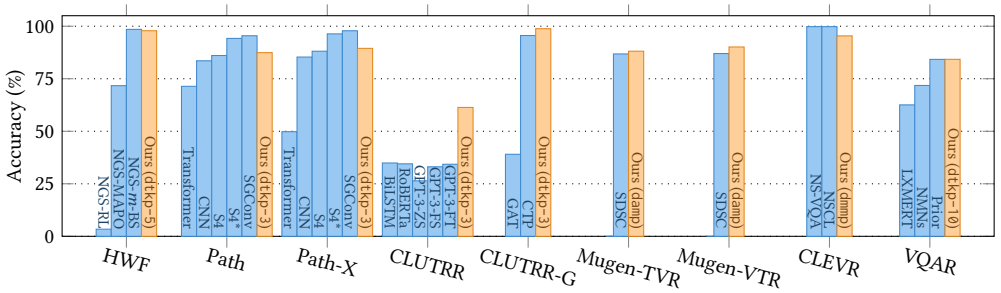


Fig. 17. Overall benchmark accuracy comparison. The best-performing provenance structure for our solution is indicated for each task. Among the shown tasks, `dtkp` performs the best on 6 tasks, `damp` on 2, and `dmmp` on 1.

(MNIST-R) and Mugen, and `dmmp` for HWF and CLEVR. Note that under positive Datalog, Scallop's `dtkp` is identical to [Huang et al. 2021], allowing us to achieve similar performance. In conclusion, allowing configurable provenance helps tailor our methods to different applications.

6.4 RQ3: Runtime Efficiency

We evaluate the runtime efficiency of Scallop solutions with different provenance structures and compare it against baseline neurosymbolic approaches. As shown in Table 4, Scallop achieves substantial speed-up over DeepProbLog (DPL) on MNIST-R tasks. DPL is a probabilistic programming system based on Prolog using exact probabilistic reasoning. As an example, on sum4, DPL takes 40 days to finish only 4K training samples, showing that it is prohibitively slow to use in practice. On the contrary, Scallop solutions can finish a training epoch (15K samples) in minutes without sacrificing testing accuracy (according to Fig. 15). For HWF, Scallop achieves comparable runtime efficiency, even when compared against the hand-crafted and specialized NGS-m-BS method.

Comparing among provenance structures, we see significant runtime blowup when increasing k for `dtkp`. This is expected as increasing k results in larger boolean formula tags, making the WMC procedure exponentially slower. In practice, we find $k = 3$ for `dtkp` to be a good balance point between runtime efficiency and reasoning granularity. In fact, `dtkp` generalizes DPL, as one can set an extremely large $k \geq 2^n$ (n is the total number of input facts) for exact probabilistic reasoning.

6.5 RQ4: Generalizability, Interpretability, and Data-Efficiency

We now consider other important desirable aspects of machine learning models besides accuracy and runtime, such as generalizability on unseen inputs, interpretability of the outputs, and data-efficiency of the training process. For brevity, we focus on a single benchmark task in each case.

Table 4. Runtime efficiency comparison on selected benchmark tasks. Numbers shown are average training time (sec.) per epoch. Our variants attaining the best accuracy are indicated in bold.

Task	Scallop				Baseline
	dmmp	damp	dtkp-3	dtkp-10	
sum2	**34**	88	72	185	21,430 (DPL)
sum3	**34**	**119**	71	1,430	30,898 (DPL)
sum4	**34**	154	77	4,329	timeout (DPL)
less-than	35	**42**	34	43	2,540 (DPL)
not-3-or-4	37	**33**	**33**	**34**	3,218 (DPL)
HWF	89	107	**120**	8,435	79 (NGS-m-BS)
CLEVR	**1,964**	1,618	2,325	timeout	–

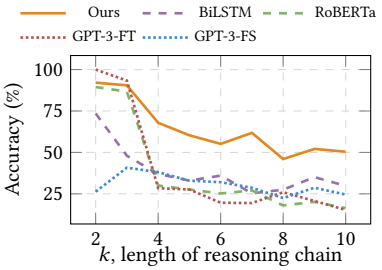


Fig. 18. Systematic generalizability on CLUTRR dataset.

