Neurosymbolic Programming in Scallop: Design, Implementation, and Applications

Ziyang Li

Department of Computer and Information Science

University of Pennsylvania

November 16, 2024

Abstract

Neurosymbolic programming combines the otherwise complementary worlds of deep learning and symbolic reasoning. It thereby enables more accurate, interpretable, and domain-aware AI solutions that surpass purely neural or symbolic approaches. While significant advances have been made in domain-specific neurosymbolic methods, the field lacks a unified programming system for general neurosymbolic applications.

This dissertation proposes Scallop, a language for neurosymbolic programming. Scallop is relational and declarative, offering expressive reasoning capabilities such as recursion, negation, and aggregation. Scallop supports discrete, probabilistic, and differentiable modes of reasoning, allowing for seamless integration with diverse neurosymbolic pipelines. Scallop employs a provenance framework, which supports numerous reasoning back-ends that balance reasoning accuracy and scalability. Additionally, Scallop offers extensive tooling to integrate with PyTorch and a foreign interface for incorporating modern foundation models.

Beyond presenting the design and implementation of Scallop, this dissertation demonstrates its versatility through applications in the domains of computer vision (CV), natural language processing (NLP), security, program analysis, planning, clinical decision making, and bioinformatics. These applications span natural language reasoning, image and video scene graph generation, security vulnerability detection, cancer mortality prediction, and RNA secondary structure prediction. Through extensive empirical studies, we demonstrate that Scallop-based neurosymbolic solutions achieve superior accuracy, interpretability, and data efficiency.

Contents

Chapter 1

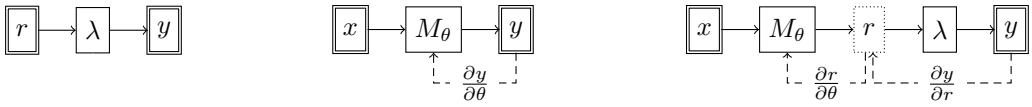
Introduction

1.1 Neurosymbolic Programming

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 procedurally, 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 λ shown in Figure 1.1a is interpretable but operates on limited, structured input r. On the other hand, a deep neural network such as M_θ shown in Figure 1.1b can operate on rich, unstructured input x but is not interpretable. Modern applications demand the capabilities of both paradigms. Examples include question answering [75], code completion [14], and mathematical problem solving [49], among many others. 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 [12]. It seeks to integrate symbolic knowledge and reasoning with neural architectures for better efficiency,



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

Figure 1.1: Comparison of different paradigms. Logic program λ 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.

interpretability, and generalizability than the neural or symbolic counterparts alone. Consider the task of hand-written formula evaluation [51], 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 $\langle x = \text{1+3÷5}, y = 1.6 \rangle$. A neurosymbolic program for this task, such as the one shown in Figure 1.1c, 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 λ to parse the sequence, evaluate the parsed formula, and output the final result 1.6.

Despite significant strides in individual neurosymbolic applications [103, 60, 15, 51, 63, 97], 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 1.1c, 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 λ that expresses 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 symbolic 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.

1.2 Scallop: What and Why

We have developed Scallop, a programming language that realizes all of 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.

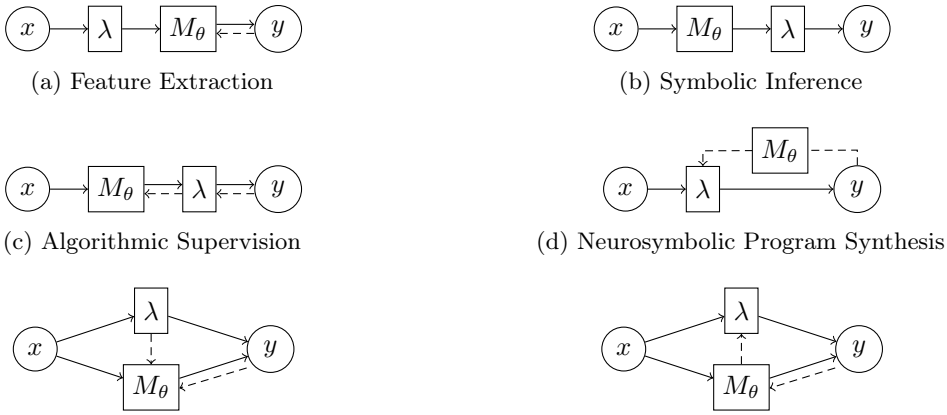
Our design choices were inspired by the following key observations. First, much of the world's data is stored in relational databases. Relations are also flexible enough to represent diverse kinds of data ranging from high-level visual and language features, to formal programs, to molecular structures. Second, a declarative language for symbolic reasoning allows computation to be expressed concisely via high-level rules, thereby alleviating programmer effort. Finally, the relational paradigm offers a suitable abstraction for advanced features needed for neurosymbolic programming, such as query planning, hardware (GPU) acceleration, and probabilistic and differentiable reasoning.

Our aim with Scallop is to provide a cohesive language and framework for integrating neural and symbolic components. In doing so, we seek to enable programmers to build neurosymbolic solutions that are more efficient, generalizable, and interpretable.

1.3 Building Blocks for Neurosymbolic Methods

A language that integrates neural and symbolic components can be applied to construct diverse and adaptable solutions. Broadly, a neurosymbolic solution to any given task involves the flexible interplay of neural and symbolic components, each serving distinct yet complementary roles in problem-solving. From the existing literature, several building blocks have emerged as crucial for effective neurosymbolic solutions, as depicted in Figure 1.2. We proceed to discuss each of these core building blocks in detail.

Feature Extraction The feature extraction process involves deriving symbolic features from an input x through a symbolic component, denoted here as λ, before passing these features to a neural model M_θ for training. Although feature extraction is an established practice in machine learning and typically not classified as neurosymbolic, it nevertheless exemplifies a functional integration of symbolic and neural elements. In this approach, learning is confined to the neural component, while the symbolic aspect serves to pre-process the input data.



(a) Feature Extraction (b) Symbolic Inference

(c) Algorithmic Supervision (d) Neurosymbolic Program Synthesis

(e) Neural Relaxation (f) Symbolic Distillation

Figure 1.2: Neurosymbolic compositions of neural component (M_θ) and symbolic component (λ), which serve as building-blocks for more complex neurosymbolic applications. We use solid arrows to denote forward data-flows, and dashed arrows to denote backward data-flows used to supervise the learning of the target component.

Notably, advanced feature extraction goes beyond simple tabular data and often incorporates sophisticated reasoning mechanisms to construct complex data structures. For instance, in program analysis, source code can be pre-processed into intricate structures such as abstract syntax trees (ASTs), data-flow graphs, symbolic constraints, or relational databases [23, 55, 106]. Neural networks may thus benefit from more comprehensive, structured information for downstream tasks, such as proposing bug fixes, detecting vulnerabilities, and analyzing type information even within binary code.

Symbolic Inference Symbolic inference involves performing posterior analysis on the outputs of a neural network M_θ using a symbolic component λ provided by a programmer. This analysis can serve various purposes, such as filtering nonsensical outputs, verifying output integrity, or combining multiple information sources symbolically to derive additional insights. Though straightforward in concept, an advanced symbolic inference component λ may handle probabilistic information, deriving a distribution rather than just the most likely output.

For instance, in the task of handwritten formula recognition $\langle x = \text{1+3÷5}, y = 1.6 \rangle$, after the neural network generates probability distributions for individual symbols, a probabilistic symbolic inference engine could synthesize a distribution over possible rational numbers. Another example is RNA secondary structure prediction, where a neural network predicts per-nucleotide structures, and a probabilistic RNA folding algorithm then parses this probabilistic sequence to generate the

top-k most likely structural parses. In Chapter 4, we cover many symbolic inference solutions where the M_θ are foundation models.

Algorithmic Supervision Algorithmic supervision extends symbolic inference by enabling the symbolic component λ to propagate learning signals to the neural network M_θ. As before, we assume that λ is provided by the programmer. While Figure 1.1 demonstrates one example of algorithmic supervision through differentiability in λ, it generally suffices for λ to propagate the learning signal. In this way, the symbolic "algorithm" λ serves as a guiding supervisor for the neural network M_θ.

Algorithmic supervision also functions as a form of weak supervision, as it does not require direct, fully supervised labels for M_θ; only the end label y is needed. This reduces the need for extensive data labeling or feature engineering, simplifying the training process. Numerous applications in Scallop leverage this approach, including the previously mentioned task of learning to evaluate handwritten formulas [51, 54]. This tutorial explores additional, advanced examples of algorithmic weak supervision in Chapter 6.

Neurosymbolic Program Synthesis Neurosymbolic program synthesis involves learning the symbolic program λ with the support of neural networks. This paradigm resembles the classical syntax-guided synthesis task [3], but replaces the traditional algorithmic synthesis procedure with a neural network M_θ. Here, the symbolic program λ is responsible for generating the expected outputs, and it may be iteratively refined to better align with a dataset.

This approach offers the advantage of interpretability, as the learned symbolic component is a white-box program that can be inspected and verified by humans [25]. Traditionally, synthesizing λ requires defining a limited domain-specific language [26] since general-purpose languages render synthesis computationally intractable. However, with the recent development of large language models (LLMs) capable of generating programs in general-purpose languages like Python, the synthesis of λ can now be achieved more efficiently [58].

Neural Relaxation Neural relaxation involves relaxing a deterministic and discrete symbolic reasoning component λ by replacing certain components in the pipeline with neural networks M_θ. This enables portions of previously symbolic components to be approximated by neural networks, improving adaptability to unseen scenarios.

For instance, consider the challenge of designing a neurosymbolic controller for drones: while effective deterministic controllers exist for standard maneuvers, they may struggle to adapt to out-of-domain scenarios, such as operating near the ground, in strong winds, or in proximity to other drones. By relaxing certain aspects of the controller into a neural network M_θ, the system gains greater flexibility and responsiveness in handling such scenarios, while being able to learn rapidly [68, 19].

Symbolic Distillation Symbolic distillation extracts information from a black-box neural network and converts it into a symbolic form λ. Although this process involves generating and refining λ, similar to neurosymbolic program synthesis, symbolic distillation focuses on translating otherwise uninterpretable weights from a well-trained neural network M_θ into an interpretable form.

This technique has been applied to scientific discovery in fields such as animal behavior analysis [90]. A symbolic program describing behaviors like "two mice running towards each other" can be distilled from a neural network trained on data of mice interactions. Another application is explanation synthesis for predicting cancer patient mortality [99]. For a model trained to predict 6-month mortality, symbolic distillation can generate explanations of specific predictions, providing clearer insights for clinical decision-making supported by machine learning systems.

Other Compositions In addition to the primary building blocks, there are other notable neurosymbolic compositions. For example, AlphaGo [84] is centered around a symbolic algorithm—Monte Carlo Tree Search—with neural networks for policy evaluation and move selection, creating a synergistic decision-making process. On the other hand, ChatGPT plugins [69] use a large language model as the primary system, which can invoke symbolic components like a Python interpreter, database retrieval, or web search to enhance functionality. As the field of neurosymbolic AI continues to evolve, we anticipate that more diverse and innovative compositions will emerge, broadening the scope and applications of neurosymbolic approaches.

1.4 Application Domains

In this section, we discuss the data modalities for which Scallop is best suited and explore the application domains where Scallop has shown effectiveness. We also identify the limitations of Scallop, highlighting tasks where it may be less effective.

Scallop can be broadly applied to applications that require both neural models and programmatic reasoning modules. It is particularly useful when the neural model requires additional training. With a fully differentiable, end-to-end neurosymbolic pipeline, strong supervision is not necessary for the neural model. Instead, *algorithmic supervision* can be used, offering benefits such as data efficiency and generalizability.

Data Modalities Scallop is capable of handling diverse data modalities by virtue of being based on the relational data model. The relational paradigm enables it to work seamlessly with existing relational databases and tabular data, encompassing information from knowledge bases, electronic health records, and internet documents. Additionally, natural language data from NLP tasks can be ingested in various forms: as raw sentences, embeddings (tensors), or structured representations such as relational databases or functional programs. Image data from computer vision can be converted into semantic representations like scene graphs. Videos, which extend images with a temporal dimension, can similarly be represented as spatio-temporal scene graphs for analysis in Scallop. Computer programs can be transformed into relational databases, capturing detailed information such as abstract syntax trees and control-flow graphs.

Application Domains We have applied Scallop across diverse domains, including natural language processing (NLP), computer vision (CV), planning, program and security analysis, bioinformatics, and healthcare. In the domain of NLP, we have applied Scallop to tasks that require reasoning, such as retrieving documents in a database, or analyzing data from sources such as electronic health records or legal documents. In the domain of computer vision, rather than focusing on low-level perception tasks like object segmentation and tracking, we have applied Scallop to hybrid tasks such as visual question answering and for supporting the training of scene graph generation models. In security analysis, we have applied Scallop to tasks like taint analysis, vulnerability detection, and fault localization. In bioinformatics, we have employed Scallop in applications such as predicting RNA secondary structures and RNA splicing. It is important to note that not all Scallop solutions follow a uniform architecture. We adapt different building blocks (Figure 1.2) depending upon each task's unique characteristics.

Applications Where Scallop May Be Less Effective We identify three examples where Scallop may not significantly enhance the task-solving process due to challenges in defining the

reasoning component or the appropriate intermediate representation.

1. *Generating Text with Subjective Criteria*. A common use-case of language models like GPT is generating text that satisfies subjective criteria in style or content, such as empathy or political neutrality. While language models can generate coherent paragraphs, identifying specific logical components for integration is challenging. The abstract nature of such tasks makes it difficult to pinpoint areas where logical reasoning would offer substantial value beyond what current language models provide.

2. *Basic Math Calculations* (e.g., $+$, $-$, $\times$, $\div$). This task is inherently symbolic and straightforward. Existing tools like Python or MATLAB can perform these operations directly, and there is no clear need for a perceptual model. The task is purely logical and lacks components that would benefit from Scallop's relational or perceptual capabilities.

3. *Low-Level Motor Control for Robots*. Scallop's syntax is more suited to defining high-level discrete logical rules rather than handling low-level numerical processing of sensory signals. Thus, for tasks like motor control based on raw sensor inputs, imperative languages such as C or Python may be more effective for specifying the numerical algorithms.

1.5 Contributions

In summary, this dissertation makes the following contributions:

Core Scallop Language Design and Implementations

1. I present the design of the core Scallop language.
2. I present the compiler from the Scallop language to a low-level language, SclRam.
3. I present a provenance framework for SclRam that supports tagged computation.
4. I present the formal semantics of SclRam with the support of provenance framework.
5. I present practically useful provenances for discrete, probabilistic, and differentiable reasoning.
6. I present the top-k-proofs provenance for scalable approximated probabilistic reasoning.

7. I present a foreign interface for Scallop language that allows for tight integration of external modules and databases.

8. I present a general relational interface for foundation models.

9. I present a plugin library for Scallop with plugins for foundation models such as large language models and vision language models.

Scallop Application and Evaluation

1. I present the application of Scallop to synthetic neurosymbolic reasoning tasks including MNIST-R, Hand-Written Formula, and PathFinder.

2. I present a new synthetic planning benchmark, PacMan-Maze, and demonstrated the neurosymbolic application Scallop on the benchmark.

3. I present two neurosymbolic solutions involving Scallop on a natural language reasoning task CLUTRR that involves kinship reasoning.

4. I present two neurosymbolic solutions involving Scallop on two visual question answering benchmarks: CLEVR and VQAR.

5. I present the applications involving Scallop and foundation models on a variety of natural language and multi-modal benchmarks including GSM8K, Date Reasoning, Tracking Shuffled Objects, Amazon Product Search, among others.

6. **I propose** the neurosymbolic application of Scallop on video scene graph generation.

7. **I propose** the application of Scallop on real-life whole-project vulnerability detection, a preliminary study of a neurosymbolic solution for the task. This involves the construction of a novel vulnerability dataset, CWE-Bench-Java.

8. **I propose** the application of Scallop on RNA secondary structure prediction, and a preliminary study of our neurosymbolic solution on the ArchiveII dataset.

Chapter 2

Basics of Programming in Scallop

In this chapter, we present Scallop as a relational logic programming language. It is a Datalog-based language extended with features such as negation, aggregation, disjunctive heads, algebraic data types, foreign functions, and foreign predicates. We provide a comprehensive overview of the core language encompassing all of these constructs.

2.1 Relations, Data Types, and Facts

The fundamental data type in Scallop is a relation which comprises a set of 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`). A comprehensive list is provided in Table 2.1. For example, Listing 2.1 declares two binary relations, `mother` and `father`. Note that we declare multiple relations with one `type` keyword. Values of relations can be specified via individual tuples or a set of tuples of constant literals, as shown in line 5 and line 8 in Listing 2.1. The type of facts must conform to the statically declared relation type. All the tuples under `mother` and `father` are of arity 2 and both elements are strings. Note that the keyword `rel` is chosen as a shorthand for `relation`, which is used to define relations.

As a shorthand, primitive values can be named and declared as constant variables, as shown in line 2 in Listing 2.2. 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

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

// Christine is Bob's mother
rel mother("Christine", "Bob")

// Bob is father of two kids, Alice and John
rel father = {("Bob", "Alice"), ("Bob", "John")}
```

Listing 2.1: Basic relation and fact definitions representing a family.

```
// Relationships declared as constants
const FATHER = 0, MOTHER = 1, GRANDMOTHER = 2, ...

// father's mother is grandmother
rel composition(FATHER, MOTHER, GRANDMOTHER)
// mother's brother is uncle
rel composition(MOTHER, BROTHER, UNCLE)

// A family kinship graph
rel kinship = {
  ("Christine",MOTHER,"Bob"), // Bob's mother is Christine
  ("Bob",FATHER,"Alice"),     // Alice's father is Bob
  ("Bob",FATHER,"John"),      // John's father is also Bob
}
```

Listing 2.2: An alternative way to declare kinship relations. Here, kinship relations are abstracted into constant integers. We use the relation `composition` to represent higher-order kinship rules.

constant unsigned integers is `usize`. Similarly, the type of the `kinship` relation will be inferred as `(String, usize, String)`. We note that this new representation of family graph is equivalent to the one defined in Listing 2.1, albeit just using one relation (`kinship`) instead of two (`father` and `mother`).

2.1.1 Nullary, Unary, and Binary Relations

Nullary or Boolean Relations Many things can be represented as relations. We start with the most basic programming construct, boolean. While Scallop allows values to have the boolean type, relations themselves can encode boolean values. The example shown in Listing 2.3 contains an arity-0 relation named `is_target`. There is only one possible tuple that could form a fact in this relation, that is the empty tuple `()`. Consider the relation `is_target` as a set. If the set contains no element (i.e., empty), then it encodes boolean "false"; otherwise, the set could contain at most and exactly one tuple, and the relation encodes the boolean "true".

Type	Primitive Types in Scallop
Unsigned Integers	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`
Signed Integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`
Floating Points	`f32`, `f64`
Character	`char`
Boolean	`bool`
String	`String`
Time	`Duration`, `DateTime`

Table 2.1: The list of primitive types in Scallop along with their descriptions.

```
// Declaration of the type of a 0-arity relation
type is_target()

rel is_target()        // Fact declaration
rel is_target = {()}   // Set containing an empty tuple
```

Listing 2.3: Declaration of type and fact for a 0-arity (or boolean) relation.

Unary Relations Unary relations are relations of arity 1. We can define unary relations for "variables" as we see in other programming languages. Listing 2.4 declares a relation named greeting containing one single string of "hello world!". It shows three ways of declaring a single fact in the relation. The first two were introduced earlier but the third one omits the parenthesis since the relation is unary.

Binary Relations As the name suggests, binary relations are relations of arity 2. We demonstrate binary relations using a graph (Figure 2.1) and its Scallop representation (Listing 2.5). As shown in the code, we define an enum type named `Node` containing three variants, `A`, `B`, and `C`, corresponding to the three nodes in the graph. The unary relation `node` is thus a set containing the three nodes, and the `edge` relation is a binary relation containing directed edges in the graph.

2.1.2 Type Inference

Scallop supports type inference, meaning that not all types need to be explicitly annotated. In Scallop, types are inferred during the compilation process. When taking the code shown in Listing 2.5, Scallop is capable of inferring that `node` relation is of type `(Node,)`, while the `edge` relation is of type `(Node, Node)`. Type inference will fail if conflicts are detected. For instance, the Listing 2.6 shows one piece of Scallop code which results in an error message during compilation. This is due to that both a value of type `Node` and one of `String` are observed as the second element of the `edge` relation.

```
1 rel greeting("hello world!")
2 // or
3 rel greeting = {("hello world!",)}
4 // or
5 rel greeting = {"hello world!"}
```

Listing 2.4: Declaration of a unary relation `greeting`.

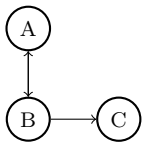


Figure 2.1: A sample graph with three nodes.

```
1 // An enum type Node
2 type Node = A | B | C
3
4 // The relations replicating the graph
5 rel node = {A, B, C}
6 rel edge = {(A, B), (B, A), (B, C)}
```

Listing 2.5: The relations and facts representing the graph shown in Figure 2.1.

2.2 Logic 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 `=`. If the body "holds", then we derive the atom of the head. Listing 2.7 shows three rules defining the `grandmother` relation. We say that the body of a rule can be grounded if every single variable can be substituted by values in existing facts in the database. For instance, the body of the rule on line 6 in Listing 2.7 can be grounded by two facts, `father("Bob", "Alice")` and `mother("Christine", "Bob")`. The variable `c` can be grounded with "Christine", `b` can be grounded with "Bob", while `a` can be grounded with "Alice". Notably, the variable `b` appears in both the `mother(c, b)` atom as well as the `father(b, a)` atom, meaning that the value being used to ground the variable `b` has to appear in both facts.

In a rule, conjunction is specified using `and`-separated atoms within the rule body whereas disjunction can be 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. Conjunctions and disjunctions can also be expressed using logical connectives like `and`, `or`, and `implies`. For instance, the last rule (line 9-10 of Listing 2.7) is equivalent to the two rules above combined.

Scallop performs a few compilation checks to ensure that the program is well-formed. First of all, the rules need to type check. In the case of Listing 2.7, all the shown relations are binary `String` relations, and therefore type inference succeeds. Moreover, all the variables appearing in the head atom must be *bounded* by atoms in the body. Consider the first rule (line 2) as an

```
> rel edge = {(A, B), (B, "1")}

[Error] cannot unify types `Node` and `String`, where
the first is declared here
  REPL:0 | rel edge = {(A, B), (B, "1")}
         |                  ^
and the second is declared here
  REPL:0 | rel edge = {(A, B), (B, "1")}
         |                        ^^^
```

Listing 2.6: A piece of Scallop code that has a conflict detected by type inference. We also show the error message thrown when compiling the code.

```
1  // A few facts under the base relations
2  rel father = {("Bob", "Alice"), ("John", "Harry")}
3  rel mother = {("Christine", "Bob")}
4
5  // Father's mother is grandmother
6  rel grandmother(c, a) = mother(c, b) and father(b, a)
7  // Mother's mother is also grandmother
8  rel grandmother(c, a) = mother(c, b) and mother(b, a)
9
10 // == is equivalent to... ==
11
12 // Mother or father's mother is grandmother
13 rel grandmother(c, a) = mother(c, b) and
14                         (mother(b, a) or father(b, a))
```

Listing 2.7: A set of logic rules computing the `grandmother` relation from `father` and `mother` relations. Given the facts declared at the top, we can derive the fact `grandmother("Christine", "Alice")`, which means that "Christine is the grandmother of Alice."

example, in which variable `a` is bounded by the `father` relation, while variable `c` is bounded by `mother`. Therefore, the head atom of the rule is bounded and well-formed. For the last rule (line 9-10) where the body contains disjunctions, head variables need to be bounded for all branches in the body. This is indeed true since `a` is bounded by both atoms in the disjunction.

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. Listing 2.8 shows a few examples. Specifically, the first shows that floating point weight and height can be used to compute body mass index (BMI). In the second example, strings are concatenated together using FF, producing the result `full_name("John Doe")`.

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 last example shown in Listing 2.8

```
1  // E1: Computing body mass index (BMI) by arithmetic
2  type person(name: String, weight_kg: f32, height_m: f32)
3  rel bmi(name, w / (h * h)) = person(name, w, h)
4
5  // E2: Computing ful name by concatenating strings
6  rel first_name("John"), last_name("Doe")
7  rel full_name($string_concat(x, " ", y)) =
8      first_name(x) and last_name(y)
9
10 // E3: Potentially failing
11 rel denominator = {0, 1, 2}        // three denominators
12 rel result(6 / x) = denominator(x) // results = {3, 6}
```

Listing 2.8: A set of logic rules that make use of the foreign functions in Scallop.

