A Theoretical Guarantee

A.1 Proof Semiring

Definition A.1 *Given a program $\mathcal{P} = (\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$, the collection of sets of proofs $\mathcal{S}$ is defined to be*

$$\{S \mid S \in \mathbb{P}(\mathbb{P}(\mathcal{F})), \forall F \in S, F \textit{ is a proof}\}.$$

Note that F being a proof implies that there is no disjunction conflict in F. That is,

$$\forall f_1, f_2 \in F, j \in \mathcal{J}, f_1 \in j \implies f_2 \notin j$$

Definition A.2 *The two binary operators $\oplus$ and $\otimes$: $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ are defined as*

$$\begin{aligned} S_1 \oplus S_2 &= S_1 \cup S_2, \\ S_1 \otimes S_2 &= \{F \mid F = F_1 \cup F_2, (F_1, F_2) \in S_1 \times S_2, \\ &\qquad F \textit{ contains no disjunction conflict}\}. \end{aligned}$$

Theorem A.3 $(\mathcal{S}, \oplus, \otimes, \emptyset, \{\emptyset\})$ *forms a commutative semiring, which we call* ***Proof Semiring***.

Proof We show that (1). $\emptyset$ is a $\oplus$ identity, (2). $\{\emptyset\}$ is a $\otimes$ identity, (3). $\oplus$ and $\otimes$ are commutative and associative, (4). multiplication is distributive, and (5). multiply by $\emptyset$ annihilates the operand.

1. $\emptyset$ is a $\oplus$ identity. Given $S \in \mathcal{S}$,
$$S \oplus \emptyset = S \cup \emptyset = S = \emptyset \cup S = \emptyset \oplus S.$$
2. $\{\emptyset\}$ is a $\otimes$ identity. Given $S = \{F_1, F_2, \ldots, F_n\} \in \mathcal{S}$, we have
$$S \otimes \{\emptyset\} = \{F_1 \cup \emptyset, F_2 \cup \emptyset, \ldots, F_2 \cup \emptyset\} = S$$
3. $\oplus$ is commutative: Given S_1 and S_2,
$$S_1 \oplus S_2 = S_1 \cup S_2 = S_2 \cup S_1 = S_2 \oplus S_1.$$
$\otimes$ is commutative: Given S_1 and S_2,
$$S_1 \otimes S_2 = \{F_1^1 \cup F_1^2, \ldots\} = S_2 \otimes S_1$$
$\oplus$ is associative: Given $S_1, S_2, S_3 \in \mathcal{S}$,
$$S_1 \oplus (S_2 \oplus S_3) = S_1 \cup (S_2 \cup S_3) = (S_1 \cup S_2) \cup S_3.$$
$\otimes$ is associative: Given $S_1, S_2, S_3 \in \mathcal{S}$ and $S_i = \{F_1^i, F_2^i, \ldots, F_{n_i}^i\}$. We denote
$$F_{xyz} = \begin{cases} F_x^1 \cup F_y^2 \cup F_z^3 & \text{if no disjunction conflict} \\ \emptyset & \text{otherwise,} \end{cases}$$
where $x \in 1 \ldots n_1$, $y \in 1 \ldots n_2$, $z \in 1 \ldots n_3$. We then have
$$\begin{aligned} S_1 \otimes (S_2 \otimes S_3) &= \{F_{111}, F_{112}, \ldots, F_{n_1 n_2 n_3}\} \\ &= (S_1 \otimes S_2) \otimes S_3 \end{aligned}$$
4. Distributive. Given S_1, S_2, and $S_3 \in \mathcal{S}$ similar to above, we have
$$\begin{aligned} S_1 \otimes (S_2 \oplus S_3) &= S_1 \otimes (S_2 \cup S_3) && (5) \\ &= (S_1 \otimes S_2) \cup (S_1 \otimes S_3) && (6) \\ &= (S_1 \otimes S_2) \oplus (S_1 \otimes S_3) && (7) \end{aligned}$$
5. Multiplying $\emptyset$ annihilates the operand:
$$S_1 \otimes \emptyset = \emptyset.$$

Therefore $(\mathcal{S}, \oplus, \otimes, \emptyset, \{\emptyset\})$ forms a semiring. ■

Theorem A.4 *$\mathcal{S}$ is naturally ordered and ω-complete [17].*

Proof We define a partial order $\leq$ such that $S_1 \leq S_2 \iff S_1 \subset S_2$. Therefore our $\mathcal{S}$ is naturally ordered. In addition, our chain has a strict upper bound which is $\mathbb{P}(\mathcal{F})$, as $\forall S \in \mathcal{S}, S \subset \mathbb{P}(\mathcal{F})$. Hence $\mathcal{S}$ is also ω-complete. ■