Fig. 19. The predicted most likely (action, mod) pair for example video segments from Mugen dataset.

Table 5. Testing accuracy of various methods on HWF when trained with only a portion of the data. Numbers are in percentage (%).

%Train	Scallop	NGS		
	dtkp-5	RL	MAPO	m-BS
100%	97.85	3.4	71.7	98.5
50%	95.7	3.6	9.5	95.7
25%	92.95	3.5	5.1	93.3

We evaluate Scallop's generalization ability for the CLUTRR task. Each data-point in CLUTRR is annotated with a parameter k denoting the length of the reasoning chain to infer the target kinship relation. To test different solutions' *systematic generalizability*, we train them on data-points with $k \in \{2, 3\}$ and test on data-points with $k \in \{2, \ldots, 10\}$. As shown in Fig. 18, the neural baselines suffer a steep drop in accuracy on the more complex unseen instances, whereas the accuracy of Scallop's solution degrades more slowly, indicating that it is able to generalize better.

Next, we demonstrate Scallop's interpretability on the Mugen task. Although the goal of the task is to see whether a video-text pair is aligned, the perceptual model in our method extracts interpretable symbols, i.e., the action of the controlled character at a certain frame. Fig. 19 shows that the predicted (action, mod) pairs perfectly match the events in the video. Thus, our solution not only tells whether a video-text pair is aligned, but also *why* it is aligned.

Lastly, we evaluate Scallop's data-efficiency on the HWF task, using lesser training data (Table 5). While methods such as NGS-MAPO suffer significantly when trained on less data, Scallop's testing accuracy decreases slowly, and is comparable to the data-efficiency of the state-of-the-art neurosymbolic NGS-m-BS method. PacMan-Maze task also demonstrates Scallop's data-efficiency, as it takes much less training episodes than DQN does, while achieving much higher success rate.

6.6 RQ5: Analysis of Failure Modes

Compared to purely neural models, Scallop solutions provide more transparency, allowing programmers to debug effectively. By manually checking the interface relations, we observed that the main source of error lies in inaccurate predictions from the neural components. For example, the RoBERTa model for CLUTRR correctly extracts only 84.69% of kinship relations. There are two potential causes—either the neural component is not powerful enough, or our solution is not providing adequate supervision to train it. The former can be addressed by employing better neural architectures or more data. The latter can be mitigated in different ways, such as tuning the selected provenance or incorporating integrity constraints (discussed in Section 3.2) into training and/or inference. For instance, in PacMan-Maze, the constraint that "there should be no more than one *goal* in the arena" helps to improve robustness.

7 RELATED WORK

We survey related work in four different but overlapping domains: provenance reasoning, Datalog and logic programming, probabilistic and differentiable programming, and neurosymbolic methods.

Relational Algebra and Provenance. Relational algebra is extensively studied in databases [Abiteboul et al. 1995], and extended with recursion [Jachiet et al. 2020] in Datalog. The provenance semiring framework [Green et al. 2007] was first proposed for positive relational algebra (RA^+) and later extended with difference [Amsterdamer et al. 2011] and fixed-point [Dannert et al. 2021]. It is deployed in variants of Datalog [Khamis et al. 2021] and supports applications such as program synthesis [Si et al. 2019]. Scallop employs an extended provenance semiring framework and demonstrates its application to configurable differentiable reasoning.

Datalog is a declarative logic programming language where rules are logical formulas. Despite being less expressive than Prolog, it is widely studied in both the programming language and database communities for language extensions and optimizations [Khamis et al. 2021]. A variety of Datalog-based systems has been built for program analysis [Scholz et al. 2016] and enterprise databases [Aref et al. 2015]. Scallop is also a Datalog-based system that extends [Huang et al. 2021] with numerous language constructs such as negation and aggregation.