```
1  // Type declaration of edge relation
2  type edge(x: Node, y: Node)
3
4  // Transitive closure computing path
5  rel path(x, y) = edge(x, y)
6  rel path(x, z) = path(x, y) and edge(y, z)
```

Listing 2.9: The `edge-path` program defining a transitive closure that computes `path`s given a set of edges.

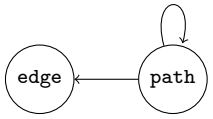


Figure 2.2: The dependency graph associated with the `edge-path` program.

(line 10-12), when dividing 6 by `denominator`, the result is not computed for denominator `0` since it causes a FF failure. The purpose of this semantics is to support probabilistic extensions rather than silent suppression of runtime errors. When dealing with floating-point numbers, tuples with `NaN` (not-a-number) are also discarded.

2.3 Recursion, Negation, and Aggregation

In this section we discuss some slightly advanced features of logic rules in Scallop, namely recursion, negation, and aggregation. These features are key to an expressive language for Scallop and making it applicable to a diverse set of applications.

2.3.1 Recursion

A powerful programming construct in Scallop is to declaratively define recursion. Within a rule, if a relational predicate appearing in the head appears in the body, the rule is recursive. More generally, 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. For instance, Listing 2.9 shows a program with recursion. In the program, `path` depends on `edge` (line 5-6) and `path` itself

```
1 type fib(bound x: i32, y: i32)    // type definition
2 rel fib = {(0, 1), (1, 1)}        // base cases
3 rel fib(x, y1 + y2) =             // recursive case
4     fib(x - 1, y1) and fib(x - 2, y2) and x > 1
5 query fib(5, y)                   // result: fib(5, 8)
```

Listing 2.10: Definition of Fibonacci number in Scallop. We note that `fib` is by definition an infinite relation. To make computations feasible, we add the `bound` keyword on the first line, which we delay the discussion till Section 2.5.

(line 6). Based on this information, we can draw a dependency graph for the program, shown in Figure 2.2. Since there is a self-loop on the `path` relation, we say that the program is recursive.

Recursion is also very useful in recursive mathematical definitions. For example, the definition of Fibonacci numbers is recursive. Recall the formal definition of Fibonacci numbers:

$$\text{fib}(x) = \begin{cases} \text{fib}(x-1) + \text{fib}(x-2) & \text{if } x > 1, \\ 1 & \text{otherwise} \end{cases}$$

In Scallop, we encode the function fib as a binary relation between the integer input and output, shown in Listing 2.10. On line 2, we define the base cases for fib(0) and fib(1). In terms of the recursive case, we obtain the results of `y1` $=$ fib$(x-1)$ and `y2` $=$ fib$(x-2)$ and compute the sum `y1 + y2`. This almost literally translates to the recursive rule on line 4. We note that an extra constraint `x > 1` must be added in order for the computation to terminate. At the end, when the atom `fib(5, y)` is queried, Scallop will return that a fact `fib(5, 8)` suggesting that 8 is the result of computing fib(5).

2.3.2 Negation

Scallop supports stratified negation using the `not` operator on atoms in the rule body. Listing 2.11 shows a rule defining the `has_no_children` relation as any person `p` who is neither a father nor a mother (line 7-8). In the rule, the underscore (`_`) stands for *wildcard* which is used to match any value. Note that we need to bound `p` by a positive atom `person` in order for the rule to be well-formed. In the rule that does not compile, the variable `p` can be anything other than `"Bob"` or `"Christine"`, meaning that it is impossible to enumerate the values. Scallop rejects these kinds of programs by ensuring that all variables that occur in the head are bounded by positive atoms in the body. At the end, we have the relation `has_no_children` containing one single tuple `("Alice")`,

```
1  // A family containing three people
2  rel person = {"Alice", "Bob", "Christine"}
3  rel father("Bob", "Alice")      // Bob is Alice's father
4  rel mother("Christine", "Bob") // Christine is Bob's mother
5
6  // Compute the person who has no children
7  rel has_no_children(p) = person(p) and
8      not father(p, _) and not mother(p, _)
9
10 // !! This rule does not compile: p is not bounded !!
11 rel error(p) = not father(p, _) and not mother(p, _)
```

Listing 2.11: A Scallop program that computes the person who has no children given the kinship relations within a family. Note that we also show one rule (line 11) which cannot compile due to the existance of an unbounded variable `p`.

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

Listing 2.12: A rule that has negative circular dependency, causing the compiler to reject the program.

since according to the facts defined above, `"Alice"` is not a parent of anyone.

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 example shown in Listing 2.11, `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 rule shown in Listing 2.12 is rejected by the compiler, as the negation is not stratified.

2.3.3 Aggregation

Scallop also supports stratified aggregation. We use the assignment symbol `:=` to retrieve the results obtained from aggregations. 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 construct specifies the binding variables, the aggregation body to bound those variables, and the result variable(s) to assign the result. The rule with aggregation in Listing 2.13 reads, "variable `n` is assigned the count of `p`, such that `p` is a person". Specifically, `n` is the result of the aggregation, `count` is the aggregator, `p` is the qualified variable for aggregation, and `person(p)` is the body of the aggregation. At the end, `num_people(3)` is derived since there are 3 facts in the `person` relation. 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. On line 7 we also

```
1  rel person = {"Alice", "Bob", "Christine"}
2
3  // count the number of people, which should be 3
4  rel num_people(n) = n := count(p: person(p))
5
6  // a syntax sugar that is equivalent to the above rule
7  rel num_people = count(p: person(p))
```

Listing 2.13: A simple rule with aggregation counting the number of people.

```
1  // Bob is a parent of Alice, Christine is a parent of Bob
2  rel person = {"Alice", "Bob", "Christine"}
3  rel parent = {("Bob", "Alice"), ("Christine", "Bob")}
4
5  // Implicit group-by:
6  // >> result: {("Bob", 1), ("Christine", 1)}
7  rel num_child(p, n) = n := count(c: parent(p, c))
8
9  // Explicit group-by:
10 // >> result: {("Alice", 0), ("Bob", 1), ("Christine", 1)}
11 rel num_child(p, n) = n := count(c: parent(p, c)
12                                  where p: person(p))
```

Listing 2.14: A few examples with group-by aggregation. Notice that the resulting fact `("Alice", 0)` is not derived by the rule with implicit group-by operation.

show a syntax sugar when the result of the aggregation directly corresponds to the tuples to be stored in the head relation.

Further, Scallop supports SQL-style group-by operations. If a variable is bounded in the aggregation body and is also used in the head of the rule, we say that variable is a group-by variable. In Listing 2.14, we compute the number of children of each person `p`, which serves as the group-by variable. However, depending on whether we explicitly bound the group-by variable `p`, we get different results. On line 11, we explicitly use a `where` clause to bound the variable `p` with everyone in the `person` relation. As such, we would also find the number of children of `"Alice"`, which is 0. For the rule on line 7, on the other hand, we do not explicitly bound the group-by variable `p`, meaning that no information is present other than the `parent` relation. Since `"Alice"` is not a parent of anyone, the entry `("Alice", 0)` will not exist in the result.

Finally, quantifier aggregators such as `forall` and `exists` return one boolean variable. For instance, for the aggregation shown in Listing 2.15, 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". At the end, we would obtain a fact `integrity_constraint(true)`, meaning that the constraint is satisfied given the kinship facts shown on line 1-2.

There are a couple of syntactic checks on aggregations. First, similar to negation, aggregation

```
rel father("Bob", "Alice")   // Bob is Alice's father
rel daughter("Alice", "Bob") // Alice is Bob's daughter

// An integrity constraint for kinship graphs
rel integrity_constraint(sat) = sat := forall(a, b:
        father(a, b) implies (son(b, a) or daughter(b, a)))
```

Listing 2.15: A rule encoding an integrity constraint about kinship graphs, making use of the `forall` and `implies` operators.

```
// An independent probabilistic fact
rel 0.95::kinship(FATHER, A, B)

// A mutually exclusive set of probabilistic facts
rel kinship = {
    0.95::(FATHER, A, B); // A is B's father with 0.95 prob
    0.01::(MOTHER, A, B); // A is B's mother with 0.01 prob
    ...
}
```

Listing 2.16: Probabilistic facts within the `kinship` relation written in Scallop in two different ways. Note that in the second example, facts are separated by semicolons (`;`), meaning that the facts are mutually exclusive.

also needs to be stratified—a relation cannot be dependent on itself through an aggregation. Second, the binding variables must be bounded by at least one positive atom in the body of the aggregation. Lastly, the body of the rule and the body of an aggregation form nested scopes. A variable in the inner scope is shadowed if the variable is *redefined by an aggregation* in the outer scope.

2.4 Programming with Probabilities

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 shown in Listing 2.16, 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

```
rel top_1_kinship(r,a,b) = r := top<1>(rp: kinship(rp,a,b))
// result: { 0.95::top_1_kinship(FATHER, A, B) }
```

Listing 2.17: A Scallop rule using the `top` sampler. Following Listing 2.16, for each pair of people `a` and `b`, we find the top 1 kinship relation between them.

```
// Grandmother's daughter is 90% likely one's mother
// Note: she could also be one's aunt
rel 0.9::mother(a,c) = grandmother(a,b) and daughter(b,c)

// == the above rule is desugared to... ==
rel 0.9::prob_of_rule() // one auxilliary nullary relation
rel mother(a,c) = grandmother(a,b) and daughter(b,c) and
                  prob_of_rule()
```

Listing 2.18: A probabilistic rule where the probability is encoded in the head.

same surface syntax as aggregations, allowing sampling with group-by. The following rule shown in Listing 2.17 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. 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 rule shown in Listing 2.18 states that a grandmother's daughter is one's mother with 90% confidence. Probabilistic rules are syntactic sugar. They are implemented by introducing in the rule's body an auxiliary nullary (i.e., boolean) fact that is regarded true with the tagged probability.

2.5 On-Demand Computations

In normal Scallop, facts are computed in a bottom-up fashion. That is, for each rule, we start from grounding the body with existing facts, and derive the fact in the head. Typically, this would derive all possible outcomes for a relation, which may be costly. Worse, it may even be impossible to derive fully due to the derived relation being infinite. One example is the computation of Fibonacci number (also shown previously in Listing 2.10). Fibonacci number itself is infinite, so given the base cases for 0 and 1, it is expected that the computation for all Fibonacci numbers will never terminate. Such Scallop program is shown in Listing 2.19. However, often times we have a specific query for these inifinite relations. As shown on line 6 in Listing 2.20, we are querying for the

```
1 type fib(x: i32, y: i32)
2 rel fib = {(0, 1), (1, 1)}
3 rel fib(x, y1 + y2) = fib(x - 1, y1) and fib(x - 2, y2)
4 // NOTE: will not terminate...
```

Listing 2.19: First implementation of Fibonacci number, which would result in a non-terminating execution due to the `fib` being an infinite relation.

```
1 // adding adornment to define on-demand pattern
2 type fib(bound x: i32, y: i32)
3 rel fib = {(0, 1), (1, 1)}
4 rel fib(x, y1 + y2) = fib(x - 1, y1) and fib(x - 2, y2) and
5         x > 1 // avoid generating infinite demand
6 query fib(5, y)
```

Listing 2.20: Another implementation of Fibonacci number, which utilizes *on-demand computation* by adding the `bound` keyword on `x` when defining the `fib` relation. In the rule on line 4-5, we also include a constraint `x > 1` in order to bound the recursive generation of demand.

5th Fibonacci number, and nothing else is expected. For such cases, we might use *on-demand computation* to answer those queries, without computing the full infinite relation. Specifically, the number `5` is the *demand* for the `fib` relation.

We achieve on-demand computation in Scallop by doing the following (Listing 2.20). First, as shown on line 2, we add a `bound` keyword to the `x` variable when defining the `fib` relation. This is called an *adornment*, meaning that everytime the relation `fib` is computed, we are treating the first argument `x` as the input. For the second variable `y` that is not adorned by the `bound` keyword, it means that the value will be derived by rules. A variable not adorned by `bound` is treated a *free* variable. We say that `bound-free` (or *bf* in short) is the on-demand pattern for the `fib` relation. Without specification, normal relations have an *all-free* on-demand pattern, which means they are *not* on-demand relations.

For the rules with on-demand relations as the head atom, the well-formedness is slightly different than the regular rules. Specifically, not only do variables in positive body atoms considered bounded, but also the variables bounded by the on-demand head atom.

On-demand relations can be used to optimize execution of queries. Consider the `edge-path` example shown in Listing 2.21. Suppose we have a dense graph with thousands of edges, the normal transitive closure defined for `path` would enumerate all possible paths in the graph. However, given that we have a query on line 9 that desires to find all sources that can reach a particular sink `S`, there is no need to enumerate all the paths. The desirable demand pattern for this query would be *fb*, meaning that we want to set the second argument of the `path` as a `bound` variable (line 3).

```
// Type defs; path is declared with on-demand pattern "fb"
type Node = usize
type edge(x: Node, y: Node), path(x: Node, bound y: Node)

// Suppose we have a dense graph with thousands of edges
rel edge = { /* (0, 1), lots of tuples..., (T, S) */ }

rel path(x, y) = edge(x, y) or (edge(x, z) and path(z, y))
query path(x, S) // query a path with a sink at node S
```

Listing 2.21: The `edge-path` program with on-demand `path` relation.

$$\text{(Expr)} \quad e ::= i \mid e_1 + e_2 \mid e_1 - e_2$$

Figure 2.3: A simple language for integer arithmetic expressions. An expression can be either a simple integer i, an addition of two expressions, or a subtraction of two expressions.

```
type Expr = Int(i32)         // a simple integer
          | Add(Expr, Expr) // adding two expressions
          | Sub(Expr, Expr) // subtracting two expressions

// an expression representing 1 + (3 - 2)
const MY_EXPR: Expr = Add(Int(1), Sub(Int(3), Int(2)))

// a unary relation storing expressions
type target_expr(e: Expr)
rel target_expr = { MY_EXPR }
```

Listing 2.22: A custom algebraic data type defined in Scallop that represents the small language shown in Figure 2.3 (line 1-3). Line 4 shows one expression $1 + (3 - 2)$ expressed using `Expr` type.

With this adornment, Scallop will only compute the paths that reaches `S`, avoiding the expensive exploration of all possible paths.

2.6 Algebraic Data Types

Algebraic data types (ADTs) are powerful programming constructs that allows user to define custom data structures and enum variants. They can be used to define recursive data structures such as lists and trees. Domain-specific languages (DSLs) can also be represented using ADTs. For instance, Figure 2.3 and Listing 2.22 shows one simple integer arithmetic language expressed in Scallop as a custom ADT. We use the `type` keyword to start the declaration, and the bar (`|`) symbol to separate each ADT variants. There are three variants here, among which the `Add` and `Sub` variants are considered *recursive* because their arguments contain the `Expr` type itself. On the other hand, the `Int` variant is a *terminal*. We show one *Entity* of the custom `Expr` type declared as a constant on line 6 of Listing 2.22.

```
1  // eval relation evaluates the expr and yields int result
2  type eval(bound expr: Expr, result: i32)
3
4  // three rules handling the variants of Expr
5  rel eval(Int(i), i)
6  rel eval(Add(e1,e2), i1+i2) = eval(e1, i1) and eval(e2, i2)
7  rel eval(Sub(e1,e2), i1-i2) = eval(e1, i1) and eval(e2, i2)
8
9  // query the result of MY_EXPR
10 query eval(MY_EXPR, y)
```

Listing 2.23: A Scallop program that evaluates `Expr`.

Values of custom ADTs can be used just like any other values in Scallop. Line 9 of Listing 2.22 declares one unary relation storing such expressions, whereas line 10 shows a fact of that relation containing the constant `MY_EXPR`. We next showcase how entities can be read and created dynamically within Scallop rules.

Entities can be *destructed* by pattern matching expressions. For instance, Listing 2.23 shows three rules, each handling a certain variant of the `Expr` ADT. The first "rule" reads "evaluating the expression `Int(i)` yields an integer `i`". Although it looks like a fact, there is an unbounded variable `i` so it will be desugared and treated as a rule. The second and third rule matches on the `Add` and `Sub` variants. They recursively evaluate the sub-expressions `e1` and `e2`, and then adds or subtracts the respective results to form the final result.

The relations handling ADT entities can also be adorned by `bound` keywords to indicate on-demand computation patterns. For instance, on line 2 of Listing 2.23, we let `eval` take in expressions and yield integer results. If the pattern is not specified, Scallop will evaluate every single declared expression. However, now that we have a demand specified on line 10 (`MY_EXPR`), Scallop will only evaluate necessary expressions in order to compute the result for `MY_EXPR`, yielding the resulting fact `eval(MY_EXPR, 2)`.

2.7 Foreign Interface

Scallop supports a foreign interface which allows external definition of functions, predicates, and attributes. These constructs allow Scallop to be effective in diverse applications, including a tight integration of foundation models, which we describe in detail in Chapter 4. In this section we describe such constructs and a selection of standard library containing interfaced items. We note that the code snippets in this section may show the use of `extern` keyword, suggesting the

```
// A simple function that retrieves the day component given
// a DateTime. A "day" is a 32-bit unsigned integer (u32)
// representing the day within a month, starting from 1.
extern type $day(d: DateTime) -> u32

// Absolute value function that is generic w.r.t. a number
// type T. It takes in a value of T and returns a value
// of type T.
extern type $abs<T: Number>(x: T) -> T

// Taking the substring of a given string with a integer
// range. Note that the end index `e` is optional; if not
// provided, we retrieve the part of string after the begin
// index `b`. Otherwise, we take the substring from b to e.
extern type $substring(s: String, b: usize, e: usize?)
              -> String

// Take in an arbitrary amount of strings and concatenate
// them into the result string. Note that the strs argument
// is a vararg, denoted by the "..." symbol.
extern type $string_concat(strs: String...) -> String

// Format a string using other values.
extern type $format(form: String, args: Any...) -> String
```

Listing 2.24: Example type declarations of foreign functions included in Scallop's standard library.

declaration of externally defined items. However, during normal use of Scallop, such declarations are not neccessary and most foreign constructs are imported automatically.

2.7.1 Foreign Functions

In Scallop, foreign functions are pure functions that accept basic values and returns a single basic value upon success. We have showcased simple arithmetic operations and foreign function calls in prior examples (e.g. Listing 2.8), and we will take closer look in this section. In the most simplistic form, foreign functions are defined to be `$FUNC(ARG_TYPE, ...) -> RET_TYPE`. The function starts with a dollar sign `$`, and may take in multiple arguments with declared argument types (`ARG_TYPE`). The function, upon success, must return one value of the return type. However, Scallop's foreign function interface allows advance features such as (a) generic functions with type parameters, (b) functions with optional argument, and (c) functions with variable argument (vararg). Some examples using these features are shown in Listing 2.24. We now elaborate on each of these features.

Generic Functions When defining the type of a function, we may use an additional angle brackets `<...>` after the function name, to specify the generic type parameters. Each type parameter may

be followed by a type family to give additional constraint on the type. For instance, the `$abs` function shown in Listing 2.24 is a generic function with one type parameter, `T`, that needs to be a `Number`. Signed or unsigned integers as well as floating point numbers are types under the family `Number`. The absolute value function is properly defined on any of such data types.

There are a fixed set of type families, which are `Any`, `Number`, `Integer`, and `Float`. As a syntax sugar, if the type family is not specified on a type parameter, we default its family to `Any`, allowing value of any type to be passed into the function.

When using a generic function, it is not neccessary to explicitly instantiate the function with a concrete type, as the type inference module of Scallop will find the most suitable type automatically. For instance, without special configuration, the expression `$abs(-3)` in Scallop will return the number `3` of type `i32`, as the literal number `-3` has the type `i32` by default.

Optional Argument When specifying the type of an argument to the function, we may add a question mark (`?`) at the end to denote that the argument is optional. Optional arguments must occur after non-optional arguments. For instance, the `$substring` function shown in Listing 2.24 is a function with argument `e` being optional. This means that we may call the function in two different ways: `$substring("hello", 3)` returns `"lo"` while `$substring("hello", 3, 4)` returns `"l"`.

Variable Argument There are functions that may accept an arbitrary amount of arguments. We may specify the property, vararg, by adding the ellipses (`...`) after the type of that argument. Note that the variable argument, similar to optional argument, must appear after non-vararg arguments. A foreign function may have at most one variable argument. The `$string_concat` function shown in Listing 2.24 is an example that can take in an arbitrary amount of strings and performs the concatenation. For example, `$string_concat("a", "b")` returns `"ab"` and `$string_concat("a", "b", "c")` returns `"abc"`.

Note that when specifying variable arguments, the argument that may have the arbitrary amount must be of the same type or type family. If we want arbitrary arguments, we may use the type family `Any`. For instance, the `$format` function accepts one format string and an arbitrary amount of arbitrary values. When invoked with `$format("1 + 1 = {}", 1 + 1)`, the second argument is an integer (`i32`), and the returned value will be `"1 + 1 = 2"`. But when invoked with `$format("{} > 0?  {}", 1, 1 > 0)`, the second argument is integer while the third argument is a boolean, and the returned value will be `"1 > 0?  true"`.

```
// Given a string, produce a set of (index, char) pairs
extern type string_chars(bound s:String, i:usize, c:char)

// Say that we have an RNA sequence
rel rna = {"GGCCCUUUUCAGGGCC"}

// We want to obtain the nucleotide at each position i,
// using the foreign predicate string_chars
rel nucleotide(i, n) = rna(s) and string_chars(s, i, n)
// result:
//   neucleotide(0, 'G'),
//   neucleotide(1, 'G'),
//   neucleotide(2, 'C'), ...
```

Listing 2.25: An example foreign predicate `string_chars`.

Error Handling Foreign functions may fail. When they fail, there is no value being returned and the computation for this given input will be discarded. For instance, implicit foreign function such as division might fail due to divide-by-zero, and explicit foreign function such as `$substring` might fail if the given indices are out-of-bounds of the given string. By default, no error message will be thrown and errors are silently suppressed. This is beneficial because, in a relational and declarative language where inputs can be probabilistic, a significant amount of redundant computation might occur, and external functions might be invoked on invalid inputs. Nevertheless, Scallop provides compiler and runtime nobs to allow the report of errors.

2.7.2 Foreign Predicates

Foreign predicate is a generalized interface of foreign function, which can now produce multiple outputs associated with additional informations such as probabilities. Predicates are mostly declared just like other relations in Scallop, where inputs should be associated with `bound` keywords while outputs may be associated with `free` keywords. Here, we add the `extern` keyword to denote that the `PREDICATE` should be defined externally.

```
extern type PREDICATE(bound IN: TYPE, ..., OUT: TYPE)
```

Conceptually, foreign predicate "relates" the inputs and the outputs. This means that given a specific input to the predicate, multiple facts involving the input and outputs may be produced by the predicate.

In Listing 2.25 we showcase one foreign predicate `string_chars`, that could help in obtaining the nucleotides ($\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$) in an RNA sequence string. Taking the string `s` as an input, `string_chars` produces `(s, i, n)` triplets where `i` is the index of a character in the string, and `n` is the character

```
// Given two floating point numbers, compute the
// probability of which the two numbers are equal.
extern type soft_eq(bound x: f32, bound y: f32)

// Compute the output probability
rel output() = soft_eq(0.9, 1.0) // 0.998::output()
```

Listing 2.26: The usage of an example foreign predicate `soft_eq` which may return probabilities associated with the output.

itself. It is clear that `string_chars` returns multiple facts as the output, whereas foreign functions introduced in the previous section can only return one output.

Foreign Predicates that Produce Probabilities Foreign predicates produce facts which can be associated with additional tags. The most common use case of this feature is the encoding of probabilistic functions. For instance, in the standard library, Scallop provides a foreign predicate named `soft_eq`, that compares equality between two numbers. However, instead of returning exactly discrete false or true, the predicate wants to compute a probability of the two numbers being equal based on their distance. Formally, it is defined as follows:

$$\Pr(x = y) = \text{sech}^2\left(\frac{|y - x|}{2 \cdot \beta}\right) \tag{2.1}$$

Essentially, we have a parameter β dictating the threshold which the two numbers could be different. When $x = y$, we have the $\Pr(x = y) = 1$. When $x = 0.9$ and $y = 1.0$ and the parameter $\beta = 1.0$, we have $\Pr(x = y) \approx 0.998$, meaning that the two numbers are very close to each other. In Scallop, such program may be written as Listing 2.26.

Other use of foreign predicates producing probabilities include the similarity between vectors or high-dimensional tensors. We are going to show more examples of foreign predicates returning facts augmented with probabilities in Chapter 4.

2.7.3 Foreign Attributes

In Scallop, attribute is a higher-order construct that can be used to annotate any Scallop program item, including declaration of functions, predicates, facts, and rules. Attributes are constructs that starts with an `@` sign, and may be accepting arbitrary arguments, both positional and keyworded. Conceptually, one may think of attributes as taking in the annotated item, returning another item.

```
// [edge.csv]
from,to
0,1
1,2
```

Listing 2.27: A CSV file storing edges.