Theorem A.5 *The end result S_q can be expressed as*

$$S_q = \bigoplus_{F \text{ derives } q} \Big(\bigotimes_{f \in F} S_f \Big).$$

Proof Under the provenance semiring framework [17], we define a $\mathcal{S}$-Relation $R : \mathcal{G} \to \mathcal{S}$, such that

$$R(f) = \{\{f\}\} = S_f, \forall f \in \mathcal{F}.$$

With $\mathcal{S}$ being a commutative ω-continuous semiring, $\mathcal{Q}$ being a datalog query, and our $\mathcal{S}$-Relation R, by Definition 5.1 [17], we have

$$\mathcal{Q}(R)(t) = \bigoplus_{\tau \text{ yields } t} \Big(\bigotimes_{t' \in \mathit{leaves}(\tau)} R(t') \Big),$$

where τ ranges over all $\mathcal{Q}$-derivation trees for t. In our case, we seek the result $t = q$, which is, $S_q = \mathcal{Q}(R)(q)$. At the same time, we know that τ is a derivation tree for q and its leaf nodes t' are from our input facts $\mathcal{F}$. Note that $\mathit{leaves}(\tau)$ is simply a proof F in our case and each $t' \in \mathit{leaves}(\tau)$ is an input fact $f \in \mathcal{F}$. Therefore we know that $t' \in \mathcal{F}$ and $R(t') = S_{t'}$. At last, we can express S_q as

$$S_q = \mathcal{Q}(R)(q) = \bigoplus_{F \text{ derives } q} \Big(\bigotimes_{f \in F} S_f \Big),$$

as expected. ■

Proposition A.6 $|S_q| = \mathcal{O}(2^{|\mathcal{F}|})$.

Proof (Sketch) Theoretically, $2^{|\mathcal{F}|}$ is the absolute upper bound as there could be at most $2^{|\mathcal{F}|}$ proofs, given by that each input fact $f \in \mathcal{F}$ can be in or not in a proof. ■

In reality, this upper bound can rarely be achieved. The actual size of S_q is always determined by various factors including input facts, rules, and disjunctions.

A.2 Top-k Proof Semiring

We repeat our definitions of $\oplus^{(k)}$ and $\otimes^{(k)}$ here:

Definition A.7 *With a $\texttt{Top}_k : \mathcal{S} \to \mathcal{S}$ defined as keeping the top-k proofs, we define*

$$S_1 \otimes^{(k)} S_2 = \texttt{Top}_k(S_1 \otimes S_2),$$
$$S_1 \oplus^{(k)} S_2 = \texttt{Top}_k(S_1 \oplus S_2).$$

Proposition A.8 *The approximated set of proofs $\tilde{S}_q$ can be expressed as*

$$\tilde{S}_q = \bigoplus^{(k)}_{F \text{ derives } q} \Big(\bigotimes^{(k)}_{f \in F} S_f \Big).$$

Proof (Sketch) First show that $\mathcal{S}$ still form a semiring under $\oplus^{(k)}$ and $\otimes^{(k)}$ with the exact same proof as in **Theorem A.3**. Then follow **Theorem A.5** to show that this expression still holds. ■

Proposition A.9 $|\tilde{S}_q| = \mathcal{O}(k)$.

Proof (Sketch) This follows directly from the definition of $\oplus^{(k)}$ and $\otimes^{(k)}$ as at each step the size of the resulting set of proofs is capped by k. ■

A.3 Approximation Analysis

Proposition A.10 *We give an approximation error bound*

$$|\Pr(S_q) - \Pr(\tilde{S}_q)| \leq \textstyle\sum_{F \in S_q \setminus \tilde{S}_q} \Pr(F).$$

This is a loose bound given by the difference between S_q and $\tilde{S}_q$. Equality happens when all the proofs in S_q are disjoint.

Proposition A.11 *For a program $\mathcal{P} = (\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$, if $\mathcal{J} = \emptyset$, then we have $\tilde{S}_q = \texttt{Top}_k(S_q)$.*

The proof of this proposition can be found in Theorem 1 of [12]. Under that setting, there is no $\mathcal{J}$ and therefore $\mathcal{J} = \emptyset$. At the same time the top-k derivation tree is equivalent to our top-k proof.

Category	Function Name
Scene Graph	INITIAL, FIND NAME, FIND ATTR, RELATE, RELATED REVERSE
Knowledge Graph	FIND KG, FIND HYPERNYM
Logic Operators	AND, OR

Table 3: Basic functions used to generate questions in VQAR.

B VQAR Dataset Collection

B.1 Dataset Generation

We focus on the task of multi-hop VQA with external common-sense knowledge. For this purpose, we generate an object retrieval VQA dataset, called VQAR, by building upon two existing datasets, GQA [18] and CRIC [16]. These datasets comprise real-world images from the Visual Genome and have complementary qualities necessary for our task. In particular, we use curated scene graphs of the images from the GQA dataset, and we use curated knowledge graphs related to visual questions from the CRIC dataset.