Probabilistic Programming is a paradigm for programmers to model distributions and perform probabilistic sampling and inference. Such systems include ProbLog [Dries et al. 2015], Pyro [Bingham et al. 2018], Turing [Ge et al. 2018], and PPL [van de Meent et al. 2018]. When integrated with modern ML systems, they are well suited for statistical modeling and building generative models. Scallop also supports ProbLog-style exact probabilistic inference as one instantiation of our provenance framework. But advanced statistical sampling and generative modeling is not yet supported and left for future work.

Differentiable Programming (DP) systems seek to enable writing code that is differentiable. Common practices for DP include symbolic differentiation and automatic differentiation (auto-diff) [Baydin et al. 2015], resulting in popular ML frameworks such as PyTorch [Paszke et al. 2019], TensorFlow [Abadi et al. 2015], and JAX [Bradbury et al. 2018]. However, most of these systems are designed for real-valued functions. Scallop is also a differentiable programming system, but with a focus on programming with discrete, logical, and relational symbols.

Neurosymbolic Methods have emerged to incorporate symbolic reasoning into existing learning systems. Their success has been demonstrated in a number of applications [Chen et al. 2020; Li et al. 2020; Mao et al. 2019; Minervini et al. 2020; Wang et al. 2019; Xu et al. 2022; Yi et al. 2018]. Similar to [Li et al. 2020; Mao et al. 2019; Yi et al. 2018], we focus primarily on solutions involving perception followed by symbolic reasoning. Other ways of incorporating symbolic knowledge include *semantic loss* [Xu et al. 2018, 2022], *program synthesis* [Chen et al. 2021a; Shah et al. 2020], and invoking large language models (LLMs) [Cheng et al. 2022; Zelikman et al. 2023]. Most aforementioned neurosymbolic methods build their own domain-specific language or specialized reasoning components, many of which are programmable in Scallop and are instantiations of our provenance framework [Chen et al. 2020; Mao et al. 2019; Xu et al. 2018, 2022]. TensorLog [Cohen et al. 2017], DPL [Manhaeve et al. 2021], and [Huang et al. 2021] can be viewed as unified neurosymbolic frameworks of varying expressivity. Scallop is inspired by DPL, and additionally offers a more scalable, customizable, and easy-to-use language and framework.

8 CONCLUSION

We presented Scallop, a neurosymbolic programming language for integrating deep learning and logical reasoning. We introduced a declarative language and a reasoning framework based on provenance computations. We showed that our framework is practical by applying it to a variety of

machine learning tasks. In particular, our experiments show that Scallop solutions are comparable and even supersede many existing baselines.

In the future, we plan to extend Scallop in three aspects: 1) Supporting more machine learning paradigms, such as generative modeling, open domain reasoning, in-context learning, and adversarial learning. 2) Further enhancing the usability, efficiency, and expressiveness of Scallop's language and framework. We intend to provide bindings to other ML frameworks such as TensorFlow and JAX, leverage hardware such as GPUs to accelerate computation, and support constructs such as algebraic data types. 3) Applying Scallop to real-world and safety-critical domains. For instance, we intend to integrate it with the CARLA driving simulator [Dosovitskiy et al. 2017] to specify soft temporal constraints for autonomous driving systems. We also intend to apply Scallop in the medical domain for explainable disease diagnosis from electronic health records (EHR) data.

ACKNOWLEDGMENTS

We thank Neelay Velingker, Hanjun Dai, Hanlin Zhang, and Sernam Lin for helpful comments on the presentation and experiments. We thank the anonymous reviewers and artifact reviewers for useful feedback. This research was supported by grants from DARPA (#FA8750-19-2-0201), NSF (#2107429 and #1836936), and ONR (#N00014-18-1-2021).

ARTIFACT AVAILABILITY

All software necessary to reproduce the experiments in this paper is available at [Li et al. 2023]. Additionally, the latest source code of Scallop and its documentation is available at https://github.com/scallop-lang/scallop.