```
// [edge_path.scl]
@file("edge.csv", header=true)
type edge(from: u32, to: u32)
query edge // {(0, 1), (1, 2)}
```

Listing 2.28: A Scallop program that can load the edges in the given CSV file in Listing 2.27.

The following example in Listing 2.28 shows the use of a `@file` attribute to annotate a relation named `edge`. Specifically, it is telling Scallop to load an external CSV (comma-separated values) file, shown in Listing 2.27 into the `edge` relation. Conceptually, the `@file` attribute processes the otherwise empty relation `edge` and returns a relation `edge` filled with content loaded from the file.

In the standard library of Scallop, there are many existing foreign attributes. For example, `@storage` can be used to annotate a relation to specify the internal storage used for the relation, which can help programmers optimize the performance of the Scallop program. Moreover, `@cmd_arg` retrieves command line argument (if available) into the annotated relation. However, the power of having foreign attributes is only showcased when the set of attributes can be extended by external plugins and libraries. External databases, models, and applications can all become foreign attributes that annotate Scallop relations. We delay the discussion to Chapter 4.

Chapter 3

Core Reasoning Framework

The preceding chapter presented Scallop's surface language to express discrete reasoning. However, the language must also support differentiable reasoning to enable end-to-end training. In this chapter, 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 3.1). We then present a low-level representation SclRam, its operational semantics, and its interface to the rest of a Scallop application (Sections 3.2-3.3). We next present how our provenance framework enables probabilistic and differentiable reasoning (Sections 3.5-3.7). Lastly, we discuss practical extensions in Section 3.8.

3.1 Provenance Framework

A provenance framework propagates additional information (e.g. probability, proofs) alongside relational tuples in a Scallop program's execution. The framework is based on the theory of *provenance semirings* [33]. Figure 3.1 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 $\ominus$ serves as a customizable stopping mechanism for fixed-point iteration.

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

Figure 3.1: Core algebraic interface for provenance T.

$$
\begin{array}{rrcl}
\text{(Predicate)} & p & & \\
\text{(Aggregator)} & g & ::= & \texttt{count} \mid \texttt{sum} \mid \texttt{max} \mid \texttt{exists} \mid \ldots \\
\text{(Expression)} & e & ::= & p \mid \gamma_g(e) \mid \pi_\alpha(e) \mid \sigma_\beta(e) \\
 & & \mid & e_1 \cup e_2 \mid e_1 \bowtie e_2 \mid e_1 \times e_2 \\
 & & \mid & e_1 - e_2 \mid e_1 \triangleright e_2 \\
\text{(Rule)} & r & ::= & p \leftarrow e \\
\text{(Stratum)} & s & ::= & \{r_1, \ldots, r_n\} \\
\text{(Program)} & \overline{s} & ::= & s_1; \ldots; s_n
\end{array}
$$

Figure 3.2: Abstract syntax of core fragment of SclRam.

The above components together 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 by implementing this interface.

Example 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 operations like 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 chapter.*

A provenance must satisfy a few properties. First, $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ should form a commutative 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 3.3), it must also be *absorptive*, i.e., $t_1 \oplus (t_1 \otimes t_2) = t_1$ [20]. Moreover, we need $\ominus\, \mathbf{0} = \mathbf{1}$, $\ominus\, \mathbf{1} = \mathbf{0}$, $\mathbf{0} \not\Subset \mathbf{1}$, $\mathbf{0} \Subset \mathbf{0}$, 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.

$$
\begin{array}{rlcll}
\text{(Constant)} & \mathbb{C} & \ni & c & ::= \; int \mid bool \mid str \mid \ldots \\
\text{(Tuple)} & \mathbb{U} & \ni & u & ::= \; c \mid (u_1, \ldots, u_n) \\
\text{(Tagged-Tuple)} & \mathbb{U}_T & \ni & u_t & ::= \; t :: u \\
\text{(Fact)} & \mathbb{F} & \ni & f & ::= \; p(u) \\
\text{(Tagged-Fact)} & \mathbb{F}_T & \ni & f_t & ::= \; t :: p(u) \\
 & & & & \\
\text{(Set of Tuples)} & U & \in & \mathcal{U} & \triangleq \; \mathcal{P}(\mathbb{U}) \\
\text{(Set of Tagged-Tuples)} & U_T & \in & \mathcal{U}_T & \triangleq \; \mathcal{P}(\mathbb{U}_T) \\
\text{(Set of Facts)} & F & \in & \mathcal{F} & \triangleq \; \mathcal{P}(\mathbb{F}) \\
\text{(Database)} & F_T & \in & \mathcal{F}_T & \triangleq \; \mathcal{P}(\mathbb{F}_T)
\end{array}
$$

Figure 3.3: Annotations of semantic domains for SclRam.

3.2 SclRam Intermediate Language

Scallop programs are compiled to a low-level representation called SclRam. Figure 3.2 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 with aggregator g), projection (π_α with mapping α), and selection (σ_β with condition β), and binary operations union ($\cup$), product ($\times$), join ($\bowtie$), difference ($-$), and anti-join ($\triangleright$). We note that there are other binary operations such as intersection ($\cap$) which could be expressed by combining the above core operations.

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 an 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 $P_{s_i} \cap P_{s_j} = \emptyset$ for all $i \neq j$ in a program. Stratified negation and aggregation from the surface language are 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 Figure 3.3 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 $U_T \models u$ 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.

3.3 Operational Semantics of ScLRam

We now present the operational semantics for our core fragment of ScLRam in Figure 3.4. A ScLRam program $\overline{s}$ takes as input an *extensional database* (EDB) F_T, and returns an *intentional database* (IDB) $F'_T = [\![\overline{s}]\!](F_T)$. 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 Figure 3.4. 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 a (Cartesian) product $e_1 \times e_2$, each pair of incoming tuples is combined, and we use the provenance multiplication $\otimes$ to compute their tags.

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. Figure 3.5e and Figure 3.5f compare the evaluations of a difference expression under different semantics. While the tuple $(2, \mathrm{B})$ is removed from the outcome under untagged semantics, it is preserved under the tagged semantics.

Aggregation. Aggregators in ScLRam are discrete functions g operating on sets 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 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

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

$$\frac{t :: p(u) \in F_T}{t :: u \in [\![p]\!](F_T)} \text{ (Predicate)} \quad \frac{t :: u \in [\![e]\!](F_T) \qquad \beta(u) = \text{true}}{t :: u \in [\![\sigma_\beta(e)]\!](F_T)} \text{ (Select)}$$

$$\frac{t :: u \in [\![e]\!](F_T) \qquad u' = \alpha(u)}{t :: u' \in [\![\pi_\alpha(e)]\!](F_T)} \text{ (Project)} \quad \frac{t :: u \in [\![e_1]\!](F_T) \cup [\![e_2]\!](F_T)}{t :: u \in [\![e_1 \cup e_2]\!](F_T)} \text{ (Union)}$$

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

$$\frac{t :: u \in [\![e_1]\!](F_T) \qquad [\![e_2]\!](F_T) \not\models u}{t :: u \in [\![e_1 - e_2]\!](F_T)} \text{ (Diff-1)}$$

$$\frac{t_1 :: u \in [\![e_1]\!](F_T) \qquad t_2 :: u \in [\![e_2]\!](F_T)}{(t_1 \otimes (\ominus\, t_2)) :: u \in [\![e_1 - e_2]\!](F_T)} \text{ (Diff-2)}$$

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

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

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

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

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

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

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

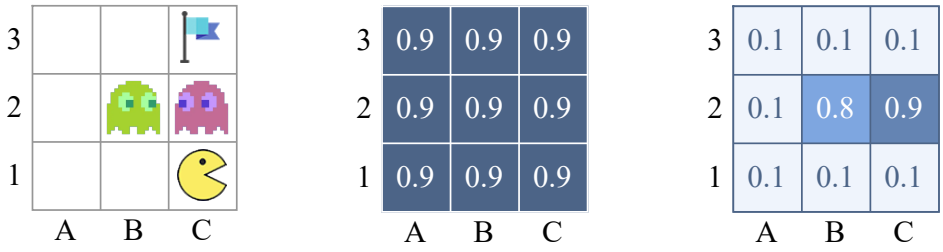
(Saturation) $F_T^{\texttt{old}} \doteq F_T^{\texttt{new}}$ iff $\forall t^{\texttt{new}} :: p(u) \in F_T^{\texttt{new}}, \exists t^{\texttt{old}} :: p(u) \in F_T^{\texttt{old}}$ such that $t^{\texttt{old}} \ominus t^{\texttt{new}}$

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

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

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

Figure 3.4: Operational semantics of core fragment of SclRam.



(a) Maze illustration (b) `grid_cell` (c) `enemy`

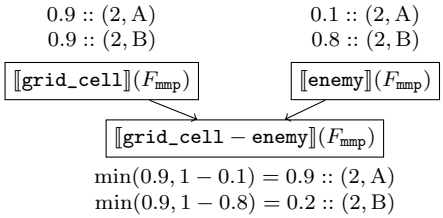
Scallop: `rel safe_cell(x, y)= grid_cell(x, y)and not enemy(x, y)`

↓

ScLRam Code: `safe_cell ← grid_cell − enemy`

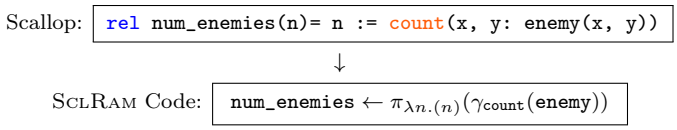
(d) A Scallop program and the compiled ScLRam program associated with it

(e) Untagged semantics (f) ScLRam tagged semantics

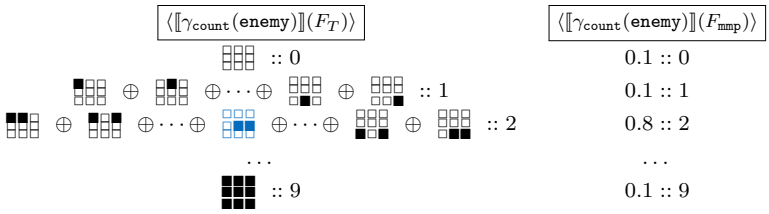


(g) ScLRam with `max-min-prob`

Figure 3.5: An example maze configuration is shown in (a), where each cell is represented by a tuple like $(1, \text{A})$. Suppose under the relations `grid_cell` and `enemy`, the cells are annotated by probabilities (shown in (b) and (c)). In (d), we demonstrate a Scallop rule computing the `safe_cell`s, which are cells that do not contain an enemy. The rule makes use of negation, and the compiled ScLRam code involves a difference operation on `grid_cell` and `enemy` relations. Figures (e), (f), and (g) illustrate evaluation of the ScLRam code under different semantics, where (g) instantiates the tagged semantics with `max-min-prob` provenance.



(a) A Scallop program and the compiled ScLRam program associated with it



(b) Evaluation of the aggregation expression

Figure 3.6: An example counting enemies in a PacMan maze shown in Figure 3.5a. Shown in (a) are the Scallop rule and compiled ScLRam rule with aggregation. For example, we have $t_{2\text{B}} :: \texttt{enemy}(2, \text{B})$ where $t_{2\text{B}} = 0.8$. In (b), we show two normalized ($\langle . \rangle$ defined in Figure 3.4) 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.,  $= t_{3\text{A}} \otimes (\ominus t_{3\text{A}}) \otimes \cdots \otimes (\ominus t_{1\text{C}})$. We mark the correct world  which yields the answer 2.

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. Figure 3.6 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 evaluating e may contain duplicate tuples tagged by distinct tags, owing to expressions such as union, projection, or aggregation. Thus, we perform *normalization* by joining ($\oplus$) the distinct tags corresponding to the same tuple. From here, there are three cases to merge the newly derived tuples ($\langle[\![e]\!](F_T)\rangle$) 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 tags ($t^{\texttt{old}} \oplus t^{\texttt{new}}$). Combining all cases, we obtain a set of newly tagged facts under predicate p.

Recursion in ScLRam is performed similarly to least fixed point iteration in Datalog [1]. 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.

```
rel path(x,y,u,v) = edge(x,y,u,v) and not enemy(u,v)
rel path(x,y,u,v) = path(x,y,z,w) and edge(z,w,u,v)
                    and not enemy(u,v)
```

↓

$$\texttt{temp} \leftarrow \pi_{\lambda((z,w),(x,y),(u,v)).((u,v),(x,y))}(\pi_{\lambda(x,y,z,w).((z,w),(x,y))}(\texttt{path}) \bowtie \texttt{edge})$$

$$\texttt{path} \leftarrow \pi_{\lambda((u,v),(x,y)).(x,y,u,v)}(\pi_{\lambda(x,y,u,v).((u,v),(x,y))}(\texttt{edge}) \triangleright \texttt{enemy})$$

$$\texttt{path} \leftarrow \pi_{\lambda((u,v),(x,y)).(x,y,u,v)}(\texttt{temp} \triangleright \texttt{enemy})$$

(a) The top box shows a Scallop program computing whether there is a path $(x, y) \to (u, v)$ without enemy, using transitive closure. The two Scallop rules are compiled to 3 SclRam rules (shown in the bottom box) where the first rule computes an auxilliary relation `temp` and the last two rules correspond to the rules in the Scallop program.

Iteration i	1	2	3	4	5	6	7
$t^{(i)}_{\texttt{1C-3C}}$ in $F^{(i)}_T$	–	$\boxed{\nearrow}$	=	$\boxed{\nearrow} \oplus \boxed{\nearrow} \oplus \cdots \oplus \boxed{\nearrow}$	=	$\boxed{\nearrow} \oplus \boxed{\nearrow} \oplus \cdots \oplus \boxed{\nearrow}$	=
$t^{(i)}_{\texttt{1C-3C}}$ in $F^{(i)}_{\texttt{mmp}}$	–	0.1	0.1	0.2	0.2	0.9	0.9
$t^{(i)}_{\texttt{1C-3C}}$ satu.?	–	F	T	F	T	F	T
$F^{(i)}_{\texttt{mmp}}$ satu.?	F	F	F	F	F	F	T

(b) An illustration of the tags that are evolving over iterations. In the figure, = means unchanged tag, satu. stands for saturated, while T and F represent true and false, respectively.

Figure 3.7: A demonstration of the fixed-point iteration to check whether actor at `1C` can reach `3C` without hitting an enemy (within the maze configuration shown in Figure 3.5a). The Scallop rule to derive this is defined on the top, and we assume bidirectional edges are populated and tagged by **1**. Let $t_{\texttt{1C-3C}}$ be the tag associated with `path(1,C,3,C)`. We use a symbol like $\boxed{\nearrow}$ to represent a conjunction of negated tags of `enemy` along the illustrated path, e.g. $\boxed{\nearrow} = (\ominus t_{\texttt{2C}}) \otimes (\ominus t_{\texttt{3C}})$. 2nd iter is the first time $t_{\texttt{1C-3C}}$ is derived, but the path $\boxed{\nearrow}$ is blocked by an enemy. On 6th iter, the best path $\boxed{\nearrow}$ is derived in the tag. After that, under the `max-min-prob` provenance, both the tag $t_{\texttt{1C-3C}}$ 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, SclRam with `mmp` saturates slower but allowing to explore better reasoning chains.

Figure 3.8: Execution pipeline with external interface.

Note that we define a specialized least fixed point operator $\mathbf{lfp}^\circ$, which stops the iteration once the whole database is *saturated*. Figure 3.7 illustrates an evaluation involving recursion and database saturation. The whole database saturates on the 7th iteration, and finds the tag associated with the optimal path in the maze. Termination is not universally guaranteed in ScLRam due to the presence of features such as value creation. But its existence can be proven on 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.

3.4 External Interface and Execution Pipeline

So 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 in Figure 3.8.

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 assigned the tag $\mathbf{1}$ 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.

Example 2 *The external interface of the* `max-min-prob` *provenance from Example 1 is* $([0, 1], [0, 1], id, id)$*, where the input and output spaces are the real numbers between 0 and 1, and the tagging and recover functions are the identity function* $id := \lambda x.x$*.*

$$
\begin{array}{rrcl}
\text{(Literal)} & \nu & ::= & v_i \mid \neg v_i \\
\text{(Conjunctive Clause)} & \eta & ::= & \nu_1 \wedge \cdots \wedge \nu_l \\
\text{(DNF Formula)} & \Phi \ni \phi & ::= & \eta_1 \vee \cdots \vee \eta_k
\end{array}
$$

Figure 3.9: Definitions related to boolean formulas in disjunctive normal form

3.5 Exact Probabilistic Reasoning with Provenance

We say that a provenance T is *probabilistic* if its input space I and output space O are real values in the range $[0, 1]$. As such, the `max-min-prob` provenance shown in Example. 1 is a probabilistic provenance. However, while useful in practice, `max-min-prob` only computes an approximation of the real probabilities. In this section, we start by introducing a more robust provenance that derives exact probabilities.

We introduce the provenance `proofs-prob` which keeps track of boolean formulas in disjunctive normal form (DNF). At a high level, the boolean formula encodes the full lineage of how a fact in the IDB is derived from existing facts in the EDB. The definitions for DNF formulas are shown in Figure. 3.9. Suppose there are n facts in the EDB with independent and identically distributed (i.i.d.) probabilities; we create n boolean variables each labeled $v_1, \ldots, v_n$. Then, a literal in the boolean formula is either a positive or a negated ($\neg$) boolean variable. A set of distinct literals connected by *and* ($\wedge$) form a conjunctive clause, while a set of clauses connected by *or* ($\vee$) form a disjunctive normal form formula ϕ. We note that there are two special DNF formulas, namely *true* ($\top$) and *false* ($\bot$). $\top$ is a singleton DNF formula with one empty conjunctive clause, where $\bot$ is an empty DNF formula. As such, the formal definition of `proofs-prob` is defined as follows:

Definition 1 *The base* ***`proofs-prob`*** *(**`pp`**) provenance is defined as the 7-tuple* $(\Phi, \bot, \top, \vee, \wedge, \neg, =)$*, where* $\vee$*,* $\wedge$*, and* $\neg$ *are operations on boolean formulae that perform the corresponding operation before normalizing the formula back into DNF. The external interface for* ***`proofs-prob`*** *provenance is defined as* $([0, 1], [0, 1], \tau_{pp}, \rho_{pp})$ *for:*

$$\tau_{pp}(p_i) = v_i \tag{3.1}$$

$$\rho_{pp}(\phi) = \mathit{WMC}(\phi, \Gamma) \tag{3.2}$$

where WMC is the function for Weighted Model Counting, and $\Gamma(v_i) = p_i$ *is the mapping from boolean variables* (v_i) *to their corresponding base probabilities* (p_i)*.*

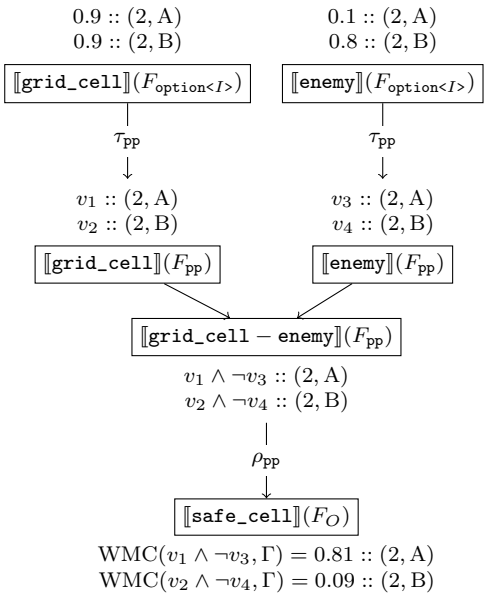


Figure 3.10: The SclRam evaluation result with `proofs-prob` on the rule shown in Figure. 3.5. As shown in the first row, we assume that facts in `grid_cell` and `enemy` are provided as base facts with given probabilities. Therefore, each of the 4 shown facts on the second row is assigned a unique boolean variable $v_1, \ldots, v_4$. The third row has the two IDB facts tagged by boolean formulas such as $v_1 \wedge \neg v_3$.

Following Figure. 3.5, we show one example of `proofs-prob`'s derivation process in Figure. 3.10. Here, in addition to the tag propagation process shown in the middle section, we also include the tagging phase at the top and the recovery phase at the bottom. The tagging function τ_{pp} transforms the input probabilities into internal tags. After the derivation of boolean formulas, we apply recovery function ρ_{pp} to compute the probabilities of the resulting facts. At the end, we note that the result computed from the weighted model counting (WMC) process is the probability of the corresponding tagged fact.

WMC essentially computes the *weight* of the boolean formula given the weights of the boolean variables. Here, we directly treat the probabilities associated with each input fact as the weight of the assigned boolean variables. Note that WMC is #P-complete, which presents a considerable tradeoff between computing exact probabilities and maintaining a feasible runtime. Indeed, compared to `max-min-prob` whose operations are all $\mathcal{O}(1)$, it is significantly more expensive to compute the exact probabilities. In later sections, we describe optimizations that facilitate efficient learning while maintaining various degrees of approximation.

```
rel label = {0.9::(o12, "cat"), 0.01::(o12, "flower")}
rel is_a = {("cat", "mammal"), ("mammal", "animal")}

// R1: recursively compute the labels of a given object
rel label(obj, np) = label(obj, n) and is_a(n, np)