Scene and Knowledge Graphs. Starting with the image and scene graph pairs from the GQA dataset, we further pre-process the scene graphs to generate cleaner questions, as follows. We only include the top 500 most frequently occurring object names, which covers more than 88% of all object occurrences. We retain 609 attributes and 229 relationships after normalizing their names. Finally, we ensure that every image has more than 5 objects so that its scene graph is complex enough. After pre-processing, we are left with 80,178 images with their scene graphs.

The knowledge graph provided by the CRIC dataset comprises triplets of the form $\langle \mathsf{e}_1, \mathsf{r}, \mathsf{e}_2 \rangle$, where e_1 and e_2 are two entities, and r describes a relationship between them, e.g., $\langle \mathsf{giraffe}, \mathsf{is\_a}, \mathsf{animal} \rangle$. We represent each type of relationship as a separate binary relation. There are 10 different types of relationships, such as is_a, used_for, and capable_of. We considered two alternatives to CRIC: OK-VQA [25] and KB-VQA [34]. OK-VQA includes common-sense knowledge as part of the question itself, and thus precludes multi-hop reasoning.

KB-VQA comprises over 160M probabilistic common-sense knowledge triplets drawn from Wikilinks, but is noisy.

Programmatic Query Generation. Existing programmatic VQA questions typically seek aggregated results which makes them liable to exploitable bias. For instance, a binary choice question may be answered by an educated guess without using reasoning. We therefore generate object identification queries that require reasoning to varying degrees. Such queries are harder to exploit, since objects vary from scene to scene.

We use GQA's domain specific language to generate programmatic queries for our purpose. Such a query is composed of a functions sequence that successively identify a set of objects, where the final set of objects are the targets to our query. We define a suite of 9 such functions as shown in Table 3. Consider for instance the RELATE function. Viewing the scene graph as a relation $\langle \mathsf{subject}, \mathsf{predicate}, \mathsf{object} \rangle$, this function identifies the object, given the subject and predicate. Then, the natural language question in Figure 2 corresponds to the following programmatic query:

[INITIAL, RELATE(left), FIND HYPERNYM(animal), FIND ATTR(tall)]

The number of clauses n determines the degree of multi-hop reasoning in the query, which we call a query of type Cn. Thus, the above example is a query of type C4. Furthermore, such queries are straightforward to translate into Datalog, allowing them to be executed using Scallop. The Datalog counterpart of the above query is also shown in Figure 2.

Our query generation procedure always starts with the INITIAL function which refers to all objects in the scene graph. It then traverses through the scene graph and the knowledge graph to identify valid clauses to append to the query. Lastly, we execute the resulting query using Scallop to obtain the ground truth answer. We control the difficulty of the query by the number of its clauses.

Since we are not targeting the natural language questions, we only generate these questions in functional program form. For each image, we generate 10 different question and answer pairs for each clause length 2 to 6, to obtain 4 million data points in total. We split them into training (60%),

validation (10%), and testing (30%) sets, and ensure that all the questions about the same image occur within the same split to test generalizability.

C Experiments

C.1 Synthetic Experiment Setup

Models. Our perception model uses two convolutional layers and two fully connected layers, which takes in the MNIST image as input, and output a distribution on 10 possible numbers, 0-9. This model is trained from scratch in an end-to-end fashion.

Training Hyper-parameters. The learning rate for both DeepProbLog and Scallop is 0.01; the batch sizes for Scallop is 64, and 2 for DeepProbLog, as batch size 64 for DeepProbLog converges too slow. We set the epoch size to 20, where both of the methods converge before 5 epochs.

Evaluation Metric. Our evaluation metric is accuracy. If the predicted outcome is the same as the correct one, the accuracy is 1, otherwise, the accuracy is 0.

C.2 VQAR Experiment Setup

Models. Our perception model uses pre-trained fixed-weight Mask RCNN and ResNet models, which take as input an image and produces feature vectors (along with bounding boxes). Then, input facts representing names, attributes, and object relationships are extracted by 3 separate trainable MLP classifiers. We note that these classifiers integrated with our reasoning engine are trained from scratch in an end-to-end fashion. We also note that to ensure a fair comparison, the visual input (features + bounding boxes) we feed to all baselines (including LXMERT) are the same.

Baselines. We use three baselines that are representative of different state-of-the-art approaches to combining perception and reasoning: (1). Neural Module Network (NMNs), which uses a set of neural modules, one per basic function, (2). DATALOG-RL, a reinforcement learning approach supervised by a discrete logic reasoning engine, (3). DeepProbLog, a probabilistic logic programming approach, and (4). LXMERT, a transformer based approach.

Dataset. To evaluate performance, we sample 50K tasks from the training split, 5K from the validation split, and 5K from the testing split. To measure generalizability and sample complexity, we sample 10 to 10K tasks of type C2 for training, and 1K tasks each of type C2 to C6 for testing.