REFERENCES

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

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

Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. On the Limitations of Provenance for Queries With Difference. (2011). arXiv:1105.2255

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural Module Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* https://doi.org/10.1109/CVPR.2016.12

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).* https://doi.org/10.1145/2723372.2742796

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

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).

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: Composable Transformations of Python+NumPy Programs.* https://github.com/google/jax.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Conference on Neural Information Processing Systems (NeurIPS).* arXiv:2005.14165

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). https://doi.org/10.1561/2500000049

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374

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

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)*.

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

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. 2019. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning (ICML)*. PMLR, 1282–1289. arXiv:1812.02341

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

Andrew Cropper and Sebastijan Dumančić. 2022. Inductive Logic Programming At 30: A New Introduction. 74 (2022). https://doi.org/10.1613/jair.1.13507

Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. 2021. Semiring Provenance for Fixed-Point Logic. In *Conference on Computer Science Logic (CSL)*. https://doi.org/10.4230/LIPIcs.CSL.2021.17

Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Conference on Robot Learning (CoRL)*. arXiv:1711.03938

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)*. https://doi.org/10.1007/978-3-319-23461-8_37

Difei Gao, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2019. From Two Graphs to N Questions: A VQA Dataset for Compositional Reasoning on Vision and Commonsense. (2019). arXiv:1908.02962

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

Ross B. Girshick. 2015. Fast R-CNN. (2015). arXiv:1504.08083

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. https://doi.org/10.1109/ICASSP.2013.6638947

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

Albert Gu, Karan Goel, and Christopher Ré. 2021. Efficiently Modeling Long Sequences with Structured State Spaces. (2021). arXiv:2111.00396

Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. 2022. How to Train Your HiPPO: State Space Models with Generalized Orthogonal Basis Projections. (2022). arXiv:2206.12037

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017). https://doi.org/10.1561/2500000010

Thomas Hayes, Songyang Zhang, Xi Yin, Guan Pang, Sasha Sheng, Harry Yang, Songwei Ge, Qiyuan Hu, and Devi Parikh. 2022. MUGEN: A Playground for Video-Audio-Text Multimodal Understanding and GENeration. arXiv:2204.08058

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)*.

Drew A Hudson and Christopher D Manning. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/CVPR.2019.00686

Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaida. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *ACM International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3380567

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. 2017. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2901–2910. https://doi.org/10.1109/cvpr.2017.215

Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. 2015. Image Retrieval Using Scene Graphs. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/CVPR.2015.7298990

Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2021. Convergence of Datalog over (Pre-) Semirings. arXiv:2105.14435

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998). https://doi.org/10.1109/5.726791

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving Quantitative Reasoning Problems with Language Models. (2022).

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)*. https://doi.org/10.48550/arXiv.2006.06649

Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. 2022. What Makes Convolutional Models Great on Long Sequence Modeling? (2022). arXiv:2210.09298

Ziyang Li, Jiani Huang, and Mayur Naik. 2023. *Reproduction package for article "Scallop: A Language for Neurosymbolic Programming"*. https://doi.org/10.5281/zenodo.7804200

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (2019). arXiv:1907.11692

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

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

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

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level Control Through Deep Reinforcement Learning. *Nature* 518, 7540 (2015). https://doi.org/10.1038/nature14236

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

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.18653/v1/D16-1264

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. (2019). arXiv:1910.01108

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)*. https://doi.org/10.1145/2892208.2892226

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)*.

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.24963/ijcai.2019/847

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

Hao Tan and Mohit Bansal. 2019. LXMERT: Learning Cross-Modality Encoder Representations from Transformers. (2019). arXiv:1908.07490

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

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

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph Attention Networks. (2017). arXiv:1710.10903

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

Saining Xie, Chen Sun, Jonathan Huang, Zhuowen Tu, and Kevin Murphy. 2018. Rethinking Spatiotemporal Feature Learning: Speed-Accuracy Trade-offs in Video Classification. In *European Conference on Computer Vision (ECCV)*. https://doi.org/10.1007/978-3-030-01267-0_19

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

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)*.

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)*.

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

Received 2022-11-10; accepted 2023-03-31