// R2: query for objects that is an animal or a plant
rel target(obj) = label(obj,"animal") or label(obj,"plant")
```

(a) A rule used in common sense reasoning for deriving the label of an object given an ontology graph represented by the relation (`is_a`).

$$\dfrac{\dfrac{\texttt{label}(o_{12}, \texttt{cat}) \quad \{\{v_1\}\} \qquad \texttt{is\_a}(\texttt{cat}, \texttt{mammal}) \quad \{\{v_2\}\}}{\texttt{label}(o_{12}, \texttt{mammal}) \quad \{\{v_1, v_2\}\}}\ [\texttt{AND}] \qquad \texttt{is\_a}(\texttt{mammal}, \texttt{animal}) \quad \{\{v_3\}\}}{\texttt{label}(o_{12}, \texttt{animal}) \quad \{\{v_1, v_2, v_3\}\}}\ [\texttt{AND}]$$

(b) Proof construction with conjunction applying `R1`.

$$\dfrac{\texttt{label}(o_{12}, \texttt{animal}) \quad \{\{v_1, v_2, v_3\}\} \qquad \texttt{label}(o_{12}, \texttt{plant}) \quad \{\{v_4, v_5\}\}}{\texttt{target}(o_{12}) \quad \{\{v_1, v_2, v_3\}, \{v_4, v_5\}\}}\ [\texttt{OR}]$$

(c) Proof construction with disjunction applying `R2`.

Figure 3.11: Derivation of set-of-proofs under different operations.

3.6 Top-K Proofs Provenance for Scalable Reasoning

The probabilistic nature of our problem setting opens up room for approximation. A key observation is that, when the inference system is used in a learning setting, the probability of a ground truth fact should significantly outweigh the other facts, forming a skewed distribution. We can exploit this property by only including the "most likely" proofs.

First, we introduce a different way of formalizing the proofs and top-k proofs. We treat each DNF boolean formula ϕ as a set of proofs, where each proof is a set of literals. As such, $\bot = \emptyset$ while $\top = \{\emptyset\}$, a singleton set with $\emptyset$ being the only element. We showcase the process of proof construction using an example in Figure 3.11. Formally, the disjunction ($\vee$) operation is defined as the set union ($\cup$), while the conjunction ($\wedge$) operation is defined as cartesian product over proof-wise union:

$$\phi_1 \vee \phi_2 = \phi_1 \cup \phi_2 \tag{3.3}$$

$$\phi_1 \wedge \phi_2 = \{\eta_1 \cup \eta_2 \mid \eta_1 \in \phi_1, \eta_2 \in \phi_2\} \tag{3.4}$$

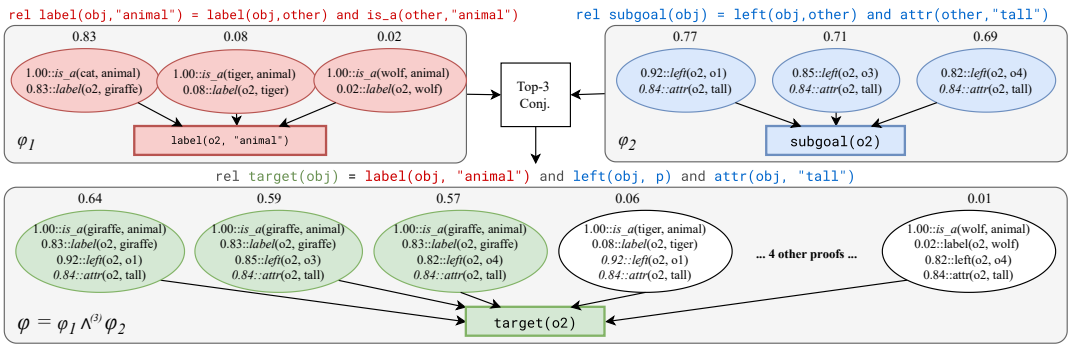


Figure 3.12: Illustration of top-k conjunction using $k = 3$. Each ellipse represents a proof of the fact shown in the box. Given the top 3 proofs for each of "label(o_2, "animal")" and "subgoal(o_2)", we wish to derive the top 3 proofs for their conjunction, "target(o_2)". The join yields 9 possible proofs. After computing the likelihood for each of the 9 proofs, we keep the top 3 most likely ones (green ellipses) and discard the rest (white ellipses).

In order to perform the approximation, we define the modified disjunction and conjunction operations, namely $\oplus^{(k)}$ and $\otimes^{(k)}$, where k is a tunable parameter controlling the level of approximation.

$$\phi_1 \vee^{(k)} \phi_2 = \text{top}_k(\phi_1 \cup \phi_2) \tag{3.5}$$

$$\phi_1 \wedge^{(k)} \phi_2 = \text{top}_k(\{\eta_1 \cup \eta_2 \mid \eta_1 \in \phi_1, \eta_2 \in \phi_2\}) \tag{3.6}$$

The goal is to pick out the "top-k" proofs within the result, where proofs are ranked by their respective probability. Specifically, the probability of each proof, $\Pr(\eta)$ is computed the following:

$$\Pr(\eta) = \begin{cases} 0 & \text{if the proof } \eta \text{ contains conflict;} \\ \prod_{\nu \in \eta} \Pr(\nu) & \text{otherwise} \end{cases} \tag{3.7}$$

$$\Pr(\nu) = \begin{cases} \Pr(v_i) & \text{if } \nu = v_i \text{ (a positive literal)} \\ 1 - \Pr(v_i) & \text{if } \nu = \neg v_i \text{ (a negative literal)} \end{cases} \tag{3.8}$$

Intuitively, whenever $\vee$ or $\wedge$ is performed, we rank proofs by their likelihood and preserve only the top-k proofs. This allows us to discard the vast majority of proofs and thus make inference tractable. When merging two proofs during $\wedge^{(k)}$, a single proof might contain the conjunction of conflicting literals, e.g. v_i and $\neg v_i$, in which case we remove the whole proof. An example run-through of *top-3 conjunction* ($\otimes^{(3)}$) is depicted in Figure. 3.12, where we perform a normal $\otimes$

operation followed by a top-3 filtering.

To take negation $\neg^{(k)}$ on DNF 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. The top-k operation is performed at the end, as the following:

$$\neg^{(k)}\ \varphi = \text{top}_k(\text{cnf2dnf}(\{\{\neg\nu \mid \nu \in \eta\} \mid \eta \in \varphi\})) \tag{3.9}$$

As such, all tags under the top-k proofs provenance have an upper bound of k on the number of proofs, making the WMC procedure tractable. We still have each conjunction operation taking $\mathcal{O}(n^2)$ and negation taking $\mathcal{O}(2^n)$, assuming that n is the number of facts and $k \ll n$. This allows the top-k proofs provenance to be much more scalable than the `prob-proofs` provenance.

We also note that our top-k inference algorithm is reminiscent of beam search. Both methods are iterative and explore only the top-k elements at each step. However, there are two major differences that distinguish us from beam search. First, while beam search is only a heuristic, our algorithm is backed by Datalog semantics and the provenance semirings framework for its correctness. We also present formal guarantees on its approximation error bound. Secondly, our algorithm operates over the beam of proofs ϕ for each derived fact, while beam search is usually performed to search for an output.

3.7 Differentiable Reasoning

We now elucidate how provenance also supports differentiable reasoning. Suppose we have n input facts that are associated with probabilities. 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 $\nabla\vec{y} = \frac{\partial \vec{y}}{\partial \vec{r}} \in \mathbb{R}^{m \times n}$. Viewing this from a learning perspective, $\vec{y}$ can be used for computing loss in subsequent steps, while $\nabla\vec{y}$ can be used for back-propagating gradients during optimization.

In Scallop, one can obtain these elements using a *differentiable provenance*. Differentiable provenances implement the external interface by setting the input tag space $I = [0, 1]$ and the output tag space O to be the space of *dual-numbers* $\mathbb{D}$ (Figure 3.13). 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

$$
\begin{aligned}
\hat{a}_i &= (a_i, \nabla a_i) \in \mathbb{D} \\
\hat{0} &= (0, \vec{0}) \\
\hat{1} &= (1, \vec{0})
\end{aligned}
$$

$$
\begin{aligned}
\hat{a}_1 + \hat{a}_2 &= (a_1 + a_2, \nabla a_1 + \nabla a_2) \\
\hat{a}_1 \cdot \hat{a}_2 &= (a_1 \cdot a_2, a_2 \cdot \nabla a_1 + a_1 \cdot \nabla a_2) \\
-\hat{a}_1 &= (-a_1, -\nabla a_1)
\end{aligned}
$$

$$
\begin{aligned}
\min(\hat{a}_1, \hat{a}_2) &= \hat{a}_i, \text{ where } i = \text{argmin}_i(a_i) \\
\max(\hat{a}_1, \hat{a}_2) &= \hat{a}_i, \text{ where } i = \text{argmax}_i(a_i) \\
\text{clamp}(\hat{a}_1) &= (\text{clamp}_0^1(a_1), \nabla a_1)
\end{aligned}
$$

Figure 3.13: Operations on dual-number $\mathbb{D} \triangleq [0,1] \times \mathbb{R}^n$, where n is the number of input probabilities.

Prov	T	$\mathbf{0}$	$\mathbf{1}$	$t_1 \oplus t_2$	$t_1 \otimes t_2$	$\ominus\, t$	$t_1 \ominus t_2$	$\tau(r_i)$	$\rho(t)$
`dmmp`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\max(t_1, t_2)$	$\min(t_1, t_2)$	$\hat{1} - t$	$t_1^{\text{fst}} == t_2^{\text{fst}}$	$(r_i, \vec{\mathbf{e}}_i)$	t
`damp`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\text{clamp}(t_1 + t_2)$	$t_1 \cdot t_2$	$\hat{1} - t$	true	$(r_i, \vec{\mathbf{e}}_i)$	t
`dtkp`	Φ	$\bot$	$\top$	$t_1 \vee^{(k)} t_2$	$t_1 \wedge^{(k)} t_2$	$\neg^{(k)}\, t$	$t_1 == t_2$	v_i	$\text{WMC}(t, \Gamma)$

Figure 3.14: Definitions of three differentiable provenances.

together y_j-s and ∇y_j-s respectively.

Scallop provides 8 configurable built-in differentiable provenances with different empirical advantages in terms of runtime efficiency, reasoning granularity, and performance. In the following subsections, we elaborate upon 3 simple but versatile differentiable provenances, whose definitions are shown in Figure 3.14. 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.

3.7.1 `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 $\mathcal{O}(1)$, making it extremely runtime-efficient.

3.7.2 `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 $\cdot$ 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 $\mathcal{O}(n)$, which is slower than `dmmp` but is still very efficient in practice.

3.7.3 `diff-top-k-proofs` (`dtkp`)

This provenance extends the *top-k proofs* introduced semiring originally proposed in [39] to additionally support negation and aggregation. As introduced in Section 3.6 and also shown in Figure 3.14, the tags of `dtkp` are boolean formulas $\varphi \in \Phi$ in *disjunctive normal form* (DNF). The difference between `dtkp` and the original top-k proofs provenance lies only in the external interface: differentiable provenances take dual-numbers as input tags and need to output dual-numbers as output tags. Specifically, the tagging and recover functions for `dtkp` are defined as:

$$\tau_{\texttt{dtkp}}(\Pr(v_i)) = v_i \tag{3.10}$$

$$\rho_{\texttt{dtkp}}(\varphi) = \text{WMC}(\varphi, \Gamma) \tag{3.11}$$

$$\Gamma(i) = (\Pr(v_i), \vec{\mathbf{e}}_i) \tag{3.12}$$

where WMC is now a differentiable weighted-model counting procedure adopted from [59]. Other than the boolean formula φ, WMC also takes in the weights of each probabilistic variable i. Instead of simple probabilities, the weights are now dual numbers like $(\Pr(v_i), \vec{\mathbf{e}}_i)$. During differentiable WMC, the dual-number addition and multiply rules (Figure 3.13) are applied. Implementation-wise, Scallop adopts Sentential Decision Diagrams (SDD) [21] for the WMC procedure.

3.8 Practical Extensions

In this section, we discuss the practical extensions that make Scallop's computation scalable, tractable, and widely-applicable.

```
1 rel color = {0.9::(OBJ_A, "red"); 0.1::(OBJ_A, "green")}
2 rel color = {0.2::(OBJ_B, "red"); 0.8::(OBJ_B, "green")}
3
4 rel should_not_exist(obj) =
5         color(obj, "red") and color(obj, "green")
```

Listing 3.1: Two sets of mutually exclusive facts under the same relation. We assume that an object cannot be "red" and "green" at the same time. Evaluating the rule on line 4 should result in an empty relation, if the mutual exclusions are properly handled.

3.8.1 Early Removal of Facts

A fact with a tag of $\mathbf{0}$ is often useless during computation, so it does not make sense to keep the facts that are tagged by $\mathbf{0}$. In Scallop's provenance framework, we allow each provenance to specify whether we want to earlier remove such facts. We introduce a new function to the provenance interface, discard : $T \to$ Bool. If discard returns true ($\top$) when called on the tag of a fact, then the fact will be removed from subsequent computation. The default implementation of this function is $\text{discard}(t) = t \ominus \mathbf{0}$.

3.8.2 Mutual Exclusivity of Facts

Recall that Scallop allows the user to specify mutually exclusive set of probabilistic facts (Listing 2.16). Mutual exclusivity of facts are *optionally* handled by each provenance. This is because the computational cost from fully handling mutual exclusivity may not be desirable. Specifically, handling mutual exclusivity would require the logical derivation process to be encoded explicitly to make sure that the satisfiability does not solely depend on two mutually exclusive facts. While this could be achieved in many ways, the `proofs` data structure used in provenances like `prob-proofs` and `top-k-proofs` can be naturally extended to handle mutual exclusion. On the contrary, simpler provenances like `max-min-prob` and `add-mult-prob` are unable to handle mutual exclusivity due to their tags being too simple.

We take `prob-proofs` as an example to show how it can be extended to handle mutual exclusivity. In Scallop, `prob-proofs` (along with others like `top-k-proofs`) are already extended with this functionality. But for presentation purpose, we consider a new provenance, named `prob-proofs-me`, where `me` stands for *mutual exclusion*. In `prob-proofs-me`, instead of accepting a simple probability as the input tag, it now accepts a tuple of probabilities along with an optional mutual exclusion set ID ($\mathbb{N}$). That is, $I_{\texttt{prob-proofs-me}} = [0, 1] \times \texttt{option<}\mathbb{N}\texttt{>}$.

Consider the example shown in Listing 3.1. The two sets of mutually exclusive facts are transformed into two distinct mutual exclusion IDs, which we label 0 and 1. The fact `color(OBJ_A, "red")` is technically tagged by $(0.9, 0)$. The first element 0.9 is treated as a normal probability, while the mapping from this fact ID to the mutual exclusion ID is stored for future reference. When executing the rule (line 4), we derive a temporary proof containing facts `color(OBJ_A, "red")` and `color(OBJ_A, "green")`. However, when looking up the mutual exclusion information, we find that the two facts cannot co-exist in the same proof. `prob-proofs` provenance will reject such a proof, rendering the result tag to be **0**. Therefore, combined with the early removal feature, the `should_not_exist` relation is computed to be empty, as desired.

3.8.3 Specializing for Provenances

Algorithm 1: Counting over max-min-prob tagged tuples

Data: $U_{\texttt{mmp}} = \{t_1 :: u_1, t_2 :: u_2, \ldots, t_n :: u_n\}$: $\mathcal{U}_{\texttt{mmp}}$, set of tagged-tuples to count
Result: $U'_{\texttt{mmp}}$: $\mathcal{U}_{\texttt{mmp}}$
`/* sort all positive tuples according to their tags from small to large. O(nlog(n)) */`
1 $\mathbf{t}^{\text{pos}} = sorted([t_i \mid i = 1 \ldots n])$;
2 $\mathbf{t}^{\text{neg}} = [1 - t^{\text{pos}}_{n-i+1} \mid i = 1 \ldots n]$;
`/* Iterate through all possible partitions between positive and negative tags. O(n) */`
3 $U'_{\texttt{mmp}} = \{t^{\text{neg}}_n :: 0, t^{\text{pos}}_1 :: n\}$;
4 **for** $i = 1 \ldots (n-1)$ **do**
5 ⌊ Add $\min(t^{\text{pos}}_{i+1}, t^{\text{neg}}_i) :: (n-i)$ to $U'_{\texttt{mmp}}$;
6 **return** $U'_{\texttt{mmp}}$

The design of Scallop's provenance framework allows the reasoning algorithms to be specialized for each provenance. For instance, as shown in Figure 3.4, aggregation operations in principle require the enumeration of subsets, which is inherently an $\mathcal{O}(2^n)$ operation, assuming that n is the number of facts for aggregation. However, not all aggregations need this complex reasoning. For instance, the `count` aggregator, when performed over a set of `max-min-prob` tagged-tuples, can be optimized to an $\mathcal{O}(n \log(n))$ operation. We present our optimized counting algorithm in Algorithm 1. Note that we only showed the algorithm for `mmp` for simplicity, but it easily extends to `dmmp`. Scallop implements many other optimizations with varying degrees of approximations so that operations that are in principle expensive become tractable when applied to real-life scenarios.

3.8.4 Sampling Operations

Scallop supports sampling operators such as `top`, `categorical`, and `uniform`. Their implementation requires a signal that ranks the tagged facts. We therefore introduce a new function weight : $T \to \mathbb{R}$ to our provenance. As the name suggests, the weight function takes in a tag and returns its weight. For probabilistic provenances, the default implementation is just the recover function, as it returns a probability $p \in [0, 1]$ that is also a suitable weight value. Weights can then be used for ranking facts or sampling with weights.

3.8.5 Provenance Selection

Given the rich library of Scallop provenances and operations, a natural question that arises is how to select a differentiable provenance for a given Scallop application. Based on our empirical evaluation, `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 should start with `dtkp` before searching other provenances. In general, provenance selection in Scallop is analogous to hyperparameter tuning in machine learning.

Chapter 4

Programming with Foundation Models

4.1 Foundation Models and Relations

Foundation models are deep neural models that are trained on a very large corpus of data and can be adapted to a wide range of downstream tasks [6]. Exemplars of foundation models include *language models* (LMs) like GPT [9], *vision models* like Segment Anything [46], and *multi-modal models* like CLIP [74]. While foundation models are a fundamental building block, they are inadequate for programming AI applications end-to-end. For example, LMs *hallucinate* and produce nonfactual claims or incorrect reasoning chains [61]. Furthermore, they lack the ability to reliably incorporate structured data, which is the dominant form of data in modern databases. Finally, composing different data modalities in custom or complex patterns remains an open problem, despite the advent of multi-modal foundation models such as ViLT [74] for visual question answering.

Various mechanisms have been proposed to augment foundation models to overcome these limitations. For example, PAL [28], WebGPT [66], and Toolformer [82] connect LMs with search engines and external tools, expanding their information retrieval and structural reasoning capabilities. LMQL [5] generalizes pure text prompting in LMs to incorporate scripting. In the domain of computer vision, neuro-symbolic visual reasoning frameworks such as VisProg [36] compose diverse vision models with LMs and image processing subroutines. Despite these advances, programmers

```
@gpt("The height of {{x}} is {{y}} in meters")
type height(bound x: String, y: i32)
// Retrieving height of mountains
rel mount_height(m, h) = mountain(m) and height(m, h)
```

(a) Program **P1**: Extracting knowledge using GPT.

```
@clip(["cat", "dog"])
type classify(bound img: Tensor, label: String)
// Classify each image as cat or dog
rel cat_or_dog(i, l) = image(i, m) and classify(m, l)
```

(b) Program **P2**: Classifying images using CLIP.

(c) Example input-output relations of the programs.

Figure 4.1: Two example programs in Scallop using foundation models.

lack a general solution that systematically incorporates these methods under a unified framework.

Scallop supports a *declarative* framework for programming with foundation models. In this framework, relations form the abstraction layer for interacting with foundation models. Our key insight is that foundation models are ***stateless functions*** *with relational inputs and outputs*. Figure 4.1a shows a Scallop program which invokes GPT to extract the height of mountains whose names are specified in a structured table. Likewise, the program in Figure 4.1b uses the image-text alignment model CLIP to classify images into discrete labels such as `cat` and `dog`. Figure 4.1c shows relational input-output examples for the two programs. Notice that the CLIP model also outputs probabilities that allow for probabilistic reasoning.

4.2 Extensible Plugin Library

Python libraries such as the OpenAI API and the Hugging Face ecosystem have positioned Python to be the dominant language for interacting with foundation models. This motivates a plugin library that allows users to interface Python-supported foundation models of their choosing

```python
@foreign_attribute
def clip(pred: Predicate, labels: List[str]):
  # Sanity checks for predicate and labels...
  assert pred.args[0].ty == Tensor and ...

  @foreign_predicate(name=pred.name)
  def run_clip(img: Tensor) -> Facts[str]:
    # Invoke CLIP to classify image into labels
    probs = clip_model(img, labels)
    # Each result is tagged by a probability
    for (prob, label) in zip(probs, labels):
      yield (prob, (label,)) # prob::(label,)

  return run_clip
```

Listing 4.1: Snippet of Python implementation of the foreign attribute `clip` which uses the CLIP model for image classification. Notice that the FA `clip` returns the FP `run_clip`.

in a Scallop program.

Each plugin defines a collection of foreign attributes (FAs) and functions via Scallop's foreign interface with Python. Our design principle for the interface is three-fold: simplicity, configurability, and compositionality. Listing 4.1 illustrates one succinct implementation of the FA that enables the use of the CLIP model shown in Figure 4.1b.

Because FAs can contain arbitrary Python code, the plugin library augments native Scallop features with a wide range of utility functions vital to AI applications. Some examples include plugins for image editing, face detection models, and chain-of-thought prompting. The modularity of the plugin library allows users familiar with Python to create and install custom plugins with ease.

4.3 Large Language Models

Text completion. In Scallop, language models like GPT [70] and LLaMA [95] can be used as basic foreign predicates for text completion (Listing 4.2). In this case, `gpt` is an arity-2 FP that takes in `request`, a `String` as the prompt, and produces `response`, a `String` as the response. As a result, we would obtain the fact `ans("8468000")`. We note that the foreign predicate `gpt` uses the model `gpt-3.5-turbo` by default.

To make the interface more relational and structural, we provide an FA for better specification of prompts, as shown in Listing 4.3. Here, we declare a relation named `population` which produces a population number (`num`) given a location (`loc`) as input. Notice that structured few-shot examples are provided through the argument `examples`. Under the hood, the foreign attribute fills the

```
extern type gpt(bound request: String, response: String)
rel ans(a) = gpt("population of NY is", a)
```

Listing 4.2: A snippet of Scallop using `gpt` as a foreign predicate.

```
@gpt("the population of {{loc}} is {{num}}",
  examples=[("NY", 8468000), ...])
type population(bound loc: String, num: u32)
```

Listing 4.3: A snippet of Scallop using `gpt` as a foreign attribute.

```
@gpt(
  "the mountain {{name}}'s height is {{height}} meters",
  examples=[("Kangchenjunga", 8586), ("Mont Blanc", 4805)]
)
type mountain_height(bound name: String, height: i32)

rel mountains = {"Mount Everest", "K2"}
rel result(name, height) = mountains(name) and mountain_height(name, height)
```

Listing 4.4: A snippet of Scallop using `@gpt` for querying mountain heights.

prompt with the given location at the *bound* argument `{{loc}}` and invokes GPT to fill in the *free* argument `{{num}}`.

Consider the Scallop program in Listing 4.4. Following the pattern described above, the call to `gpt` prompts GPT-4 (`gpt-4-0613`) by filling in `{{mountain_name}}` with the given strings and asks it to infer the value of `{{height}}` for each mountain. The shots provided in `examples` modify the prompt to GPT-4 as shown in Figure 4.2.

Note that we prompt GPT-4 to give its answer in the form of a JSON, so the response can be converted into a relational Scallop fact to be handled by the program.

Relation extraction. Structured relational knowledge embedded in free-form textual data can be extracted by language models. We introduce a foreign attribute `gpt_extract_relation` for this purpose. For instance, the predicate declared in Listing 4.5 takes in a `context` and produces `(subject, object, relation)` triplets.

This attribute differs from the text completion attribute `gpt` in that it can extract an arbitrary number of facts for multiple relations. To motivate the need for such an attribute, we consider the *date understanding* task from the BIG-bench suite [89]. In this task, the model is given a context and asked to compute a date in MM/DD/YYYY form.

Below is an example adapted from the date understanding task:

Q: Yesterday is February 14, 2019. What is the date 1 month ago from today?

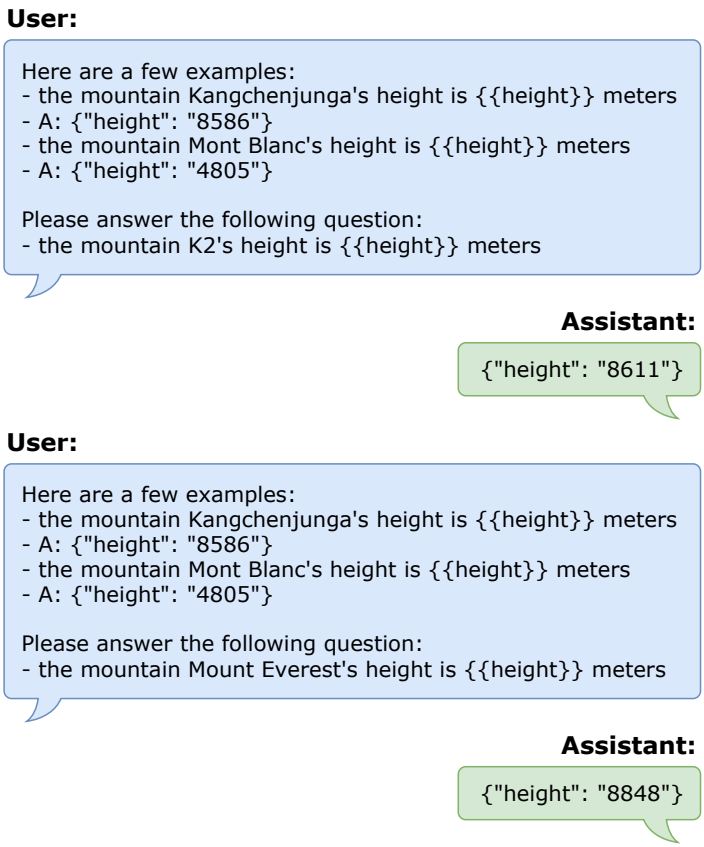


Figure 4.2: Conversation history between User (messages generated by `gpt` FA) and GPT-4 (`gpt-4-0613`) Assistant via OpenAI API after executing the program in Listing 4.4.

A: 01/15/2019

Now suppose we have access to the following relations in Scallop:

1. `mentioned_date(label, date)`: `label` is a string label for a date which is explicitly mentioned in the question context, and `date` is the corresponding MM/DD/YYYY string. When a date such as "Christmas Day" is mentioned, it will be transformed to the exact date of that year based on the common sense knowledge that the LLM possesses.

2. `goal(label)`: `label` is the date label whose MM/DD/YYYY form is requested as the answer.

3. `relationship(date_1, date_2, diff)`: the first two arguments are a pair of date labels relevant to the question, and `diff` is the time `Duration` between the dates.

```
@gpt_extract_relation(
  prompts=["What are the implied kinship relations?"],
  examples=[(
    // bound "context" argument
    "Alice and her son Bob went to...",
    // free "subject, object, relation" arguments
    // that form the relation to be extracted by GPT
    [("alice", "bob", "son"), ...]
  )]
)
type extract_kinship(
  bound context: String,
  subject: String,
  object: String,
  relation: String
)
```

Listing 4.5: A snippet of Scallop using `gpt_extract_relation` as a foreign attribute.