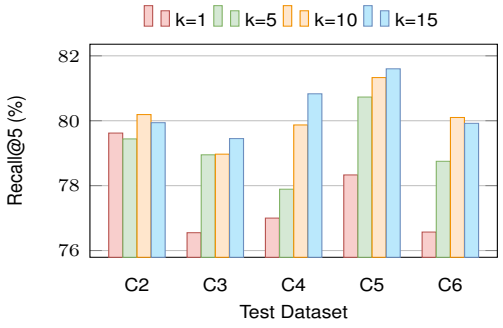
Training Hyper-parameters. All the models converge under 20 training epochs. The learning rate is tuned and is 0.0001 for Scallop, NMN, and DPL, 0.00001 for LXMERT. We select the loss function to be binary cross-entropy loss, except DPL that only supports cross-entropy loss. With batch size 16 and $k = 10$, Scallop achieves the best accuracy in reasonable training time. All the optimizers are Adam.

Model Size Comparison. The model for Scallop, datalog-RL and DeepProbLog are the same, so they share the same model size: 10.91MB for attribute classification, 14.67MB for name classification, 17.78MB for relation classification. The neural modular networks method contains 8 modular network. The `and` and `or` modules are 0.02MB, the `find_name` and `find_attribute` modules are 9.63MB, the `find_hypername` and `find_KG` are 8.61MB, the `relate` and `relate_reverse` modules are 18.06MB. The LXMERT method uses a large pretrained module, which is 836MB.

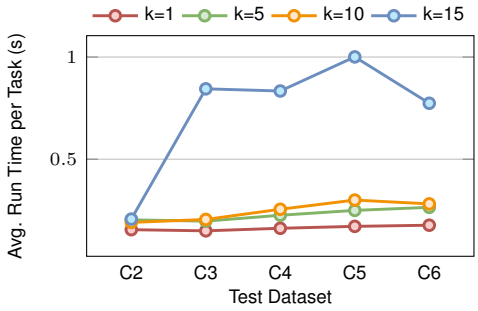
Evaluation Metric. Since our tasks essentially involve object retrieval, any ground truth label is a set of object IDs that satisfy the constraints stated in the question. For set comparison, we select the recall@5 rate as the evaluation metric. It assesses the recall on the top 5 probable predictions.

C.3 VQAR Topk

We compare the performance of Scallop under different choices of k: 1, 5, 10, and 15. We train on 10K C2 tasks and test on 1K tasks of varying clause length. As shown in Figure 12, We observe that the recall@5 score increases as k grows, as expected. However, the larger the k, the longer it takes to process a single task at training time. Our running time increases modestly from $k = 1$ to $k = 10$, and more dramatically when $k = 15$. We thus confirm that Scallop can strike a balance between efficiency and accuracy by tuning the k value, and that exact probabilistic reasoning is not required to obtain good performance on VQAR tasks.



(a) Results of training on 10000 C2 tasks and testing on 1000 tasks of types C2-C6. The recall rate grows as k increases from 1 to 15.



(b) Results of training on 10000 C2 tasks and testing on 1000 tasks of types C2-C6. Running time grows as k increases from 1 to 15.

Figure 12: Comparison of Scallop across different choices of k.

Dataset (Clause-n)	2	3	4	5	6
Timeout rate (%)	54.5%	69.7%	74.4%	70.7%	74.0%

Table 5: DeepProbLog timeout rate on 1000 tasks of types C2-C6.

C.4 DeepProbLog

We give a more fine-grained analysis of the DeepProbLog performance. In particular, we investigate the relation between timeout rate and query complexity. Again, our timeout is set to 10 seconds.

Test Dataset	Timeout Rate w/ KG	Timeout Rate w/ Rela
1000 C2	100%	21.43%
1000 C3	91.55%	73.09%
1000 C4	88.79%	70.74%
1000 C5	88.54%	62.27%
1000 C6	87.85%	75.20%

Table 4: Success Rates of DeepProbLog

Table 5. In the above table, column 2 shows DeepProbLog's timeout rate when at least one KG-related clause is presented in the programmatic query. It is worth noting that DeepProbLog performs the worst on the C2 dataset. In C2, the KG-related clause is the only clause in the programmatic query other than the first INITIAL clause. This implies that there is no constraint posed around the KG clause, leading to a huge amount of possible proofs, and in turn causing the timeout.

Column 3 shows DeepProbLog's timeout rate when at least one relation-related clause is presented in the programmatic query. As one would expect, the more relation is included in the query, the deeper the reasoning will need to be. The table clearly shows that DeepProbLog, without approximation strategy, suffers from handling deep reasoning chain, as that would lead to an exponential amount of proofs.

D Implementation Details

The Scallop implementation is composed of compilation, runtime, and weighted model counting. The compilation part takes in a Datalog program and compiles it into a positive relational algebra form. Then, the runtime executes the generated relational algebra expression and generates the query output with its top-k proofs. Last, the weighted model counting process takes in the query output with the fact probability and calculates the corresponding output probability with gradients. We implement Scallop in Rust for better efficiency.

D.1 Compilation

The compilation process takes in the high-level datalog program and compiles it into an executable form. First, the compiler preprocesses the program, ensures no parsing errors and type errors occur in the given program. Then, it analysis the datalog program and convert it into a mid level positive relational algebra form, which contains **empty**, **union**, **projection**, **selection**, **natural join** and **renaming**. These mid-level relational algebra forms will be further compiled into **join** and **disjunction**, which are directly executable by the runtime.