```
rel derived_date(label, date) =
  mentioned_date(label, date)
rel derived_date(label, date - diff) =
  relationship(label, other, diff) and
  derived_date(other, date)
rel derived_date(label, date + diff) =
  relationship(other, label, diff) and
  derived_date(other, date)
rel answer(date) =
  goal(label) and derived_date(label, date)
```

Listing 4.6: Scallop logic rules for the date understanding task.

Assuming that the above relations are supplied with complete and accurate facts, the Scallop rules in Listing 4.6 will derive the correct date in the relation `answer`. Motivated by this observation, we can use the rules annotated by `@gpt_extract_relation` in Listing 4.7 to define the GPT-4 prompt for extracting the three relations `mentioned_date`, `goal`, and `relationship` in Listing 4.8 before executing the rules above. Note that depending on the question context, the number of facts in relations `mentioned_date` and `relationship` could vary. Thus, text completion attributes are not sufficient for generating these relations.

Referring to Listing 4.7, each question provided in `prompts` (lines 2–6) corresponds to a relation of a given type signature that GPT-4 should extract from the bound argument `question` in Listing 4.8. The shots provided in `examples` (lines 7–17) are formatted as messages that are prepended to the conversation history given to GPT-4, as seen in Figure 4.3. Finally, the parameter `cot` (line 18) is a Boolean array where `cot[i]` toggles whether the ith relation should be extracted using zero-shot chain-of-thought (CoT) prompting [47].

Now suppose the bound argument `question` has value:

Jane finished her PhD in January 5th, 2008. Today is the 10th anniversary. What is

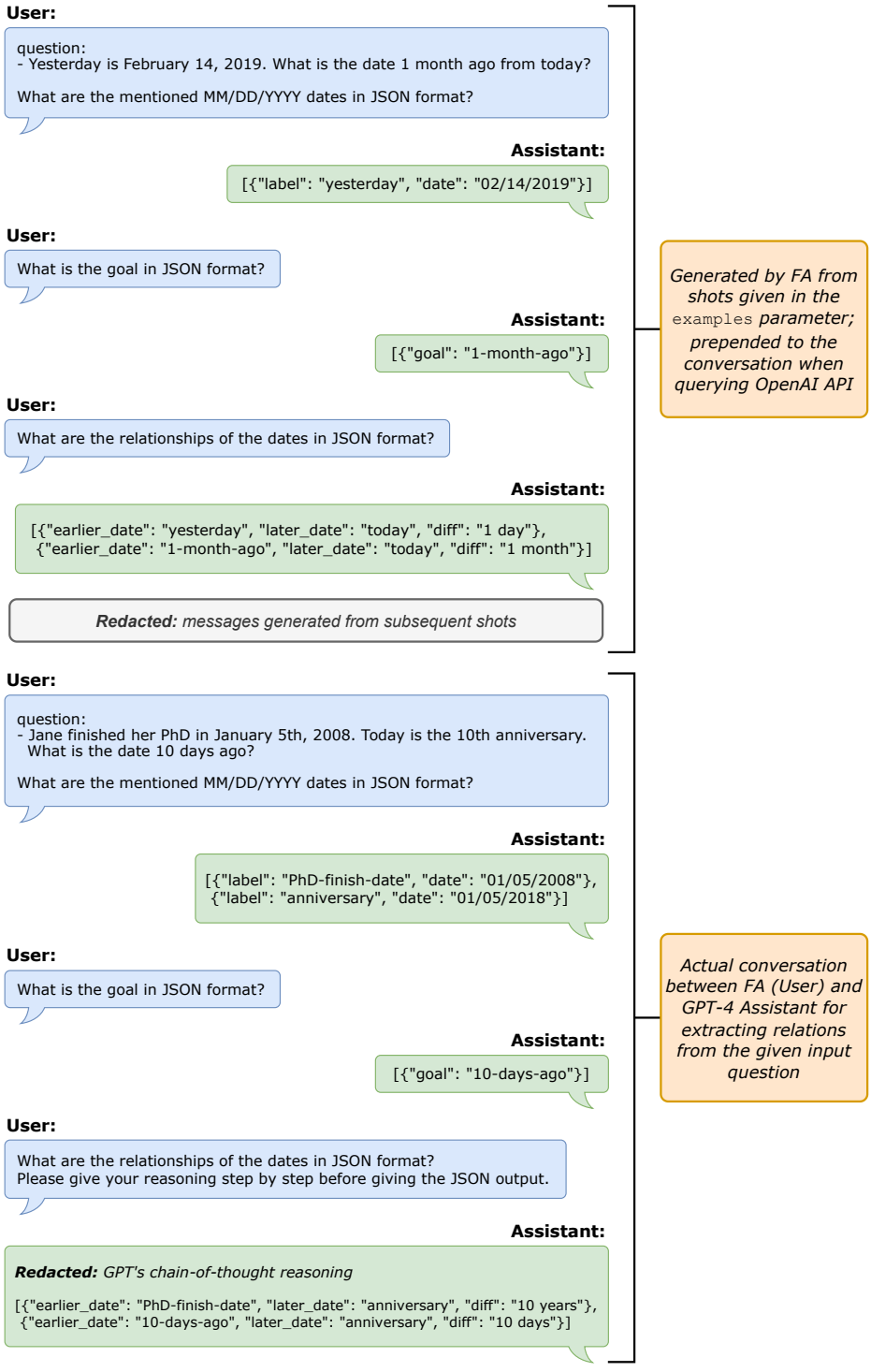


Figure 4.3: The GPT-4 conversation history after executing the program in Listing 4.7, with annotations and redactions in italics.

```
@gpt_extract_relation(
  prompts=[
    "What are the mentioned MM/DD/YYYY dates as JSONs?",
    "What is the goal in JSON format?",
    "What are the relationships of the dates as JSONs?"
  ],
  examples=[
    (
      ["Yesterday is February 14, 2019.
        What is the date 1 month ago from today?"],
      [
        [("yesterday", "02/14/2019")],
        [("1-month-ago")],
        [("yesterday", "today", "1 day"),
         ("1-month-ago", "today", "1 month")]
      ]
    ),
    // More shots hidden
  ],
  cot=[false,false,true]
)
type extract_mentioned_date (
  bound question: String, label: String, date: DateTime
),
extract_goal (bound question: String, goal: String),
extract_relationship (
  bound question: String, earlier_date: String,
  later_date: String, diff: Duration
)
```

Listing 4.7: FA-annotated rules for date understanding.

```
rel question = { "[Context] What is the date...?" }

rel mentioned_date(label, date) =
  question(q) and extract_mentioned_date(q, label, date)
rel goal(label) =
  question(q) and extract_goal(q, label)
rel relationship(l1, l2, diff) =
  question(q) and extract_relationship(q, l1, l2, diff)
```

Listing 4.8: Scallop rules for extracting 3 relations from a question for date understanding via the FA-annotated rules of Listing 4.7.

the date 10 days ago?

The GPT-4 conversation history after executing the code in Listing 4.7 and Listing 4.8 is given by Figure 4.3. With a little thought, the reader will find that applying the rules in Listing 4.6 on the relations generated by GPT-4 in Figure 4.3 will yield the correct answer: 12/26/2017.

This example points towards a general pattern for programming neuro-symbolically with foundation models. Given a problem, we decompose it into two sub-tasks. The first is to extract structured information with an LM via an FA like `gpt_extract_relation`. This is followed by logical reasoning and arithmetic over the structured data, expressed concisely as relational rules

```
@cross_encoder("nli-deberta-v3-xsmall")
type enc(bound input: String, embed: Tensor)
rel sim() = enc("cat", e) and enc("neko", e)
```

Listing 4.9: Scallop snippet using `cross_encoder` as a foreign attribute.

native to Scallop. By confining the LM's role to relation extraction, we mitigate the effects of model hallucination and make key reasoning steps more robust and interpretable.

4.4 Embedding Models and Vector Databases

```
@gpt_encoder
type $embed_text(String) -> Tensor

type question(q: String)

type context(id: i32, c: String)

rel relevant(id) = id := top<2>(
  id: question(q) and
  context(id, c) and
  soft_eq<Tensor>($embed_text(q), $embed_text(c))
)
rel relevant_context($string_concat(c1, "\n", c2)) =
  relevant(id1) and relevant(id2) and id1 < id2 and
  context(id1, c1) and context(id2, c2)

@gpt(
  prompt="Given {{ctxt}}\n{{q}}\n
    Please think step-by-step {{ans}}"
)
type qa(bound q: String, bound ctxt: String, ans: String)

rel answer(a) =
  question(q) and relevant_context(c) and qa(q, c, a)
```

Listing 4.10: Scallop program for information retrieval in the HotpotQA task.

Textual embeddings are useful in performing tasks such as information retrieval. In Scallop, embedding models are usually modeled as foreign predicates. Listing 4.9 declares an FP encapsulating a cross-encoder [67].

In line 3, we compute the cosine-similarity of the encoded embeddings using a soft-join on the variable `e`. As a result, we obtain a probabilistic fact like `0.9::sim()` whose probability encodes the cosine-similarity between the textual embeddings of `"cat"` and `"neko"`.

One application of these techniques is in information retrieval. For example, consider the task from HotpotQA [102]. In this Wikipedia-based question answering (QA) dataset, the model takes

```
@owl_vit(["human face", "rocket"])
type find_obj(
    bound img: Tensor, id: u32,
    label: String, cropped_image: Tensor
)
```

Listing 4.11: Scallop snippet using `owl_vit` as a foreign attribute.

```
@stable_diffusion("stable-diffusion-v1-4")
type gen_image(bound txt: String, img: Tensor)
```

Listing 4.12: Scallop snippet using `stable_diffusion` as a foreign attribute.

an input with 2 parts: 1) a question, and 2) 10 Wikipedia paragraphs as the context for answering the question. Among the 10 Wikipedia pages, at most 2 are relevant to the answer, while the others are distractors.

In Listing 4.10, we implement an adaptation of FE2H [52]. The method is a 2 stage procedure. First, we turn the 10 documents into a vector database by embedding each document with the `gpt_encoder` FP (lines 1–2, 11). We then use cosine similarity (via Scallop's built-in `soft_eq`) to select the 2 documents most relevant to the question (lines 8–15), which are provided as context to GPT-4 to do QA (lines 17–24). By retrieving only 2 documents, the context we generate is inherently less distracting than the naive context of all 10 documents.

4.5 Vision and Multi-Modal Models

Image classification models. Image-text alignment models, such as CLIP [74], can be used off-the-shelf as zero-shot image classification models. Figure 4.1b shows an example usage of the `@clip` attribute. We also note that dynamically-generated classification labels can be provided to CLIP via a bounded argument in the predicate.

Image segmentation models. OWL-ViT [62], Segment Anything Model (SAM) [46], and Dual-Shot Face Detector (DSFD) [50] are included in Scallop as image segmentation (IS) and object localization (LOC) models. IS and LOC models can provide many outputs, such as bounding boxes, classified labels, masks, and cropped images.

For instance, the OWL-ViT model can be used and configured as shown in Listing 4.11. Here, the `find_obj` predicate takes in an image, and finds image segments containing "human face" or "rocket". According to the names of the arguments, the model extracts 3 values per segment: ID,

Figure 4.4: The face-tagging input (left) and output (right) of the image with descriptive filename `microsoft_ceos.jpeg`.

label, and cropped image. Note that each produced fact is tagged with a probability, representing the model's confidence.

Image generation models. Visual generative models such as Stable Diffusion [78] and DALL-E [76] can be regarded as relations as well.

Listing 4.12 shows the declaration of the `gen_image` predicate, which encapsulates a diffusion model. As can be seen from the signature, it takes in a `String` text as input and produces a `Tensor` image as output. Optional arguments such as the desired image resolution and the number of inference steps can be supplied to dictate the granularity of the generated image.

An example in compositionality. To demonstrate Scallop's usefulness in composing foundation models of various modalities, we introduce our face-tagging task based on that of VisProg [36]. In our task, the model is given an image with a descriptive natural-language filename, and needs to output an edited image where all faces relevant to the description are boxed with their names. An example input-output pair is shown in Figure 4.4.

The code for face-tagging is provided in Listing 4.13. Our solution obtains a set of possible names from GPT-4 (lines 5–12) and candidate faces from the DSFD face detection model (lines 14–26). These are provided to CLIP for object classification (lines 28–31), after which probabilistic reasoning filters the most relevant face-name pairs (lines 33–37). Finally, the program calls image-editing foreign functions from the plugin library that use the face-name pairs to draw the captioned face boxes (code omitted).

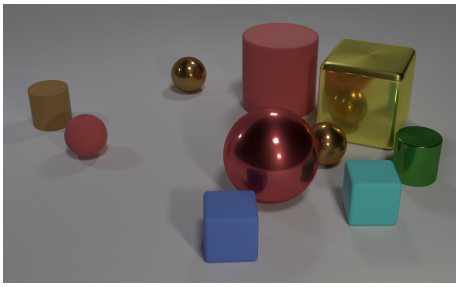
```
type input_path(String)
type input_name(String)
rel image($load_image(path)) = input_path(path)

@gpt(
  prompt="Give a semicolon-delimited list of people that
    could appear in an image titled `{{name}}', where each
    item is a person's name: {{list}}"
)
type list_gpt(bound name: String, list: String)

rel names(list) = input_name(name) and list_gpt(name, list)

@face_detection(
  ["cropped-image", "bbox-x", "bbox-y",
   "bbox-w", "bbox-h"],
  enlarge_face_factor=1.3
)
type face(bound img: Tensor, id: u32,
  face_img: Tensor, x: u32, y: u32, w: u32, h: u32
)

rel face_image(id, face_img) =
  image(img) and face(img, id, face_img, _, _, _, _)
rel face_bbox(id, x, y, w, h) =
  image(img) and face(img, id, _, x, y, w, h)

@clip(prompt="the face of {{}}", score_threshold=0.8)
type face_name(
  bound face: Tensor, bound list: String, name: String
)

rel identity(id, name) =
  name := top<1>(name:
    face_name(img, $string_concat(list), name) and
    face_image(id, img) and names(list)
  )

// Omitted: code for labeling identified faces w/ boxes
```

Listing 4.13: Scallop program for face-tagging.

4.6 Case Study: Visual Question Answering on Scene Images

In this section, we describe the Scallop program for one of our benchmark applications, CLEVR [43]. In Figure 4.5, we illustrate a concrete example of the program we described executing on an image from the CLEVR dataset. We are given two inputs, namely the image (left) and the question (top-right), and we are supposed to produce an answer (bottom-right). In general, the images in CLEVR dataset may contain up to 10 primitive objects, each with a pre-defined set of shapes, colors, materials, and sizes. There is a range of questions whose answer maybe numbers (counting questions), true/false (existence or comparing or assertive questions), or properties (querying color).

We decompose our solution to this application into three sub-tasks: a) extracting a structured



```
Question: Is there a yellow cube?
```

```
Programmatic Query:
  Exists(FilterColor(
    FilterShape(Scene(),"cube"),
    "yellow"))
```

```
Answer: true
```

Figure 4.5: An example problem in CLEVR. Our model is supposed to answer the given question based on the image shown on the left.

```
@owl_vit(["cube", "sphere", "cylinder"],
  expand_crop_region=10, limit=10,
  flatten_probability=true)
type segment_image(
  bound img: Tensor, id: u32,
  cropped_image: Tensor, area: u32,
  bbox_center_x: u32, bbox_bottom_y: u32)
```

Listing 4.14: Definition of the relation used for image segmentation, using OWL-ViT.

scene graph from the input image, b) extracting an executable query program from the input natural language (NL) question, and c) combining both to answer the question based on the scene graph. Here, a) and b) require the processing of unstructured data such as image and natural language question, and therefore may be *neural*. On the other hand, c) can be programed and fully symbolic. We may choose to have both neural networks for a) and b) to be trained by our end-to-end pipeline. But in light of the advancements of foundation models such as GPT-4 [70] and CLIP [74], in this section we present an off-the-shelf no-training solution. We next describe how we solve each of these sub-tasks.

4.6.1 Image to structured scene graph

To convert image to structured scene graph, we use two vision models, namely OWL-ViT [62] and CLIP [74]. We use OWL-ViT for obtaining object segments and CLIP models for classifying object properties. The goal is to construct scene graph which contains the following information: the shape, color, material, and size for each object, and the spatial relationships between each pair of objects.

Our object detection predicate is defined in Listing 4.14. We use the `@owl_vit` foreign attribute to decorate a predicate `vit_segment_image`. Here, the image has one bounded argument which

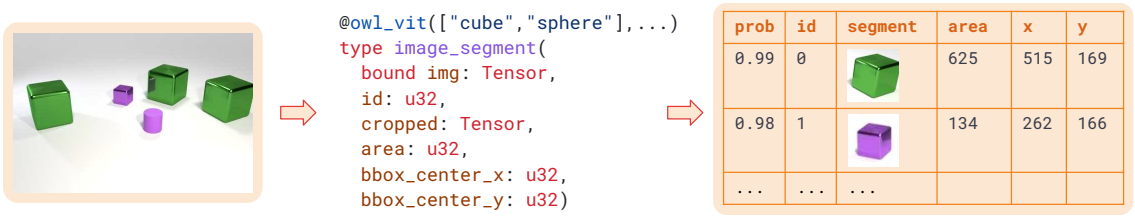


Figure 4.6: An illustration of the `segment_image` relation.

```
1  // the user provide the image with its directory
2  type img_dir(directory: String)
3
4  // load the image as a tensor
5  type image(img: Tensor)
6  rel image($load_image(d)) = img_dir(d)
7
8  // segment the image using our segment_image relation
9  rel obj_seg(id, seg, area, x, y) =
10   image(img) and segment_image(img, id, seg, area, x, y)
11 rel obj(id) = obj_seg(id, _, _, _, _)
```

Listing 4.15: The Scallop program that loads and segments a CLEVR image.

is the input image, and it produces image segments represented by 5 tuples, containing segment id (`id`), segmented image (`cropped_image`), the area of segment (`area`), the center x coordinate (`bbox_center_x`), and the bottom y coordinate (`bbox_bottom_y`). Specifically, segmented images can be passed to downstream image classifiers, the area is used to classify whether the object is big or small, and the coordinates are used to determine spatial relationships between objects. We illustrate the produced table in Figure 4.6.

Note that the arguments we pass to `@owl_vit` contain expected labels of `cube`, `sphere`, and `cylinder`. Because OWL-ViT does not perform well at classifying given geometric objects by shape, we do not use it to query the labels associated with each object. Rather, these labels identify the image segments the model extracts from the base image.

We set `expand_crop_region` to be 10, which expands the cropped images by the given factor. Since the bounding boxes of the objects are tight, enlarging the crop region can help subsequent classifiers to better see the object. With the `limit` set to 10, OWL-ViT only generates 10 image segments. Lastly, we set `flatten_probability` to be `true`. This is due to that OWL-ViT is not trained on CLEVR, so it produces very low confidence scores on all recognized objects. In order to not let the scores affect downstream computation, we overwrite the probability to 1 for all objects.

With all the above setup, we may load the image specified by the image directory path using

```
// Classify each object into a certain shape
@clip(["cube", "cylinder", "sphere"],
  prompt="a {{}} shaped object")
type classify_shape(bound obj_img: Tensor, shape: String)
rel shape(o, s) = obj_seg(o, seg, _, _, _) and
  classify_shape(seg, s)

// Classify each object into a certain color
@clip(["red", "blue", "yellow", "purple", "gray", ...],
  prompt="a {{}} colored object")
type classify_color(bound obj_img: Tensor, color: String)
rel color(o, c) = obj_seg(o, seg, _, _, _) and
  classify_color(seg, c)
```

Listing 4.16: Classifier relations using CLIP.

```
// obtain the object position
rel obj_pos(o, x, y) = obj_seg(o, _, _, x, y)

// left/right spatial relation
rel relate(o1, o2, if x1 < x2 then "left" else "right") =
  obj_pos(o1, x1, _) and obj_pos(o2, x2, _) and o1 != o2

// front/behind spatial relation
rel relate(o1, o2, if y1 > y2 then "front" else "behind") =
  obj_pos(o1, _, y1) and obj_pos(o2, _, y2) and o1 != o2
```

Listing 4.17: Scallop rules for deriving spatial relations between pairs of objects.

the foreign function `$load_image`, and then segment the image using the `segment_image` predicate defined previously. Our code is illustrated in Listing 4.15.

We next define the classifiers for shape, color, material, and sizes. For instance, we utilize the foreign attribute `@clip` to classify each object segment with a label among three possible shapes: `cube`, `sphere`, and `cylinder` (Listing. 4.16). In order to interface with CLIP, we write a prompt `"a {{}} shaped object"`. Each label is used to replace the `{{}}` pattern in the prompt, producing short phrases like "a cube shaped object". Then, the three prompts are passed to CLIP along with the object image, and facts with labels are returned with probabilities. The classifier for color is done similarly, shown also in Listing. 4.16.

The spatial relationship (`left`, `right`, `front`, and `behind`) is derived from object coordinates (Listing 4.17). We note that we are not using a neural component for this because the spatial relationships from object coordinates are fairly precise. Combining everything together, we have produced the relationships `color`, `shape`, `material`, `size`, and `relate`, forming the scene graph of the image.

```
type Query = Scene()
    | FilterShape(Query, String) // and material/color/size
    | MoreThan(Query, Query)     // and less-than/equals
    | SameSize(Query)            // and color/material/size
    | QueryColor(Query)          // and size/shape/material
    | Count(Query)
    | Exists(Query)
    | Relate(Query, String)
    | // ... and other variants
```

Listing 4.18: The DSL for representing the NL questions in the CLEVR dataset, defined in Scallop.

```
@gpt_semantic_parse(
  header="""
    Please convert a question into its programmatic form
    according to the following language:

    Expr = Scene() | FilterShape(Expr, String) | ...

    Please pick shapes among \"cylinder\", ...;
    Colors are among \"red\", \"blue\", ...;
    Materials are among \"shiny metal\" and ...;
    Sizes are among \"large\" and \"small\";
    Spatial relations are among \"left\", ..."""",
  prompt="""Question: {{s}} \n Query: {{e}}""",
  examples=[
    ("How many red objects are there?",
      "Count(FilterColor(Scene(), \"red\"))"),
    ("Is there a cube?",
      "Exists(FilterShape(Scene(), \"cube\"))"),
    ...],
  model="gpt-4")
type parse_query(bound s: String, q: Query)

// convert the input NL question to a programmatic query
type question(q: String)
rel prog_query(q) = question(s) and parse_query(s, q)
```

Listing 4.19: A semantic parser relation `parse_query`.

4.6.2 NL question to programmatic query

We use the GPT-4 model [70] for converting a natural language question into a programmatic query. The first step is defining the domain specific language (DSL) for querying the CLEVR dataset, as shown in Listing 4.18. Notice that the DSL is represented by the user-defined algebraic data type (ADT) `Query`, which contains constructs for getting objects, counting objects, checking existence of objects, and even comparing counts obtained from evaluating multiple queries.

We then create the semantic parser for the DSL by configuring a relation to parse natural language question into a programmatic `Query`, shown in Listing 4.19. For this, we utilize the `@gpt_semantic_parse` foreign attribute provided in Scallop. Other than the `model` argument which is used to specify the OpenAI model to call, we pass the 3 main arguments to `gpt_semantic_parse`,

namely `header`, `prompt`, and `examples`. `prompt` constitutes the system prompt, while the structures `examples` are expanded with the `prompt` into the few-shot examples. In Figure 4.7 we show one specific example of "conversation" with LLM to precisely parse the NL question into `Query`.

4.6.3 Putting it all together

The last part which brings everything together is the semantics of our `Query` DSL, shown in Listing 4.18. We can start by treating each variant of our DSL as a *function*. Assuming we have $O_{\text{all}} = \{o_1, o_2, \ldots, o_n\}$ representing the set of all objects in the scene. Then we have an arbitrary set of objects represented as $O \in \mathcal{P}(O_{\text{all}})$ where $\mathcal{P}$ is the powerset operation. Suppose $\mathcal{O} = \mathcal{P}(O_{\text{all}})$, we may give the following function types and functional semantics to a few set of variant in our DSL (Figure 4.8). For instance, `Scene` is a function that returns all the objects in the scene. `Count` takes in a set of objects and returns the cardinality of that set. Assuming we have relational predicates such as `shape` and `color` pre-populated with facts in our scene graph, we may use them to define the functions such as `FilterShape` and `QueryColor`. Definitions of other predicates are omitted since they look similar to what we show.

Unsurprisingly, it turns out that the definition of these functions can all be translated into relational rules. We may define `eval` which recursively evaluates each "function call" into their respective output. Note that since the functions have different return types, we define different `eval_*` relations, shown in Listing 4.20. Specifically for `eval_obj`, even though the original functions return sets of objects, we may define the relation relating the function with one of object in its output set. Such representation is natural (and unique) in relational programming paradigm–it allows us to tag each output with probabilities, while in traditional functional semantics might be very hard to do.

With everything setup, we can now start defining the semantics of our DSL in Scallop (Listing 4.21). The semantics is inductively defined on the `Query` data structure. Each rule essentially encodes the evaluation of one variant in our DSL. For instance, the rule on line 2 states that evaluating the `Scene()` results in any object `o` where `o` is an object. The rule on line 3-4 handles the `FilterShape(e1, s)` query: it evaluates the subquery `e1` to obtain object `o`, and further quantify it using the `shape(o, s)` atom to make sure that it has the desired shape. For the rule handling count and exists, we directly use the corresponding aggregator in Scallop. Note that we use the explicit group-by operation with the `where` keyword, so that the default behavior is to return 0

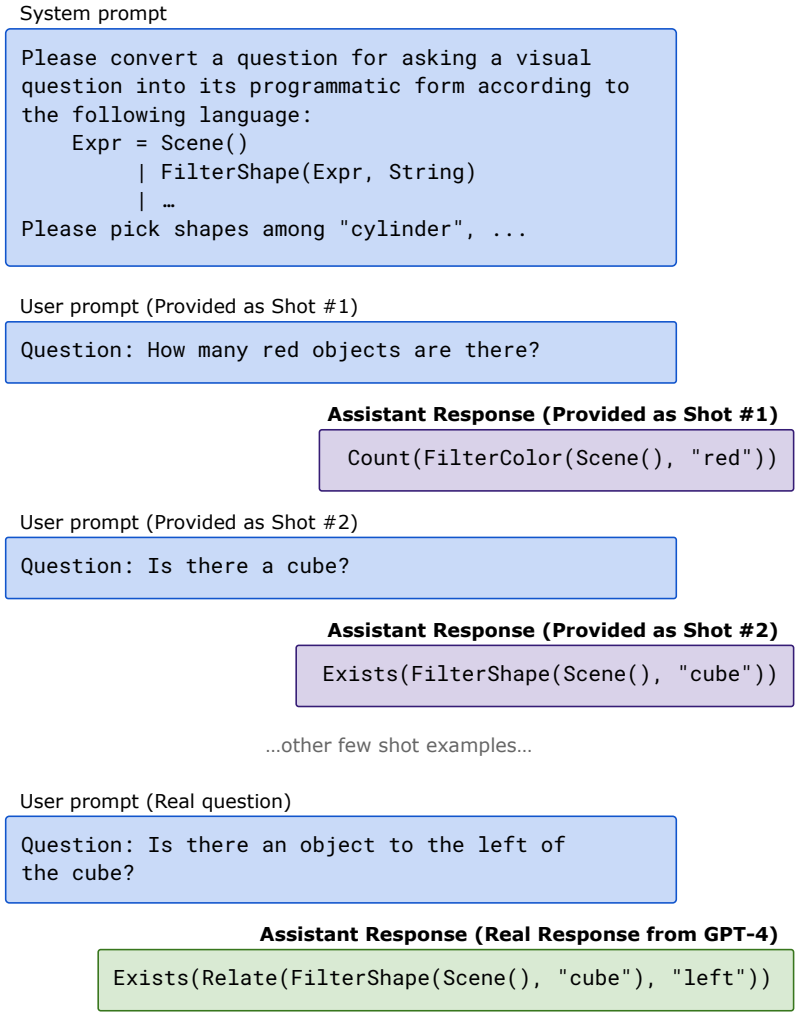


Figure 4.7: A "conversation" between user and the LLM for semantically parsing the NL question into programmatic query in our domain specific language (Listing 4.18). We use few-shot prompting in order to generate accurate programmatic query. Everything except the last bubble (green) is generated by our `@gpt_semantic_parse` foreign attribute–the assistance response for few-shot examples are also mocked to give the LLM an impression of the expected output format.

$$
\begin{array}{rll}
\texttt{Scene}: & () \to \mathcal{O} & \texttt{Scene}() = O_{\text{all}} \\
\texttt{Count}: & \mathcal{O} \to \mathbb{N} & \texttt{Count}(O) = |O| \\
\texttt{Exists}: & \mathcal{O} \to \mathbb{B} & \texttt{Exists}(O) = \mathbf{1}_{|O|>0} \\
\texttt{FilterShape}: & \mathcal{O} \times \texttt{S} \to \mathcal{O} & \texttt{FilterShape}(O, s) = \{o \mid o \in O \wedge \texttt{shape}(o, s)\} \\
\texttt{QueryColor}: & O_{\text{all}} \to \texttt{C} & \texttt{QueryColor}(o) = c \text{ where } \texttt{color}(o, c) \\
\texttt{MoreThan}: & \mathcal{O} \times \mathcal{O} \to \mathbb{B} & \texttt{MoreThan}(O_1, O_2) = \mathbf{1}_{|O_1|>|O_2|}
\end{array}
$$

Figure 4.8: The functional semantics of our defined DSL. We show the type of each "function" as well as their concrete definitions. Here, $\texttt{S} = \{\texttt{big}, \texttt{small}\}$ represents the set of shapes and $\texttt{C} = \{\texttt{red}, \texttt{blue}, \dots\}$ represents the set of all possible colors appearing in the dataset.