D.2 Runtime

The runtime execution adopts a bottom-up evaluation strategy with a tagging system for the provenance semiring. It starts with all the input facts tagged with themselves as proofs and keeps applying the rules in the **join** and **disjunction** form until a fixpoint is reached. Whenever a **join** happens on tuple t_1 tagged with F_1, and tuple t_2 tagged with F_2, the generated tuple is tagged with $F_1 \otimes F_2$, where the $\otimes$ is easily configurable. The story is similar for **disjunction** case. In terms of optimization, we adopt the leap join strategy rather than the naive join to increase the evaluation efficiency.

D.3 Weight Model Counting

The weighted model counting algorithm is the same as DeepProbLog. We depend on the sentential decision diagram to realize the weighted model counting process. To realize the gradient calculation, we also implemented a semiring system to carry the additional information during weighted model counting.

Weighted Model Counting v.s. DNF counting. Weighted model counting is a systematical way to calculate the probability of a boolean formula holds, where each variable in the formula is associated with a probability; DNF counting calculates the probability of a DNF formula being true. Since performing DNF counting is less expensive than WMC, it is a promising way to further optimize for the scalability of Scallop. However, we have not incorporated this optimization yet because (a) we are using an off-the-shelf WMC solver and (b) supporting richer forms of reasoning such as negation and aggregation will necessitate WMC. Nevertheless, we acknowledge this optimization possibility, which could be incorporated into the WMC solver to further improve the overall efficiency of Scallop on tasks for which weighted DNF counting is sufficient.

E Synthetic Task Details

E.1 Sum n numbers

The sum n numbers task is an extension from the original MNIST digit recognition task. Instead of recognizing a single digit from the image, this task takes in n images, and recognizes the sum of all the input images. For example, sum( , , 10) is corresponding to a sum2 task. In a scallop program, we have the rule sum(I1, I2, DA + DB) :- digit(I1, DA), digit(I2, DB), where I_j are the image ids in the MNIST dataset. This rule propagates the probability from low level perception in digit(, 3) and digit(, 7) to the high level answer sum(, , 10). We list the code for sum n digit tasks below.

```
Sum2 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Int).
  sum(imgA, imgB, DA + DB) :- digit(imgA, DA), digit(imgB, DB).
}
```

Figure 13: sum 2 numbers.

```
Sum3 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Symbol, Int).
  sum(imgA, imgB, imgC, DA + DB + DC) :-
    digit(imgA, DA), digit(imgB, DB), digit(imgC, DC).
}
```

Figure 14: sum 3 numbers.

```
Sum4 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Symbol, Symbol, Int).
  sum(imgA, imgB, imgC, imgD, DA + DB + DC + DD) :-
    digit(imgA, DA), digit(imgB, DB), digit(imgC, DC), digit(imgD, DD).
}
```

Figure 15: sum 4 numbers.

E.2 Sort-n-numbers

The sort n numbers task is another extension from the original MNISTT digit recognition task. In this task, the input are n images in the MNIST dataset, and the desired output is to sort them in order. For example, sort2(, ,0,1) means the given input and has the order 0, 1 from small to large. In the scallop program to sort two numbers, we have the corresponding rules: sort(imgA, imgB, 0, 1) :- digit(imgA, DA), digit(imgB, DB), DA <= DB. sort(imgA, imgB, 1, 0) :- digit(0, DA), digit(1, DB), DA > DB. This means, if the first number is smaller or equal to the second number, then we given them the order (0, 1), else we give them the order (1, 0). We manually assign the order if two numbers are the same. The corresponding scallop programs are shown below:

```
Sort2 {
  decl digit(Symbol, Int).
  decl sort_2(Int).
  sort_2(0) :- digit(0, DA), digit(1, DB), DA <= DB.
  sort_2(1) :- digit(0, DA), digit(1, DB), DA > DB.
}
```

Figure 16: sort 2 numbers.

```
Sort3 {
  decl digit(Symbol, Int).
  decl sort_3(Int).
  decl digit_abc(Int, Int, Int).
  digit_abc(DA, DB, DC) :- digit(0, DA), digit(1, DB), digit(2, DC).
  sort_3(0) :- digit_abc(DA, DB, DC), DA <= DB, DB <= DC. // 0, 1, 2
  sort_3(1) :- digit_abc(DA, DB, DC), DA <= DC, DC < DB.  // 0, 2, 1
  sort_3(2) :- digit_abc(DA, DB, DC), DB < DA, DA <= DC.  // 1, 0, 2
  sort_3(3) :- digit_abc(DA, DB, DC), DB <= DC, DC < DA.  // 1, 2, 0
  sort_3(4) :- digit_abc(DA, DB, DC), DC < DA, DA <= DB.  // 2, 0, 1
  sort_3(5) :- digit_abc(DA, DB, DC), DC < DB, DB < DA.   // 2, 1, 0
}
```

Figure 17: sort 3 numbers.

```
Sort4 {
  decl digit(Symbol, Int).
  decl sort_4(Int).
  decl digits(Int, Int, Int, Int).
  digits(D0, D1, D2, D3) :- digit(0, D0), digit(1, D1), digit(2, D2), digit(3, D3).
  sort_4(0) :- digits(D0, D1, D2, D3), D0 <= D1, D1 <= D2, D2 <= D3. // 0, 1, 2, 3
  sort_4(1) :- digits(D0, D1, D2, D3), D0 <= D1, D1 <= D3, D3 < D2.  // 0, 1, 3, 2
  sort_4(2) :- digits(D0, D1, D2, D3), D0 <= D2, D2 < D1, D1 <= D3.  // 0, 2, 1, 3
  sort_4(3) :- digits(D0, D1, D2, D3), D0 <= D2, D2 <= D3, D3 < D1.  // 0, 2, 3, 1
  sort_4(4) :- digits(D0, D1, D2, D3), D0 <= D3, D3 < D1, D1 <= D2.  // 0, 3, 1, 2
  sort_4(5) :- digits(D0, D1, D2, D3), D0 <= D3, D3 < D2, D2 < D1.   // 0, 3, 2, 1
  sort_4(6) :- digits(D0, D1, D2, D3), D1 < D0, D0 <= D2, D2 <= D3.  // 1, 0, 2, 3
  sort_4(7) :- digits(D0, D1, D2, D3), D1 < D0, D0 <= D3, D3 < D2.   // 1, 0, 3, 2
  sort_4(8) :- digits(D0, D1, D2, D3), D1 <= D2, D2 < D0, D0 <= D3.  // 1, 2, 0, 3
  sort_4(9) :- digits(D0, D1, D2, D3), D1 <= D2, D2 <= D3, D3 < D0.  // 1, 2, 3, 0
  sort_4(10) :- digits(D0, D1, D2, D3), D1 <= D3, D3 < D0, D0 <= D2. // 1, 3, 0, 2
  sort_4(11) :- digits(D0, D1, D2, D3), D1 <= D3, D3 < D2, D2 < D0.  // 1, 3, 2, 0
  sort_4(12) :- digits(D0, D1, D2, D3), D2 < D0, D0 <= D1, D1 <= D3. // 2, 0, 1, 3
  sort_4(13) :- digits(D0, D1, D2, D3), D2 < D0, D0 <= D3, D3 < D1.  // 2, 0, 3, 1
  sort_4(14) :- digits(D0, D1, D2, D3), D2 < D1, D1 < D0, D0 <= D3.  // 2, 1, 0, 3
  sort_4(15) :- digits(D0, D1, D2, D3), D2 < D1, D1 <= D3, D3 < D0.  // 2, 1, 3, 0
  sort_4(16) :- digits(D0, D1, D2, D3), D2 <= D3, D3 < D0, D0 <= D1. // 2, 3, 0, 1
  sort_4(17) :- digits(D0, D1, D2, D3), D2 <= D3, D3 < D1, D1 < D0.  // 2, 3, 1, 0
  sort_4(18) :- digits(D0, D1, D2, D3), D3 < D0, D0 <= D1, D1 <= D2. // 3, 0, 1, 2
  sort_4(19) :- digits(D0, D1, D2, D3), D3 < D0, D0 <= D2, D2 < D1.  // 3, 0, 2, 1
  sort_4(20) :- digits(D0, D1, D2, D3), D3 < D1, D1 < D0, D0 <= D2.  // 3, 1, 0, 2
  sort_4(21) :- digits(D0, D1, D2, D3), D3 < D1, D1 <= D2, D2 < D0.  // 3, 1, 2, 0
  sort_4(22) :- digits(D0, D1, D2, D3), D3 < D2, D2 < D0, D0 <= D1.  // 3, 2, 0, 1
  sort_4(23) :- digits(D0, D1, D2, D3), D3 < D2, D2 < D1, D1 < D0.   // 3, 2, 1, 0
}
```

Figure 18: sort 4 numbers.

F VQAR Dataset Details

F.1 VQAR Stats

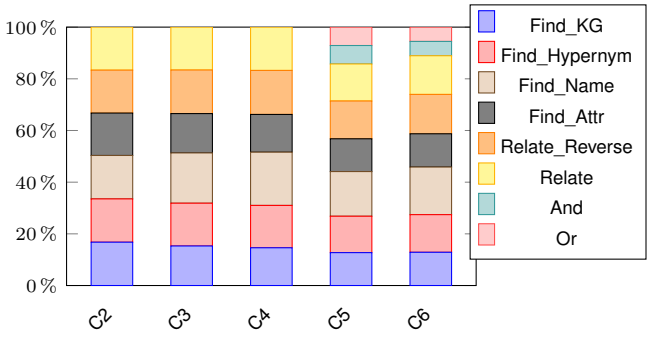


Figure 19: This is the distribution of functions in queries. We only introduce AND and OR for the questions with more than 5 clauses.