```
// for functions like Count that return numbers
type eval_num(q: Query, n: usize)

// for functions like Exists that return booleans
type eval_bool(q: Query, b: bool)

// for functions like Scene that return (set of) objects
// note: we use `u32` to represent Object IDs
type eval_obj(q: Query, obj_id: u32)

// for functions like QueryColor that return attributes
// the attributes are stringified for the result, such as
// "red", "cube", "left", "large", and etc.
type eval_str(q: Query, attr: String)
```

Listing 4.20: The type declarations of `eval_*` relations in Scallop.

```
// evaluating variants which return set of objects...
rel eval_obj(Scene(), o) = obj(o)
rel eval_obj(FilterShape(e1, s), o) =
    eval_obj(e1, o) and shape(o, s)

// evaluating variants which return numbers...
rel eval_num(e, n) = n := count(
    o: eval_obj(e1, o) where e: case e is Count(e1))

// evaluating variants which return boolean...
rel eval_bool(e, b) = b := exists(
    o: eval_obj(e1, o) where e: case e is Exists(e1))
rel eval_bool(MoreThan(e1, e2), n1 > n2) =
    eval_num(e1, n1) and eval_num(e2, n2)

// evaluating variants which return attributes...
rel eval_str(QueryColor(e),c) = eval_obj(e, o), color(o, c)
```

Listing 4.21: The semantics of CLEVR DSL defined in Scallop.

(for `count`) or `false` (for `exists`).

As shown by the rules, they are defined relatively concisely. Be reminded that while the rules look like their functional counterparts, they actually have their underlying probabilistic and differentiable semantics. As such, the probabilities produced by image segmentation models and classifiers can propagate to produce a probabilistic distribution of answers.

Chapter 5

Scallop Benchmarks and Evaluations

In this chapter, we present the benchmarks and evaluations to showcase the applicability and effectiveness of Scallop. We begin with the basic benchmarks where Scallop programs are connected to simple neural models that are trained or fine-tuned (Section 5.1). We next showcase the benchmarks where Scallop programs are interfaced with pre-trained foundation models (Section 5.2). To concretely showcase the programming paradigm with Scallop, we then dive into 4 case studies from simple to complex. Specifically, they are MNIST-Sum-2 (Section 5.3), Hand-Written Formula Evaluation (Section 5.4), PacMan-Maze (Section 5.5), and CLUTRR (Section 5.6).

5.1 Basic Scallop Benchmarks

First, we evaluate the Scallop language and framework on a basic benchmark suite comprising eight neurosymbolic applications. Here, 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?

In the following sections, we first introduce the benchmark tasks and the chosen baselines for each task (Section 5.1.1). Then, we answer **RQ1** to **RQ4** in Section 5.1.2 to Section 5.1.4 respectively.

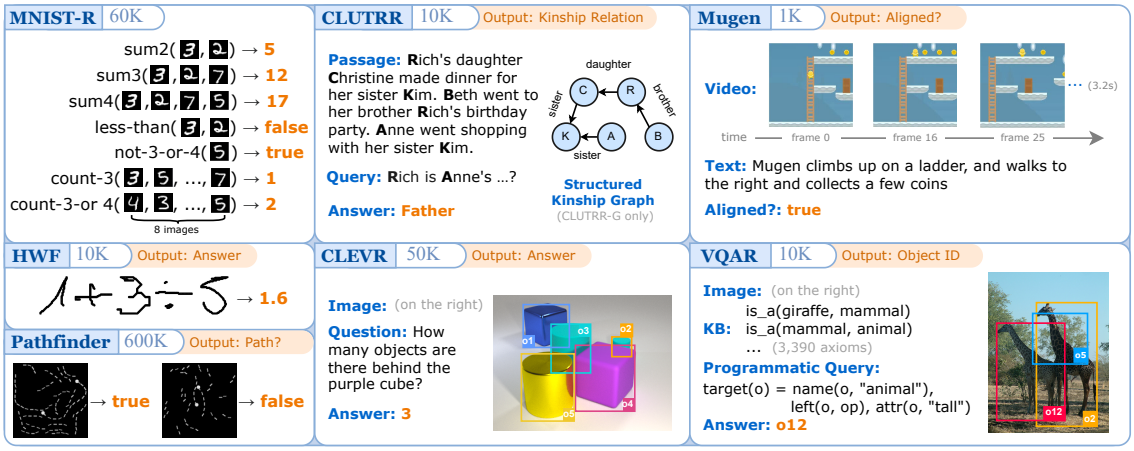


Figure 5.1: Visualization of benchmark tasks. Beside the name of each task we specify the size of the training dataset and the output domain. PacMan-Maze is omitted since it will be presented in detail in Section 5.5.

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.

5.1.1 Benchmarks and Baselines

We present an overview of our benchmarks in Figure 5.1. 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 hand-written digits from the MNIST dataset [48] 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 this test suite, we use a CNN-based model, DeepProbLog (DPL) [59], and our prior work [39] (Prior) as the baselines.

HWF *Hand-Written Formula Parsing and Evaluation.* HWF, proposed in [51], 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 [51] 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 [92], 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 [34], S4* [35], and SGConv [53].

PacMan-Maze *Playing PacMan Maze Game.* 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 $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$ 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 [85], 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 [56], BiLSTM [32], GPT-3-FT (fine-tuned), GPT-3-ZS (zero-shot), and GPT-3-FS (5-shot) [8]. 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 [96] and CTP [63] as baselines.

Mugen *Video-Text Alignment and Retrieval.* Mugen [37] is based on a game called CoinRun [16]. 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 [37].

CLEVR *Compositional Language and Elementary Visual Reasoning [42].* 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 [103] and NSCL [60], 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 [41]; 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) [27] containing triplets such as (giraffe, is-a, animal) for common-sense reasoning. The baselines for this task are NMNs [4] and LXMERT [91].

5.1.2 RQ1: Our Solutions and Expressivity

To answer **RQ1**, we demonstrate our Scallop solutions to the benchmark tasks (Table 5.1). 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 Scallop program for benchmark tasks are succinct, as indicated by the LoCs in the last column of Table 5.1. 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,

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 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 composing kinship 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 in 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 with the aid of common-sense knowledge base (KB).	✔			42

	KB*	–	`is_a(`*name1*, *name2*`)`, ...					

Table 5.1: Characteristics of Scallop solutions for each task. Structured input which is not learnt is denoted by *. Neural models used are RoBERTa [56], DistilBERT [80], and BiLSTM [32] for natural language (NL), CNN and FastRCNN [29] for images, and S3D [100] 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.

the Scallop program is an interpreter for CLEVR-DSL, a domain-specific functional language introduced in the CLEVR dataset [42].

We note that our prior work [39], 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 5.5, 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 2.2) 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. 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.

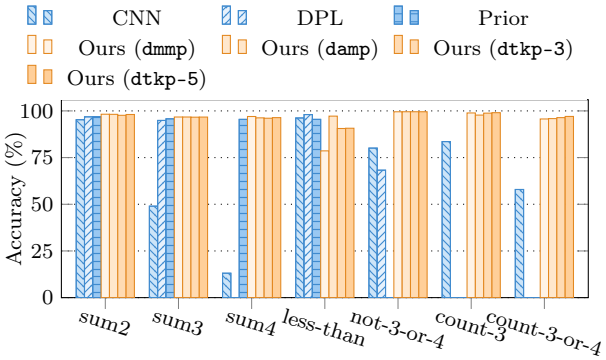


Figure 5.2: MNIST-R suite accuracy comparison.

Table 5.2: PacMan-Maze.

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

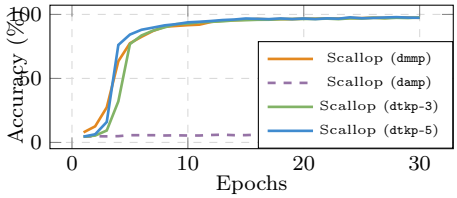


Figure 5.3: HWF learning curve.

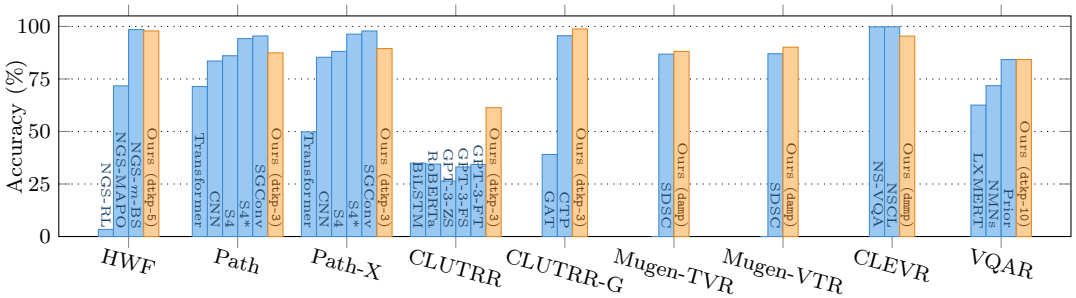


Figure 5.4: Overall benchmark accuracy comparison. The best-performing provenance structure for our solution is indicated for each task. Among the shown tasks, `dtkp` performs the best on 6 tasks, `damp` on 2, and `dmmp` on 1.

5.1.3 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 Figure 5.2, Table 5.2, and Figure 5.4. 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

Table 5.3: 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	–

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 5.2 and Figs. 5.2–5.4, 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 (MNIST-R) and Mugen, and `dmmp` for HWF and CLEVR. Note that under positive Datalog, Scallop's `dtkp` is identical to [39], allowing us to achieve similar performance. In conclusion, allowing configurable provenance helps tailor our methods to different applications.

5.1.4 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 5.3, 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 Figure 5.2). 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.

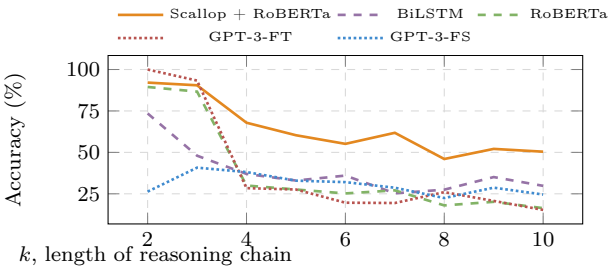


Figure 5.6: Systematic generalizability on CLUTRR dataset.

Figure 5.7: The predicted most likely (action, mod) pair for example video segments from Mugen dataset.

5.1.5 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.

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 Figure 5.6, 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. Figure 5.7 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 5.4). 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-

Table 5.4: 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

Task	Dataset	#Test Samples	Metric	Foundation Models Used
DR	DR	369	EM	GPT-4
TSO	TSO	150	EM	GPT-4
KR	CLUTRR	1146	EM	GPT-4
MR	GSM8K	1319	EM	GPT-4
QA	Hotpot QA	1000	EM	GPT-4
				ada-002
PS	Amazon ESCI	1000	nDCG	GPT-4
				ada-002
VQA	CLEVR	480	Recall@1 Recall@3	GPT-4
				OWL-ViT
	GQA	500		VilT
				CLIP
VOT	VQAR	100	MI	OWL-ViT
				VilT
				GPT-4
	OFCP	50		DSFD
				CLIP
IGE	OFCP	50	MI	DFSD
				CLIP
	IGP20	20		GPT-4
				Diffusion

Table 5.5: Characteristics of benchmark tasks including the dataset used, its size, and evaluation metrics. Metrics include exact match (EM), normalized discounted cumulative gain (nDCG), and manual inspection (MI). We also denote the foundation models used in our solution for each task.

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.

5.2 Scallop Benchmarks with Foundation Models

We apply Scallop to solve 9 benchmark tasks depicted in Figure 5.8 with foundation models. Table 5.5 summarizes the datasets, evaluation metrics, and the foundation models used in our solutions. We elaborate upon the evaluation settings and our solutions below.

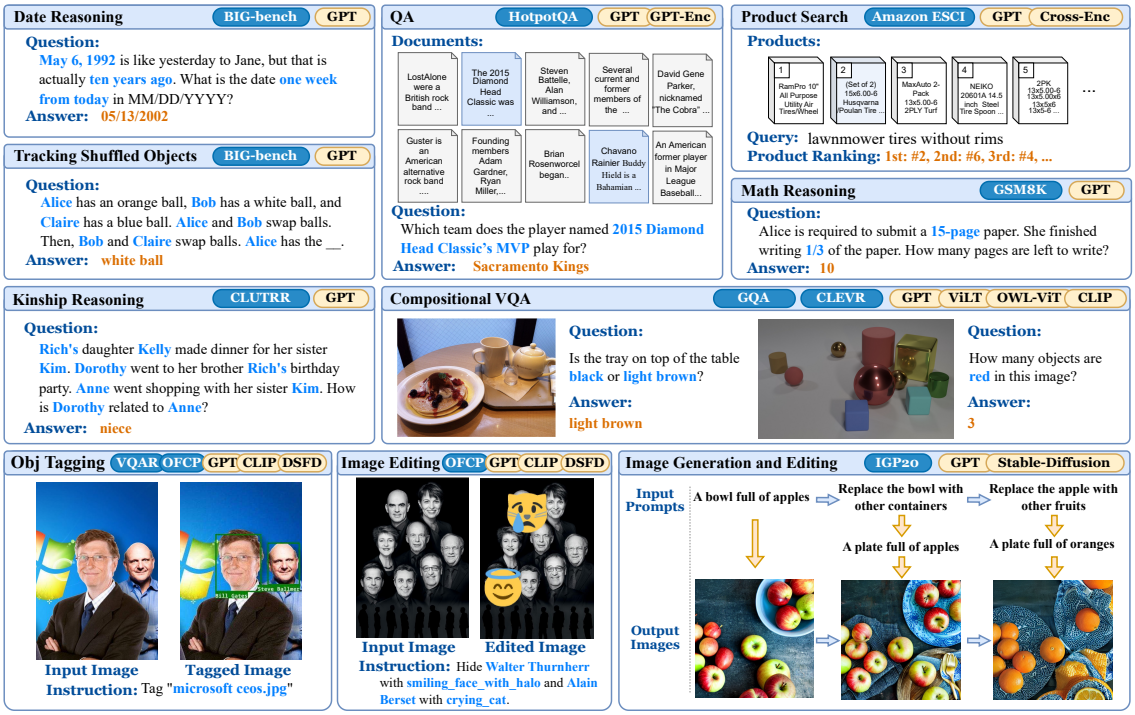


Figure 5.8: Benchmark tasks. The top of each box lists the dataset(s) and the foundation models used in our solutions.

5.2.1 Benchmarks

Date reasoning (DR). In this task adapted from BIG-bench [89], the model is given a context and asked to compute a date. The questions test the model's temporal and numerical reasoning skills, as well as its grasp of common knowledge. Unlike BIG-bench where multiple-choice answers are given, we require the model to directly produce its answer in MM/DD/YYYY form.

Our solution leverages GPT-4 (5-shot[1]) for extracting 3 relations: mentioned dates, duration between date labels, and the target date label. From here, our relational program iterates through durations to compute dates for all date labels. Lastly, the date of the target label is returned as the output.

Tracking shuffled objects (TSO). In this task from BIG-bench, a textual description of pairwise object swaps among people is given, and the model needs to track and derive which object is in a specified person's possession at the end. There are three difficulty levels depending on the number of objects to track, denoted by $n \in \{3, 5, 7\}$.

[1]In this work, k in "k-shot" means the number of examples provided to the LM component within the full solution. Each example is a ground-truth input-output pair for the LM.

Our solution for tracking shuffled objects relies on GPT-4 (1-shot) to extract 3 relations: initial possessions, swaps, and the target person whose final possessed object is expected as the answer. Our reasoning program iterates through all the swaps starting from the initial state and retrieves the last possessed object associated with the target.

Kinship reasoning (KR). CLUTRR [85] is a kinship reasoning dataset of stories which indicate the kinship between characters, and requires the model to infer the relationship between two specified characters. The questions have different difficulty levels based on the length of the reasoning chain, denoted by $k \in \{2 \dots 10\}$.

Our solution for kinship reasoning invokes GPT-4 (2-shot) to extract the kinship graph from the context. We also provide an external common-sense knowledge base for rules like "mother's mother is grandmother". Our program then uses the rules to derive other kinship relations. Lastly, we retrieve the kinship between the specified pair of people.

Math reasoning (MR). This task is drawn from the GSM8K dataset of arithmetic word problems [17]. The questions involve grade school math word problems created by human problem writers, and the model is asked to produce a number as the result. Since the output can be fractional, we allow a small delta when comparing the derived result with the ground truth.

Our solution to this task prompts GPT-4 (2-shot) to produce step-by-step expressions, which can contain constants, variables, and simple arithmetic operations. We evaluate all the expressions through a DSL, and the result associated with the goal variable is returned. By focusing the LM's responsibility solely on semantic parsing, our relational program can then achieve faithful numerical computation via DSL evaluation.

Question answering with information retrieval (QA). We choose HotpotQA [102], a Wikipedia-based question answering (QA) dataset under the "distractor" setting. Here, the model takes in 2 parts of inputs: 1) a question, and 2) 10 Wikipedia paragraphs as the context for answering the question. Among the 10 Wikipedia pages, at most 2 are relevant to the answer, while the others are distractors.

Our solution is an adaptation of FE2H [52], which is a 2-stage procedure. First, we turn the 10 documents into a vector database by embedding each document. We then use the embedding of the question to retrieve the 2 most related documents, which are then fed to a language model

to do QA. In this case, the QA model does not have to process all 10 documents, leading to less distraction.

Product search (PS). We use Amazon's ESCI Product Search dataset [77]. The model is provided with a natural language (NL) query and a list of products (23 products on average). The goal is to rank the products that best match the query. In the dataset, for each pair of query and product, a label among E (exact match), S (substitute), C (complementary), and I (irrelevant) is provided. The metric we use to evaluate the performance is nDCG. The gains are set to be 1.0 for E, 0.1 for S, 0.01 for C, and 0.0 for I.

One challenge of this dataset is that many queries contain negative statements. For example, in the query "#1 treadmill without remote", the "remote" is undesirable. Therefore, instead of computing the embedding of the full query, we decompose the query into positive and negative parts. We then perform semantic search by maximizing the similarity of the positive part while minimizing that of the negative part.

Compositional visual question answering (VQA). We choose two compositional VQA datasets, GQA [40] and CLEVR [43]. In this task, the model is given an image and a question, and needs to answer the question. For GQA, the majority of questions expect yes/no answers, while CLEVR's questions demand features like counting and spatial reasoning. We uniformly sample 500 and 480 examples from GQA and CLEVR datasets respectively. Following VQA conventions [45], we use Recall@k where $k \in \{1, 3\}$ as the evaluation metrics.

Our solution for GQA is an adaptation of VisProg [36]. We create a DSL for invoking vision modules such as ViLT and OWL-ViT, and use GPT-4 for converting questions into programs in this DSL. Our solution for CLEVR is similar, directly replicating the DSL provided by the original work. OWL-ViT and CLIP are used to detect objects and infer attributes, while the spatial relations are directly computed using the bounding box data.

Visual object tagging (VOT). We evaluate on two datasets, VQAR [39] and OFCP. For VQAR, the model is given an image and a programmatic query, and is asked to produce bounding boxes of the queried objects in the image. Our solution composes a relational knowledge base, defining entity names and relationships, with object retrieval (OWL-ViT) and visual QA (ViLT) models.

Dataset	LoC	Prompt LoC	Dataset	LoC	Prompt LoC
DR	69	48	CLEVR	178	45
TSO	34	16	GQA	82	36
CLUTRR	61	45	VQAR	53	11
GSM8K	47	28	OFCP (VOT)	33	2
HotpotQA	47	24	OFCP (IGE)	117	44
ESCI	32	7	IGP20	50	12

Table 5.6: The lines-of-code (LoC) numbers of our solutions for each dataset. The LoC includes empty lines, comments, natural language prompts, and DSL definitions. We note specifically the LoC of prompts in the table.

Online Faces of Celebrities and Politicians (OFCP) is a self-curated dataset of images from Wikimedia Commons among other sources. For this dataset, the model is given an image with a descriptive NL filename, and needs to detect faces relevant to the description and tag them with their names. Our solution obtains a set of possible names from GPT-4 and candidate faces from DSFD. These are provided to CLIP for object classification, after which probabilistic reasoning filters the most relevant face-name pairs.

Language-guided image generation and editing (IGE). We adopt the task of image editing from [36]. In this task, the instruction for image editing is provided through NL, and can invoke operations such as blurring background, popping color, and overlaying emojis. Due to the absence of an existing dataset, we repurpose the OFCP dataset by introducing 50 NL image editing prompts. Our solution for this task is centered around a DSL for image editing. We incorporate GPT-4 for semantic parsing, DSFD for face detection, and CLIP for entity classification. Modules for image editing operations are implemented as individual foreign functions.

For free-form generation and editing of images, we curate IGP20, a set of 20 prompts for image generation and editing. Instead of using the full prompt, we employ an LM to decompose complex NL instructions into simpler steps. We define a DSL with high-level operators such as generate, reweight, refine, replace, and negate. We use a combination of GPT-4, Prompt-to-Prompt [38], and diffusion model [78] to implement the semantics of our DSL. We highlight our capability of grounding positive terms from negative phrases, which enables handling prompts like "replace apple with other fruits" (Figure 5.8).

5.2.2 Experiments and Analysis

We aim to answer the following research questions:

Method	DR	TSO	CLUTRR	GSM8K
GPT-4	71.00 (0-shot)	30.00 (0-shot)	43.10 (3-shot)	87.10 (0-shot)
GPT-4 (CoT)	87.26 (0-shot)	84.00 (0-shot)	24.17 (3-shot)	**92.00** (5-shot)
Ours	**92.41**	**100.00**	**72.50**	90.60

Table 5.7: The performance on the natural language reasoning datasets. Numbers are in percentage (%).

HotpotQA			Amazon ESCI		
Method	Fine-tuned	EM	Method	Fine-tuned	nDCG
C2FM	✓	72.07%	BERT	✓	0.830
FE2H	✓	71.89%	CE-MPNet	✓	0.857
—	—	—	MIPS	✗	0.797
Ours	✗	67.3%	Ours	✗	0.798

Table 5.8: The performance on the HotpotQA and Amazon ESCI. We also include performance numbers from methods which are fine-tuned on the corresponding dataset.

RQ1. Is Scallop with foundation model programmable enough to be applicable to a diverse range of applications with minimal effort?

RQ2. How do solutions using Scallop compare to other baseline methods in the no-training setting?

RQ1: Programmability

While a user study for Scallop's programmability is out of scope in this paper, we qualitatively evaluate its programmability on three aspects. First, we summarize the lines-of-code (LoC) for each of our solutions in Table 5.6. The programs are concise, as most are under 100 lines. Notably, natural language prompts (including few-shot examples) take up a significant portion of each solution. Secondly, 8 out of 10 solutions are coded by undergraduate students with no background in logic and relational programming, providing further evidence of Scallop's user-friendliness. Last but not least, our solutions are interpretable and thus offer debuggability. Specifically, all the intermediate relations are available for inspection, allowing systematic error analysis.

RQ2: Baselines and Comparisons

We compare the performance of our solutions to existing baselines under the no-training setting. In particular, our solutions achieve better performance than comparable baselines on 6 out of 8

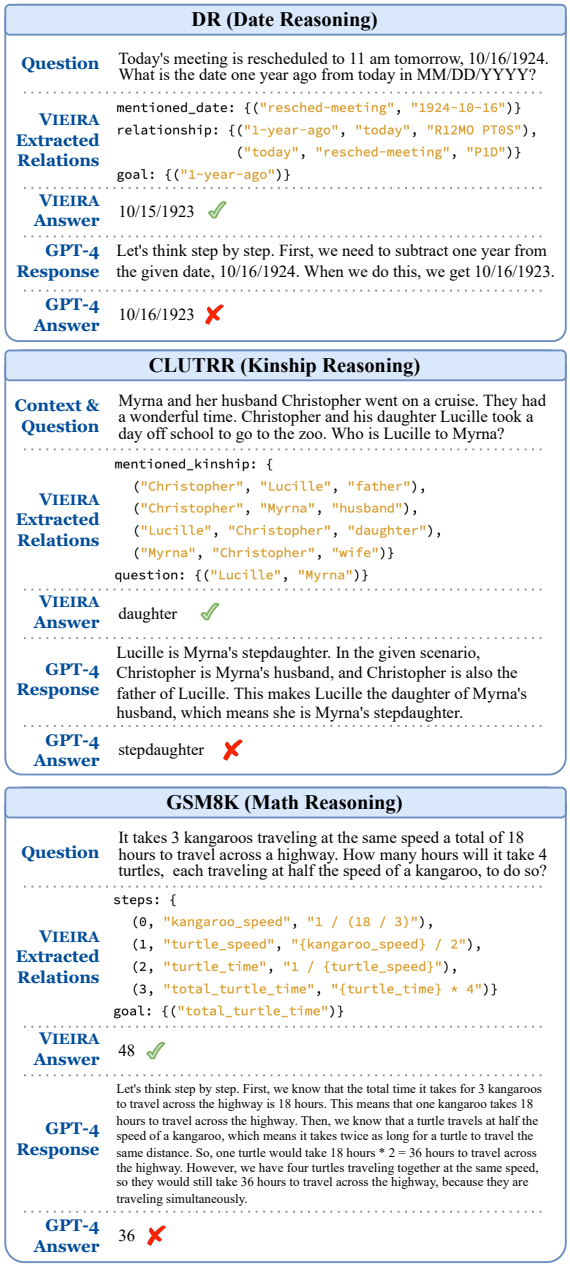
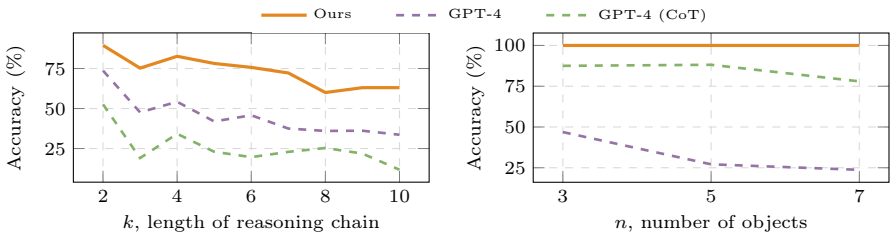


Figure 5.9: Illustrative comparisons between our solution and GPT-4 (zero-shot CoT) on selected questions from DR, CLUTRR, and GSM8K datasets. We also include the extracted relations used for subsequent reasoning.

studied datasets with baselines. Below, we classify the tasks into 4 categories and discuss the respective performance and comparisons.

Natural language reasoning. For the tasks of DR, TSO, CLUTRR, and GSM8K, we pick a generic baseline of GPT-4 under zero-shot, few-shot, and chain-of-thought (CoT) settings. All



(a) CLUTRR (b) TSO

Figure 5.10: Systematic generalizability comparisons on the CLUTRR and TSO datasets.

Method	GQA		CLEVR	
	Recall@1	Recall@3	Recall@1	Recall@3
ViLT-VQA	0.049	0.462	0.241	0.523
PNP-VQA	0.419	—	—	—
Ours	**0.579**	**0.665**	**0.463**	**0.638**

Table 5.9: Quantitative results on the VQA datasets.

our solutions also rely on GPT-4 (few-shot), but we note that our shots only include extracted facts, and not the final answer or any reasoning chains. The data in Table 5.7 indicates that our method can significantly enhance reasoning performance and reduce hallucination, exemplified by achieving a flawless 100% accuracy on the TSO dataset. Note that on GSM8K, our method scores slightly lower than the baseline; we conjecture that our solution demands more from GPT-4 itself to extract structured computation steps. On CLUTRR, our solution even outperforms fCoT [57], a special prompting technique with external tool use, by 0.6%. In Fig. 5.10 we illustrate the systematic generalizability of our methods. The performance of our solutions remains relatively consistent even when the problems become harder. We provide illustrative examples in Fig. 5.9 showing comparisons between our method and GPT-4 (zero-shot CoT).

Retrieval augmentation and semantic search. For the HotpotQA dataset, our solution is an adaptation of FE2H [52], a retrieval-augmented question answering approach. As seen in Table 5.8, with no fine-tuning, our method scores only a few percentages lower than fine-tuned methods C2FM [105] and FE2H. For the Amazon ESCI dataset, our solution performs semantic search for product ranking. While performing slightly lower than the fine-tuned methods [77, 88], our solution outperforms maximum inner product search (MIPS) based on GPT text encoder (`text-embedding-ada-002`).