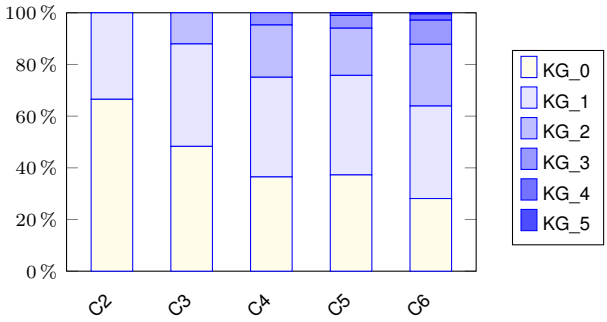


Figure 20: This is the distribution of knowledge graph related function number in queries. FIND_HYPERNAME and FIND_KG are the two basic functions that requires look into the knowledge graph. When the question has more clauses, it is more likely include knowledge base related clauses.

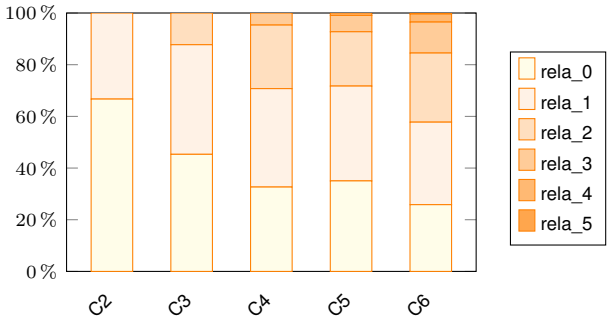


Figure 21: This is the distribution of relation related function number in queries. RELATE and RELATE_REVERSE are the two basic functions that requires look into the knowledge graph. When the question has more clauses, it is more likely include knowledge base related clauses.

F.2 VQAR Examples

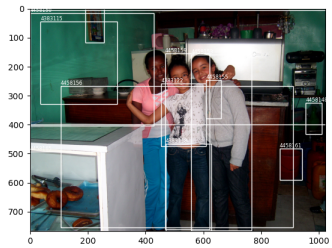
We show 6 images in our VQAR dataset in Figures 22, 23, 24, 25, 26, and 27, each paired with 2 question and answer pairs. For each question, we show its original *Programmatic Query* as well as the transformed *Datalog Query*. The object IDs are shown on the bounding boxes (in white) on the image.

Then the program is $\mathcal{P}_\theta^x = (KG_F \cup F_n \cup F_a \cup F_r, KG_R, J_n, \mathcal{Q})$. Note the universal knowledge graph KG_F is the same across different tasks.

Programmatic Query	[INITIAL, RELATE_REVERSE(left), HYPERNYM_FIND(vehicle), HYPERNYM_FIND(thing)]
Datalog Query	target(O) :– left(O, O′), name(O, vehicle), name(O, thing).
Answer	{1630226, 1630228}

Programmatic Query	[INITIAL, FIND_ATTR(parked), FIND_NAME(truck), RELATE_REVERSE(right)]
Datalog Query	target(O) :– attr(O, parked), name(O, truck), right(O, O′).
Answer	{3642007}

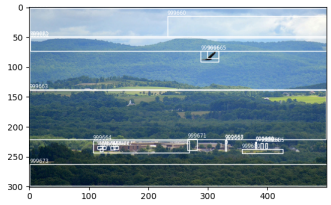
Figure 22: VQAR Example 1



Programmatic Query	[INITIAL, FIND_KG(can, hold flowers), RELATE_REVERSE(left), RELATE(left)]
Datalog Query	target(O) :– name(O, N), can(N, holdflowers), left(O, O_2), left(O_3, O).
Answer	{4458161, 4458148}

Programmatic Query	[INITIAL, RELATE_REVERSE(left), INITIAL, FIND_ATTR(blue), RELATE_REVERSE(right), OR]
Datalog Query	target(O) :– left(O, O′). target(O) :– attr(O, blue), right(O, O′).
Answer	{4458150, 4458153, 4383115, 4458156, 4383118, 4458159, 4458161, 4383122, 4458165}

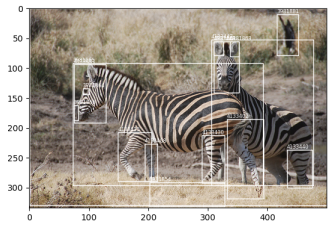
Figure 23: VQAR Example 2



Programmatic Query	[INITIAL, FIND_ATTR(cloudy), RELATE_REVERSE(in)]
Datalog Query	target(O) :– attr(O, cloudy), in(O, O′).
Answer	{999665, 999666, 999660}

Programmatic Query	[INITIAL, FIND_ATTR(black), INITIAL, FIND_KG(can be, opened or closed), AND]
Datalog Query	target(O) :– attr(O, black), name(O, N), can_be(N, opened or closed).
Answer	{999674, 999675, 999676, 999677, 999678}

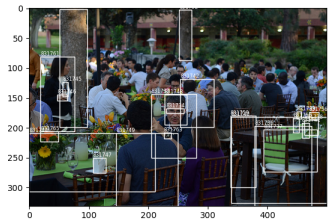
Figure 24: VQAR Example 3



Programmatic Query	[INITIAL, FIND_ATTR(grey)]
Datalog Query	target(O) :– attr(O, grey).
Answer	{3981862, 4133398, 3981863}

Programmatic Query	[INITIAL, HYPERNYM_FIND(odd-toed ungulate), HYPERNYM_FIND(herbivore)]
Datalog Query	target(O) :– name(O, odd-toed ungulate), name(O, herbivore).
Answer	{3981865, 4133447}

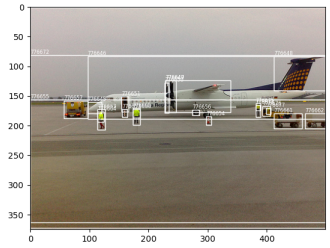
Figure 25: VQAR Example 4



Programmatic Query	[INITIAL, FIND_KG(can, hold water)]
Datalog Query	target(O) :– name(O, N), can(N, hold water).
Answer	{831745}

Programmatic Query	[INITIAL, FIND_NAME(bottle), INITIAL, RELATE(standing by), FIND_KG(can, grow branches), OR]
Datalog Query	target(O) :– name(O, bottle). target(O) :– standing_by(O′, O), name(O, N), can(N, grow branches).
Answer	{831745, 831764}

Figure 26: VQAR Example 5



Programmatic Query	[INITIAL, FIND_HYPERNYM(aircraft), FIND_ATTR(black), FIND_NAME(propeller)]
Datalog Query	target(O) :– name(O, aircraft), attr(O, black), name(O, propeller).
Answer	{776649}

Programmatic Query	[INITIAL, FIND_ATTR(neon), INITIAL, RELATE_REVERSE(by), OR]
Datalog Query	target(O) :– attr(O, neon). target(O) :– by(O, O′).
Answer	{776674, 776661, 776677, 776664, 776666, 776654}

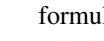
Figure 27: VQAR Example 6

G Framework Details

As noted in Section 6, the programming interface for Scallop is composed of a probabilistic relational database $(\mathcal{F}, \mathcal{J})$, and a set of Datalog rules $\mathcal{R}$. This Scallop framework is able to capture a variety of learning tasks, including but not limited to MNIST calculation and VQAR:

a. *Addition and sorting over MNIST digits*. $\mathcal{F}$ represents the output of the MNIST digit recognition network as tuples of the form `0.89::digit(`  `,3); 0.02::digit(``,4); ...` . $\mathcal{R}$ represents the logic rules for addition/sorting. For example, the rule for addition is `sum(imgA, imgB) :- digit(imgA, DA), digit(imgB, DB).`

b. *The VQAR task*. $\mathcal{F}$ represents the facts in the knowledge graph and the output of the three MLP classifiers, $\mathcal{M}_\theta = (\mathcal{M}^n_\theta, \mathcal{M}^a_\theta, \mathcal{M}^r_\theta)$, which predict names, attributes, and relations respectively. These predictions are transformed into probabilistic facts. For example, the $\mathcal{M}^n_\theta$ classifier takes in the bounding box and feature vector of the object `o1` and produces a distribution of the classified names: `0.81::name(o1, tiger); 0.15::name(o1, giraffe);` ... On the other hand, $\mathcal{M}^r_\theta$ takes in two bounding boxes and feature vectors from, say, object `ox` and `oy`. It produces a distribution of classified relations between `ox` and `oy`: `0.15::rela('on', ox, oy). 0.05::rela('behind', ox, oy).` ... $\mathcal{R}$ represents the rules in the knowledge base and the programmatic query.

c. *Formula parsing and evaluation* [23]. In this task, a vision model takes an image of a hand-written formula (e.g. ), and predicts the evaluation result. $\mathcal{F}$ encodes its output using probabilistic relations of the form `constant(`  `, 2)` and `binary_op(``, '+',`  `,`  `)`, $\mathcal{R}$ contains rules for formula evaluation, such as `eval(F, LY + RY) :- binary_op(F, '+', L, R), eval(L, LY), eval(R, RY).`

d. *Natural language reading comprehension* [4, 13]. In this task, a language model takes as input a natural language article (e.g. "Tom kicks the ball") and a natural language question (e.g. "Who kicks the ball?"), and Scallop generates the answer to the question. $\mathcal{F}$ encodes its output using probabilistic relations of the form `subject_verb(tom, kicks)`, `verb_object(kicks, ball)`, and `event(kicks, tom, ball)`. $\mathcal{R}$ represents the programmatic query `target(W) :- event(kicks, W, ball)`, which is obtained using a semantic parsing model.