```
Instruction: Replace the bowl with something
else, and change the apples to other fruits.
```

Figure 5.11: Qualitative comparison of image editing. Compared to InstructPix2Pix, our image editing method follows the instructed edits better, as it successfully changed the bowl into plate and apples to oranges.

Method	**Visual Object Tagging**		**Image Editing**
	VQAR	OFCP	OFCP
Ours	67.61%	60.82%	74.00%

Table 5.10: Quantitative results on object tagging and image editing tasks. We manually evaluate the tagged entities and the edited images for semantic correctness rates.

Compositional multi-modal reasoning. For VQA, we pick ViLT-VQA [45] (a pre-trained foundation model) and PNP-VQA [94] (a zero-shot VQA method) as baselines. As shown in Table 5.9, our method significantly outperforms the baseline model on both datasets. Compared to the neural-only baseline, our approach that combines DSL and logical reasoning more effectively handles intricate logical operations such as counting and numerical comparisons. On GQA, out method outperforms previous zero-shot state-of-the-art, PNP-VQA, by 0.16 (0.42 to 0.58). For object and face tagging, without training or fine-tuning, our method achieves 67.61% and 60.82% semantic correctness rates (Table 5.10).

Image generation and editing. For image generation and editing, we apply our technique to the OFCP and IGP20 datasets. We rely on manual inspection for evaluating our performance on the OFCP dataset, and we observe 37 correctly edited images out of the 50 evaluated ones, resulting in a 74% semantic correctness rate (Table 5.10). For IGP20, we choose as the baseline a diffusion model, InstructPix2Pix [7], which also combines GPT-3 with image editing. We show one example baseline comparison illustrated in Figure 5.11.

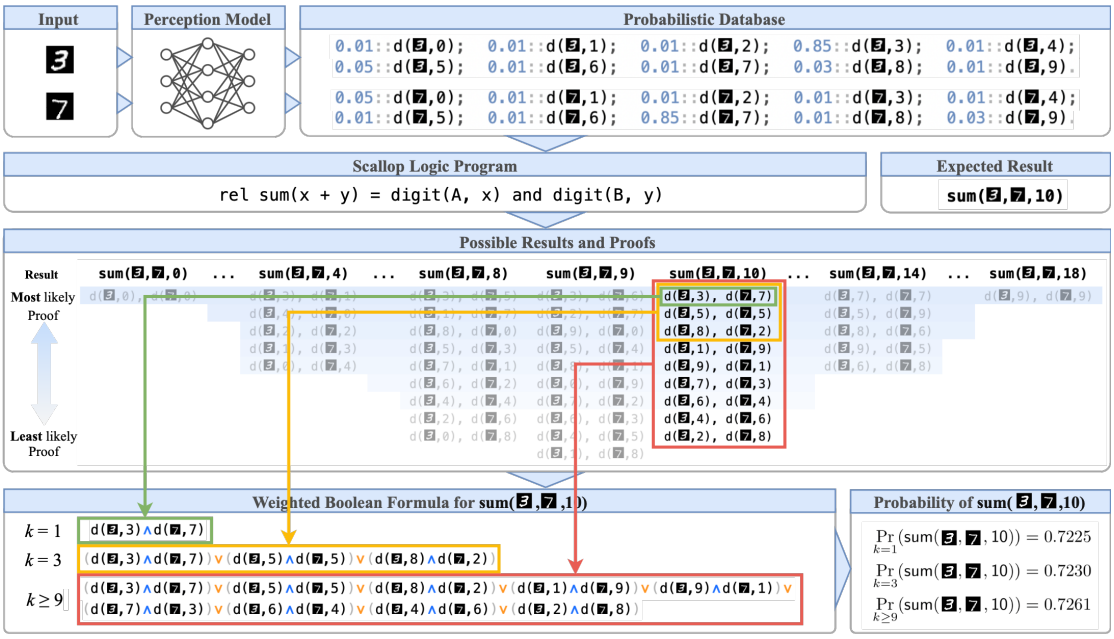


Figure 5.12: Illustration of applying Scallop's top-k-proofs provenance on the task 3 + 7 = 10 using different values of parameter k.

5.3 Case Study: Summing Two MNIST Digits

The `MNIST-Sum2` task from [59] concerns classifying sums from pairs of hand-written digits, e.g., 3 + 7 = 10. A model receives only the two MNIST digits as the input, and need to learn to recognize the two digits with only the supervision of the sum.

As depicted in Figure 5.12, we specify this task using a neural and a symbolic component, following the style of DeepProbLog [59]. The neural component is a perception model that takes in an image of hand-written digit [48] and classifies it into discrete values $\{0, \ldots, 9\}$. The symbolic component, on the other hand, is a logic program in Datalog for computing the resulting sum. The interface between the neural and symbolic components is a probabilistic database which associates each candidate output of the perception model with a probability. For instance, the fact 0.85 :: d(3, 3) denotes that image 3 is recognized to be the digit 3 with probability 0.85. The database thus consists of 20 facts—one for each of the 10 possible digits corresponding to each of the two images.

Evaluating the logic program on the probabilistic database yields a weighted boolean formula for each possible result of the sum of two digits, i.e., values in the range $\{0, \ldots, 18\}$. Each *clause* of such a formula represents a different *proof* of the corresponding result. For instance, the bottom left

```
1  class MNISTSum2Net(nn.Module):
2    def __init__(self):
3      super(MNISTSum2Net, self).__init__()
4      self.mnist_net = MNISTNet() # MNIST Digit Recognition
5      self.sum_2 = scallopy.Module( # Scallop Module
6        program="""
7          type digit_1(i32), digit_2(i32)
8          rel sum(a + b) = digit_1(a) and digit_2(b)
9        """,
10       input_mappings={"digit_1": range(10),
11                       "digit_2": range(10)},
12       output_mappings={"sum_2": range(19)},
13       provenance="diff-top-k-proofs", k=1)
14
15   def forward(self, x: Tuple[torch.Tensor, torch.Tensor]):
16     (a_imgs, b_imgs) = x # batch_size x 27 x 27 x 1
17     # recognize the two digits
18     a_distrs = self.mnist_net(a_imgs) # batch_size x 10
19     b_distrs = self.mnist_net(b_imgs) # batch_size x 10
20     # perform reasoning; result shape is batched size x 19
21     return self.sum_2(digit_1=a_distrs, digit_2=b_distrs)
```

Listing 5.1: The Scallop code for the `MNIST-Sum2` learning task.

of Figure 5.12 shows the formula representing all 9 proofs of the ground truth result 10. Each such formula is input to an off-the-shelf weighted model counting (WMC) solver to yield the probability of the corresponding result, e.g., $0.7261 :: \texttt{sum}(10)$.

The scalability of exact differentiable probabilistic reasoning is limited in practice by WMC solving whose complexity is at least #P-hard. As suggested by the discussion of top-k proofs provenance, computing only the top-k most likely proofs bounds the size of each formula to k clauses, thereby allowing to trade diminishing amounts of accuracy for large gains in scalability. Moreover, stochastic training of the deep perception models itself can tolerate noise in data. In this case, just using $k = 1$ yields a performance of 97.46%, which turns out to be empirically the best across $k \in \{1, 3, 5, 10\}$.

The actual implementation is depicted in Listing 5.1. The `mnist_net` (line 4) is the neural network that classifies individual MNIST image into a distribution of 10 classes, while `sum_2` (line 5-13) is the Scallop reasoning module for probabilistic reasoning of summing two digits. Instead of writing a separate Scallop file that contains the reasoning program, we passed the program as a string to construct the `scallopy.Module`. We note that for cleaner presentation, the program presented here is slightly different than the one in Figure 5.12. Line 10-12 tells how to turn input distributions into relational symbols (and vice versa for the output). Line 13 configures the provenance to use for the reasoning, which is the `dtkp` provenance with $k = 1$.

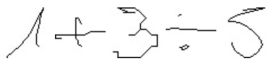


Figure 5.13: One hand-written formula $1 + 3 \div 5$ which should evaluate to 1.6.

During inference, as shown in the `forward` function (line 15-21), we pass the two (batches of) images to `mnist_net` individually to produce distributions of the two (batches of) images. Lastly, we pass the two batches of distributions to the `sum_2` reasoning module in order to obtain a batched output tensor of shape 19, where each element correspond to one of the 19 outcomes ($[0, 18]$). As such, our training pipeline is finished. All the algorithmic and differentiation detail of Scallop is hidden from the user, providing a clean programming interface.

5.4 Case Study: Evaluating Handwritten Formulas

In this case study we take the `MNIST-Sum2` one step further by allowing multiple symbols including hand-written digits and also simple arithmetic operators like $+$, $-$, $\times$, and $\div$. This is the task of hand-written formula evaluation (HWF) [51]. The input to the task is a sequence of images of hand-written symbols, forming a hand-written formula. The output is the rational number result of evaluating the formula. The dataset provided for this task contains variable-sized formulas with 1 to 7 symbols, where the operands are all single-digit numbers. For brevity of exposition, we presume that the input formulas always parse and are free of divide-by-zero errors.

A natural solution to this task is to decompose the problem into separate perception and reasoning components. The perception component is a standard convolutional neural network (CNN) that classifies each symbol into discrete classes (digits 0-9 and $+$, $-$, $\times$, $\div$). The reasoning component then takes in the classified probabilistic symbols, parses and evaluates the formula, and returns a probability distribution of the result. Notably, the neural model does not receive supervision on the label of each individual symbol in the formula. Instead, we only have supervision on the final evaluation result. Scallop's differentiable reasoning engine enables to train the resulting program in an end-to-end fashion, that is, to learn the parameters of the neural model using only supervision on observable input-output data—called *algorithmic supervision* [72].

The reasoning component is written in Scallop as shown in Listing 5.2. The program uses Datalog-like syntax. It specifies two input relations, `symbol` and `length` (line 2-3). The former

```
// [hwf.scl]
// Input: probabilistic symbols
type symbol(index: usize, symbol: String)
// Input: length of the formula
type length(n: usize)

// Helper relation
rel digit = {"0","1","2","3","4","5","6","7","8","9"}

// Parsing and evaluating the sequence of symbols
// A single number
type factor(value: f32, begin: usize, end: usize)
rel factor(x as f32, b, b + 1) = symbol(b, x) and digit(x)

// A mult/div expression
type term(value: f32, begin: usize, end: usize)
rel term(x, b, r) = factor(x, b, r)
rel term(x * y, b, e) = term(x, b, m) and symbol(m, "*") and factor(y, m + 1, e)
rel term(x / y, b, e) = term(x, b, m) and symbol(m, "/") and factor(y, m + 1, e)

// An add/minus expression which has higher precedence
type expr(value: f32, begin: usize, end: usize)
rel expr(x, b, r) = term(x, b, r)
rel expr(x + y, b, e) = expr(x, b, m) and symbol(m, "+") and term(y, m + 1, e)
rel expr(x - y, b, e) = expr(x, b, m) and symbol(m, "-") and term(y, m + 1, e)

// Obtain the result
rel result(y) = expr(y, 0, l) and length(l)
```

Listing 5.2: Formula evaluator for the HWF task in Scallop.

```
class HWFNet(nn.Module):
  def __init__(self):
    MAX_LEN = 7
    ALPHABET = ["0", ..., "9", "+", "-", "*", "/"]
    # other setup code...
    self.symbol_cnn = SymbolNet() # Symbol recognition
    # Scallop module for formula evaluation
    self.eval_formula = scallopy.Module(
      file="hwf.scl", provenance="diff-top-k-proofs", k=3,
      input_mappings={"symbol": scallopy.InputMapping(
        {0: range(MAX_LEN), 1: ALPHABET},
        retain_k=3, sample_dim=1)},
      output="result", non_probabilistic=["length"])

  def forward(self, img_seq, img_seq_len):
    length = [[(l.item(),)] for l in img_seq_len]
    # First recognize the symbols
    symbol = self.symbol_cnn(img_seq.flatten(0, 1)).view(len(length), -1)
    # Then evaluate the formula with Scallop
    (out_symbols, out_distr) = self.eval_formula(symbol=symbol, length=length)
    return (out_symbols, out_distr)
```

Listing 5.3: PyTorch module for the HWF task with Scallop.

relates each symbol image's index with its recognized symbol (digits and operators represented as strings), and the latter encodes the length of the formula. The rest of the program defines relations `factor` (lines 9-10), `term` (lines 12-15), and `expr` (lines 17-20), going up the standard context-free grammar of simple arithmetic expressions. The first argument of each of these relations is a floating

point number, with type `f32`, denoting the evaluated results of the corresponding expressions. Lastly, we fetch the expression which covers the whole formula (line 23), and store the evaluated result in the `result` unary relation.

Next, we may integrate this program into an end-to-end learning pipeline. Listing 5.3 shows the PyTorch module for the HWF task. During initialization, we setup the CNN to process each symbol image (line 6). Then we create a Scallop module to load the program from file `hwf.scl` (line 8-13). We also configure the provenance semiring to be used as `diff-top-k-proofs` with `k` set to 3. During the training or inference phase, we simply pass the symbol images to the CNN (line 17) and the result distributions to our Scallop module (line 19). Since both the CNN and the Scallop module are differentiable, we obtain an end-to-end learning pipeline. While being conceptually similar, there are a few core complexities of HWF when compared to `MNIST-Sum2`. We now explain each of them and how the Python interface helps to ease the handling of such complexities.

Varying number of inputs. First, instead of taking in a constant number of 2 digits, HWF reasoning module needs to accept formulas of varying lengths. In PyTorch, such information is encoded in 2 dimensional tensors, where the first dimension encodes the index of each symbol, and the second dimension encodes the distribution over our alphabet. For the formulas not of the maximum lengths, the tensor contains padded 0-s. To process this tensor, we setup the `input_mapping` (line 10-12) for the `symbol` relation to be a 2-dimensional mapping. The first dimension (dim 0) is mapped to `range(MAX_LEN)`, which is $0, \ldots, 6$ given that the maximum length of formula in the dataset is 7. The second dimension maps each element to one symbol inside our `ALPHABET`. For `length`, we specify that it is non-probabilistic (line 13).

The need for symbol sampling. The input space for HWF input is huge, since there could be 7 symbols with each being one of 14 classes, giving us roughly 14^7 possible derivation trees. If nothing else is done, Scallop would explore all of the derivation trees, which will be prohibitively slow. Therefore, instead of passing every single fact to Scallop, we perform sampling based on the predicted probabilities. During the configuration of `symbol`'s input mapping, the two arguments `retain_k=3` and `sample_dim=1` specify that on the `symbol` tensor, we only pick the top 3 classes for each symbol (on dimension 1). With these arguments, we are able to make the inference process more scalable. We note that for HWF, sampling only 3 reaches a good balance between training time and learning accuracy. But in general, the lower the sample rate, the longer it will take to

```
# calling hwf_net yields the set of outputs as well as the
# predicted distributions over the set of outputs
(outputs, y_pred) = hwf_net(formula_imgs)

# construct a ground truth tensor y based on the labels and
# the set of produced outputs
y = torch.tensor([
  [1.0 if abs(l - m) < 0.001 else 0.0 for m in outputs]
  for l in labels])

# compute the binary cross entropy loss
loss = binary_cross_entropy(y_pred, y)
```

Listing 5.4: The loss function used for HWF. Before applying the binary cross-entropy loss, we also use the derived `outputs` to construct one-hot vectors as the ground-truth. Notice that since HWF deals with fraction numbers, we cannot use exact comparison of derived number with the ground truth label. Instead, we apply `abs(l - m) < 0.001` to allow for floating point errors during derivation.

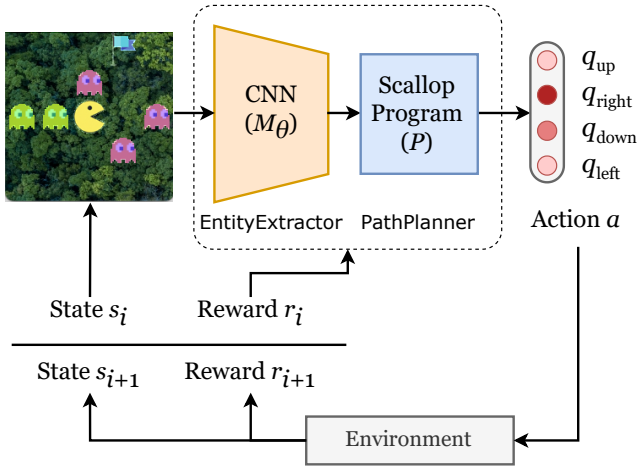
train the model end-to-end.

Unbounded set of outputs. instead of a fixed set of possible outputs ($\{0, \ldots, 18\}$) in `MNIST-Sum2`, HWF has a much larger set of potential outputs, due to the fact that formulas contain division operator ($\div$). It is not realistic to enumerate all of them, which is why we do not explicitly specify an output mapping. The outcome of this is that the `eval_formula` cannot return a straightforward vectorized tensor as the output. As shown on line 19, we obtain two results, `out_symbols` and `out_distr`. Specifically, `out_symbols` will be a list of fraction numbers that are actually derived with the sampled inputs. Meanwhile, `out_distr` will contain computed distribution over the results in `out_symbols`. We present in Listing 5.4 the loss function that is used to process such output.

5.5 Case Study: Playing the PacMan-Maze Game

We further illustrate Scallop using an reinforcement learning (RL) based planning application which we call PacMan-Maze. The application, depicted in Figure 5.14a, 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 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.

(a) Three states of one gameplay session.



(b) Architecture of application with Scallop.

Figure 5.14: Illustration of a planning application PacMan-Maze in Scallop.

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 Figure 5.14b. 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 [98]. 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.[2] Since the game states are images, we employ Deep Q-Learning [64], which approximates the Q function using a convolutional neural network (CNN) model with learned parameter θ. An end-to-end deep learning

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

```
1  class PacManAgent(torch.nn.Module):
2    def __init__(self, grid_dim, cell_size):
3    # initializations...
4    self.extract_entities =
5      EntityExtractor(grid_dim, cell_size)
6    self.path_planner = ScallopModule(
7      file="path_planner.scl",
8      provenance="diff-top-k-proofs", k=1,
9      input_mappings={"actor": cells,
10     "goal": cells, "enemy": cells},
11     output_mappings={"next_action": actions})
12
13   def forward(self, game_state_image):
14   actor, goal, enemy =
15     self.extract_entities(game_state_image)
16   next_action = self.path_planner(
17     actor=actor, goal=goal, enemy=enemy)
18   return next_action
```

Listing 5.5: Snippet of implementation in Python.

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 Figure 5.14b. 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 to the goal to determine the optimal next action. Figure 5.5 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. A snippet of the overall Scallop application in Python is shown in Figure 5.5. 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 Figure 5.6. We next illustrate three key design decisions of Scallop with respect to this program.

```
1  // [path_planner.scl]
2  // The set of possible actions to take at each state
3  type Action = UP | DOWN | RIGHT | LEFT
4
5  // The input relations from neural networks
6  type grid_cell(x: i32, y: i32), actor(x: i32, y: i32),
7       goal(x: i32, y: i32), enemy(x: i32, y: i32)
8
9  // Reasoning rules...
10 rel safe_cell(x, y) = grid_cell(x, y) and not enemy(x, y)
11 rel edge(x, y, x, yp, UP) = safe_cell(x, y) and safe_cell(x, yp), yp == y + 1
12 // Rules for DOWN, RIGHT, and LEFT edges are omitted...
13
14 rel next_pos(p, q, a) = actor(x, y), edge(x, y, p, q, a)
15 rel path(x, y, x, y) = next_pos(x, y, _)
16 rel path(x1, y1, x3, y3) = path(x1, y1, x2, y2) and edge(x2, y2, x3, y3, _)
17 rel next_action(a) = next_pos(p, q, a), path(p, q, r, s), goal(r, s)
```

Listing 5.6: The logic program of the PacMan-Maze application in Scallop.

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 Listing 5.5). 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 Listing 5.5), 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 illustrate here using the program in Listing 5.6. The program realizes the symbolic component of our example using a set of logic rules. Instead of having to explicitly encode a searching algorithm for path-finding, the logic can be declaratively specified in Scallop, simplifying the programming experience from an end-user point-of-view.

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.[3]

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, as achieved by the differentiable provenances implemented within our provenance framework. In our example, line 8 in Figure 5.5 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.

5.6 Case Study: Learning Composition Rules for Kinship Reasoning

CLUTRR [85] consists of kinship reasoning questions. Given a context that describes a family's routine activity, the goal is to deduce the relationship between two family members that is not

[3]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.

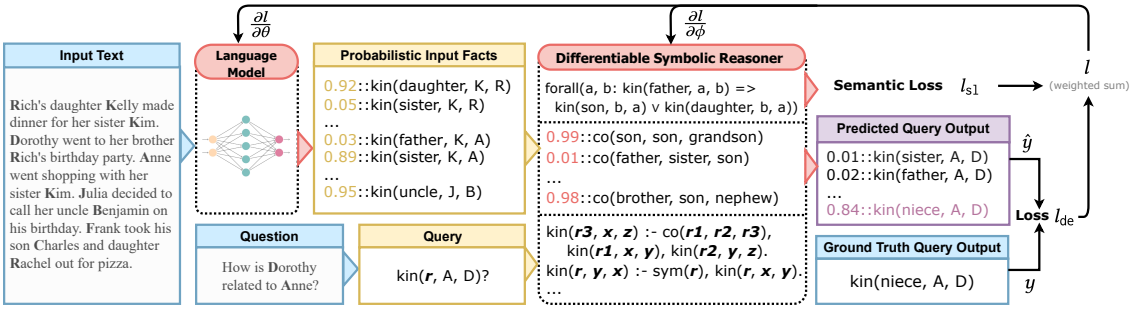


Figure 5.15: Overview of kinship reasoning with an example where "Anne is the niece of Dorothy" can be inferred from the context. We abbreviate the names with their first initials in the relational symbols, and the composite relationship with "co".

explicitly mentioned in the story.

We showcase one CLUTRR example in Figure 5.15. The input text is "Rich's daughter Kelly made dinner for her sister Kim. Dorothy went to her brother Rich's birthday party. Anne went shopping with her sister Kim. " From this narrative, we infer several relationships: Rich is Dorothy's brother, Kelly is Rich's daughter, Kim is Kelly's sister, and Anne is Kim's sister. Leveraging our common sense knowledge, we understand that one's sister's sister is also her sister, a sister's father is her father, and a brother's daughter is his niece. Consequently, we deduce that Anne is Kelly's sister, making Rich Anne's father, and Dorothy, Anne's aunt.

The family kinship graph of the CLUTRR dataset is synthetic and the names of the family members are randomized. However, the sentences included in the story are crowd-sourced and hence there is a considerable amount of naturalness inside the dataset. The CLUTRR dataset is further divided into different difficulties measured by k, the number of facts used in the reasoning chain. For training, we only use 10K data points with 5K $k = 2$ and another 5K $k = 3$, meaning that we can only receive supervision on data with short reasoning chains. The test set, on the other hand, contains 1.1K examples with $k \in \{2, \ldots, 10\}$.

5.6.1 Structured Representation: Family Graph

One natural representation of the family relationship is the family graph, as shown in Figure 5.16. The nodes in the family graph represents the family members, and the edges represents the relationship between the connected two family members. We can thus express the family graph in the form of facts.

Logic rules can be applied to known facts to deduce new ones. For example, below is a horn

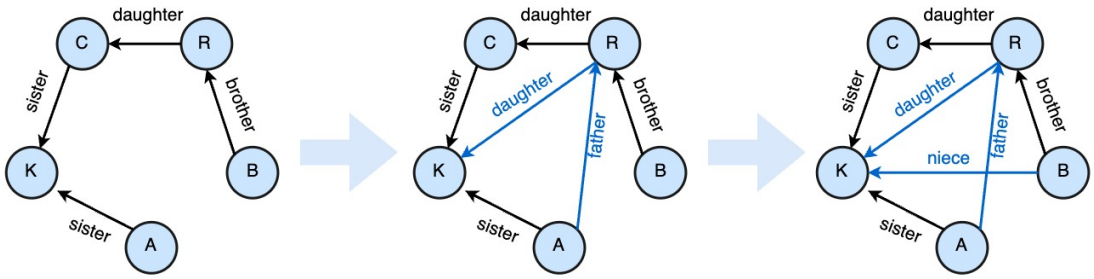


Figure 5.16: The family graph corresponding to the story shown in Figure 5.15. Edges representing family relations directly extracted from the story are colored in black, while those requiring derivation using common sense knowledge are colored in blue. Additionally, names are abbreviated using their initials.

```
1  // Relation declarations
2  type kinship(rela: String, sub: String, obj: String)
3  type composite(r1: String, r2: String, r3: String)
4  type question(sub: String, obj: String)
5
6  // Rules to derive the final answer
7  rel kinship(r3,a,c) = kinship(r1,a,b) and kinship(r2,b,c)
8                      and composite(r1,r2,r3) and a != c
9  rel answer(r) = question(s, o), kinship(r, s, o)
10
11 // Integrity constraints:
12 // (6 for kinship and 2 for rule learning)
13 rel violate(!r) = r := forall(a, b: kinship("mother", a, b)
14   => kinship("son", b, a) or kinship("daughter", b, a))
15 // Other constraints are omitted...
```

Listing 5.7: The Scallop program for reasoning over kinship graphs in CLUTRR.

clause, which reads "if b is a's brother and c is b's daughter, then c is a's niece":

$$\text{niece}(a, c) \leftarrow \text{brother}(a, b) \wedge \text{daughter}(b, c).$$

Note that the structure of the above rule can be captured by a higher-order logical predicate called "composite" (abbreviated as `comp`). This allows us to express many other similarly structured rules with ease. For instance, we can have `comp(brother, daughter, niece)` and `comp(father, mother, grandmother)`. With this set of rules, we may derive more facts based on known kinship relations. In fact, composition is the only kind of rule we need for kinship reasoning. In general, there are many other useful higher-order predicates to reason over knowledge bases, which we list out in Table 5.11.

The logic for reasoning over kinship relations is realized in Scallop in Listing 5.7. Line 2 declares

Predicate	Example
composite	`composite(mother, father, grandfather)`
transitive	`transitive(relative)`
symmetric	`symmetric(spouse)`
inverse	`inverse(husband, wife)`
implies	`implies(mother, parent)`

Table 5.11: Higher-order predicate examples.

the ternary relation `kinship` among `subject`, `object`, and their `relationships`. Line 3 then declares `composite` that is a higher-order predicate relating 3 kinship relations. We have line 7 declaring the rule that composites two existing kinship facts to derive a new kinship fact.

We note that integrity constraints are also included as logical rules. Specifically, we include a unary relation named `violate` storing boolean to encode the likelihood of integrity violations based on pre-defined rules. In this application, we choose to have violation (negative) rules rather than integrity (positive) rules for two reasons. First, it is more modular because multiple violation criteria can be "or"-ed together to form a larger violation criteria, allowing violation rules to be expressed as multiple Scallop rules. Secondly, the likelihood of integrity violation can be directly used for semantic constraint loss, which we will introduce later in Section 5.6.2.

5.6.2 Learning Pipeline

The learning pipeline concerns tightly integrating a perceptive model for relation extraction with the symbolic engine, Scallop, for logical reasoning. There are two add-ons we introduce for this specific application. First, we initialize the common sense knowledge rules used for logical deduction using language models, then further tune them through our end-to-end pipeline, alleviating human efforts. Secondly, we employ integrity constraints on the extracted relation graphs and the logical rules, to improve the logical consistency of LMs and the learned rules.

Based on this design, we formalize our method as follows. We adopt pretrained LMs to build relation extractors, denoted $\mathcal{M}_\theta$, which take in the natural language input x and return a set of probabilistic relational symbols $\mathbf{r}$. Next, we employ a differentiable deductive reasoning program, $\mathcal{P}_\phi$, where ϕ represents the weights of the learned logic rules. It takes as input the probabilistic relational symbols and the query q and returns a distribution over $\mathcal{R}$ as the output $\hat{y}$. Overall, the deductive model is written as

$$\hat{y} = \mathcal{P}_\phi(\mathcal{M}_\theta(x), q). \tag{5.1}$$

Additionally, we have the semantic loss (`sl`) derived by another symbolic program $\mathcal{P}_{\texttt{sl}}$ computing the probability of violating the integrity constraints:

$$l_{\texttt{sl}} = \mathcal{P}_{\texttt{sl}}(\mathcal{M}_\theta(x), \phi) \tag{5.2}$$

Combined, we aim to minimize the objective J over training set $\mathcal{D}$ with loss function $\mathcal{L}$:

$$J(\theta, \phi) = \frac{1}{|\mathcal{D}|} \sum_{(x,q,y)\in\mathcal{D}} w_1 \mathcal{L}(\mathcal{P}_\phi(\mathcal{M}_\theta(x), q), y) + w_2 \mathcal{P}_{\texttt{sl}}(\mathcal{M}_\theta(x), \phi), \tag{5.3}$$

where w_1 and w_2 are tunable hyper-parameters to balance the deduction loss and semantic loss. Though shown as two separate programs $\mathcal{P}_\phi$ and $\mathcal{P}_{\texttt{sl}}$, they share the same Scallop program in practice, as shown in Listing 5.7. We only need to additionally configure the Scallop module to output two relations, `answer` (for kinship prediction) and `violation` (for semantic loss).

5.6.3 Relation Extraction

Since pre-trained LMs have strong pattern recognition capabilities for tasks like Named-Entity-Recognition (NER) and Relation Extraction (RE) [93, 87], we adopt them as our neural components in Scallop. To ensure that LMs take in strings of similar length, we divide the whole context into multiple windows. The goal is to extract the relations between every pair of entities in each windowed context. Concretely, our relation extractor $\mathcal{M}_\theta$ comprises three components: 1) a Named-Entity Recognizer (NER) to obtain the entities in the input text, 2) a pre-trained language model, to be fine-tuned, that converts windowed text into embeddings, and 3) a classifier that takes in the embedding of entities and predicts the relationship between them. The set of parameters θ contains the parameters of both the LM and the classifier.

We assume the relations to be classified come from a finite set of relations $\mathbf{R}$. For example in CLUTRR [85], we have 20 kinship relations including mother, son, uncle, father-in-law, etc. In practice, we perform $(|\mathbf{R}| + 1)$-way classification over each pair of entities, where the extra class stands for "n/a". The windowed contexts are split based on simple heuristics of "contiguous one to three sentences that contain at least two entities", to account for coreference resolution. The windowed contexts can be overlapping and we allow the reasoning module to deal with noisy and redundant data. Overall, assuming that there are m windows in the context x, we

extract $mn(n-1)(|\mathbf{R}|+1)$ probabilistic relational symbols. Each symbol is denoted as an atom of the form $p(s, o)$, where $p \in \mathbf{R} \cup \{\text{n/a}\}$ is the relational predicate, and s, o are the two entities connected by the predicate. We denote the probability of such symbol extracted by the LM and relational classifier as $\Pr(p(s, o) \mid \theta)$. All these probabilities combined form the output vector $\mathbf{r} = \mathcal{M}_\theta(x) \in \mathbb{R}^{mn(n-1)(|\mathbf{R}|+1)}$.

Rule learning. Hand-crafted rules could be expensive or even impossible to obtain. To alleviate this issue, Scallop applies LMs to help automatically extract rules, and further utilizes the differentiable pipeline to fine-tune the rules. Each rule such as is attached a weight, initialized by prompting an underlying LM. Let a composition rule be $prob :: \text{comp}(r, p, q)$, it means one's r's p is their q. For example, the facts listed in Listing 5.8 means, one's father's father is always one's grandfather (probability 1.0). At the same time, one's brother's daughter is one's niece with 0.9 probability.

```
rel composite = {
  1.0::("father", "father", "grandfather"),
  0.9::("brother", "daughter", "niece"),
  // ... other weighted composite rules
}
```

Listing 5.8: A few probabilistic composite rules that are learnt.

Given that the relations $r, p, q \in R$, Scallop automatically enumerates r and p from R while querying for LM to unmask the value of q. LM then returns a distribution of words, which we take an intersection with R. The probabilities combined form the initial rule weights ϕ. This type of rule extraction strategy is different from existing approaches in inductive logic programming since we are exploiting LMs for existing knowledge about relationships.

Note that LMs often make simple mistakes answering such prompt. In fact, with the above prompt, even GPT-3 can only produce 62% of composition rules correctly. While we can edit prompt to include few-shot examples, in this work we consider fine-tuning such rule weights ϕ within our differentiable reasoning pipeline. Note that there are exponentially many rule weights to be fine-tuned. For example, the composition rule used for kinship reasoning has 3 arguments, resulting in $|R|^3 = 20^3$ candidate rules.

In practice, we use two optimizers with different hyper-parameters to update the rule weights ϕ and the underlying model parameter θ, in order to account for optimizing different types of weights.

Semantic loss and integrity constraints. In general, learning with weak supervision label is hard, not to mention that the deductive rules are learnt as well. We thereby introduce an additional semantic loss during training. Here, semantic loss is derived by a set of integrity constraints used to regularize the predicted entity-relation graph as well as the learnt logic rules. In particular, we consider rules that detect *violations* of integrity constraints. For example, "if A is B's father, then B should be A's son or daughter" is an integrity constraint for relation extractor—if the model predicts a father relationship between A and B, then it should also predict a son or daughter relationship between B and A. Encoded in first order logic, it is

$$\forall a, b, \text{father}(a, b) \Rightarrow (\text{son}(b, a) \vee \text{daughter}(b, a)).$$

The violation of this first order logic formula is encoded in Scallop as the line 13-14 in Listing 5.7. Through differentiable reasoning, we evaluate the probability of such constraint being violated, yielding our expected *semantic loss*. In practice, arbitrary number of constraints can be included, though too many interleaving ones could hinder learning.

Chapter 6

Advanced Applications

In this chapter, I propose three advanced applications for the completion of my dissertation. While preceding chapters explored simpler tasks and the use of Scallop in such domains, it is crucial to demonstrate Scallop's capability in real-life, complex scenarios. For this, I propose to explore three applications of Scallop. Specifically, these are (a) video scene graph generation for computer vision (Section 6.1), (b) whole-program security vulnerability detection for program analysis (Section 6.2), and (c) RNA secondary structure prediction for bioinformatics (Section 6.3). We now elaborate on each proposed application and some preliminary explorations.

6.1 Video Scene Graph Generation

Understanding video semantics has gained prominence due to a wide range of applications such as video search, text-video retrieval, video question answering, video segmentation, and video captioning. Video semantics constitutes two crucial aspects: *spatial semantics*, which concern the entities in the video, their individual attributes, and their semantic relationships; and *temporal semantics*, which capture actions and properties evolving through time. For example, the video described in Figure 6.1 by the phrase *"pushing a box off the desk by hand"* involves entities like *"box"* and *"hand"*, which are connected by the spatial relation *"touching"*. It also features two temporally consecutive states: the *"box"* is first *"on"* the *"desk"*, and then *"not above"* the *"desk"*.

To explicitly learn combined spatial and temporal semantics, a structured representation called *Spatio-Temporal Scene Graph* (STSG) [83, 107] has been proposed to represent entity relations

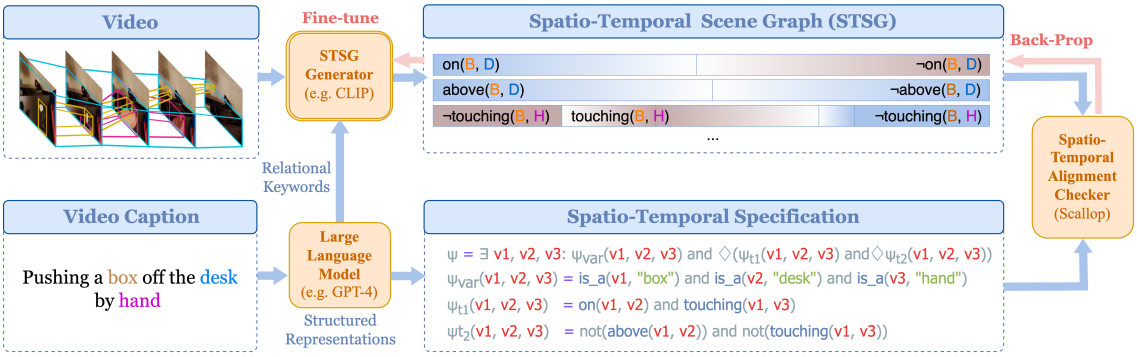


Figure 6.1: An example from 20BN demonstrating the end-to-end learning pipeline. The model M_θ processes a video to generate a probabilistic STSG. With 3-shot GPT-4, an STSL specification is derived from the video caption, which describes a temporal sequence of two events: "the box is on the desk touched by a hand" and "the box is not above the desk." The alignment checker then aligns the STSL program with the probabilistic STSG.

throughout a video. Existing approaches for learning STSG from video data are typically fully-supervised, e.g., [65, 18]. They can potentially learn high-fidelity STSGs from video data but are greatly hindered in practice due to the complexity of low-level annotations that are laborious to obtain [101]. We combine STSG and STSL into a novel framework for learning fine-grained video representations. In particular, we implement STSL and a specification checker for it atop the Scallop neurosymbolic framework. Our checker computes an *alignment score* between a pair of predicted STSG and an STSL specification describing the input video. Intuitively, the alignment score represents the likelihood of the STSL specification being satisfied over the STSG extracted from the video. We leverage the end-to-end differentiable reasoning capability of Scallop to enable training of the STSG model using weak supervision. To provide additional supervision, we incorporate contrastive learning, time-span supervision, and semantic loss into our loss function design. Further, since such specifications are usually unavailable in existing datasets, we devise a generic prompting template that utilizes a large language model like GPT-4 [71] to convert any caption into an STSL program. This enables us to leverage widely accessible video datasets with captions for learning a fine-grained STSG extraction model.

Weak supervision emerges as a promising approach to address this challenge. For example, the vast availability of video captions provides a valuable source of weak supervisory signals. However, key difficulties arise in effectively learning STSGs from such weak supervision. Is it even feasible to use video captions given the sparsity and noise in the signals they provide? Captions often focus only on the primary objects, ignoring underlying details, and many temporal signals are either hidden

or must be inferred. How can we provide useful fine-grained signals under such circumstances? To address these challenges, we propose transforming captions into *logical specifications* using large language models to explicitly reveal the hidden spatial and temporal information. This transformation creates a shared foundation to systematically align captions with predicted STSGs. The alignment process should a) capture both spatial and temporal nuances to provide fine-grained supervision for underlying STSG generators; b) allow diversity, naturalness, and fuzziness in the video and caption data; and c) account for common-sense knowledge that may be implicit or ambiguous in the captions.

We set out by designing STSL, a general and expressive Spatio-Temporal Specification Language for specifying fine-grained spatio-temporal properties. STSL is grounded in *Finite Linear Temporal Logic* (LTL_f) [22] which is used to describe temporal properties over finite traces of action and states. STSL subsumes action sequences commonly seen in video-action alignment tasks [11] while capturing additional temporal nuances such as "until" ($\mathbf{U}$) and "finally" ($\Diamond$). It also allows to express common-sense constraints for extra supervision. Finally, combined with relational predicates extracted from natural language, such as "is pushing off" and "lies above", it can even specify the open-domain spatial semantics of videos.

We begin by presenting the high-level problem definition. We are given a dataset $\mathcal{D}$ of video-caption pairs (X, C), where $X = [x_1, \ldots, x_n]$ is a video containing n frames, and C is a natural language caption describing the video. We wish to learn a neural model M_θ which extracts a spatio-temporal scene graph (STSG), $\hat{\mathbf{r}} = M_\theta(X)$ that conforms to the corresponding caption C. During training time, given a loss function $\mathcal{L}$, we aim to minimize the following main objective:

$$J(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(X,C)\in\mathcal{D}} \mathcal{L}(\Pr(M_\theta(X) \models \texttt{LLM}(C)), 1), \tag{6.1}$$

where $\texttt{LLM}(C)$ is an STSL formula ψ generated by LLM from the caption C, and $\Pr(\hat{\mathbf{r}} \models \psi)$ is the alignment score (probability of alignment) computed by our spatio-temporal alignment checker. We illustrate the full learning pipeline in Figure 6.1 and detail the process in this section.

6.1.1 Video to Probabilistic Relational Database

A probabilistic spatio-temporal scene graph is a probabilistic relational database that contains two types of facts denoted by relations `unary_atom` and `binary_atom`, for unary and binary predicates

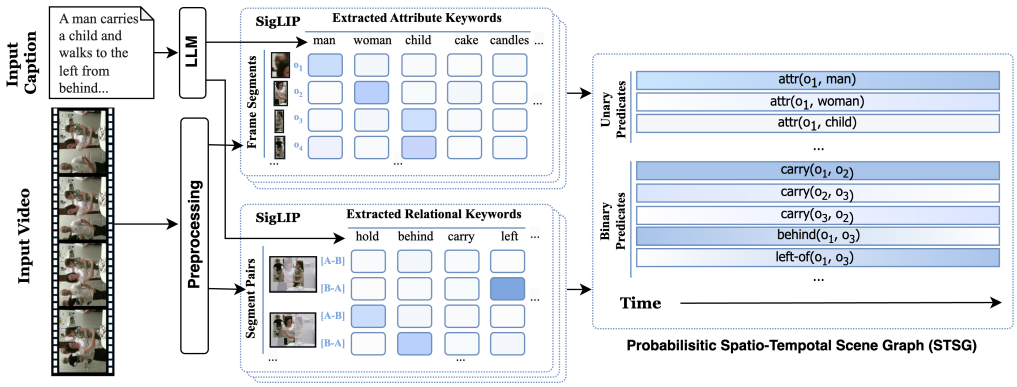


Figure 6.2: Pipeline illustration with CLIP as the backbone model for probabilistic STSG generation.

$$
\begin{aligned}
\text{(Term)}\quad & t ::= c \mid v \\
\text{(Formula)}\quad & \varphi ::= a(\bar{t}) \mid \neg a(\bar{t}) \mid \varphi_1 \wedge \varphi_2 \\
& \quad \mid \bigcirc \varphi \mid \varphi_1 \ \mathbf{U} \ \varphi_2 \mid \Box \varphi \mid \Diamond \varphi \\
\text{(Specification)}\quad & \psi ::= \exists v_1, \ldots, v_k, \text{s.t. } \varphi
\end{aligned}
$$

Figure 6.3: The formal syntax of STSL, where a represents relational predicates, c represents constants, and v represents variables. Here, $\wedge$, and $\neg$ represents logical "and", "or", and "not". Formula may also contain temporal operators $\bigcirc$ (next), $\mathbf{U}$ (until), $\Box$ (global), and $\Diamond$ (finally).

respectively, each associated with a probability denoting the likelihood that the fact is true. For example, `0.05::unary_atom("deformed", 3, e)` means that *"entity* `e` *is unlikely to be deformed at time stamp 3,"* while `0.92::binary_atom("push", 10, h, b)` indicates that *"object* `h` *is highly likely to be pushing object* `b` *at time stamp 10."* This flexible representation supports the seamless incorporation of unary and binary keywords into the database. The unified probabilistic database enables our method to be model-agnostic, supporting both closed-domain STSG classification models and open-world vision-language models for converting input video data into relational database representations. With a unified formalization, an STSG generator, M_θ, parameterized by θ, takes in pixel-based raw video data X, and generate a distribution of STSGs. This distribution is then encoded as a predicted probabilistic relational database, $\hat{\mathbf{r}} = M_\theta(X)$.

6.1.2 Spatio-Temporal Specification Language

Linear Temporal Logic (LTL) [73] is a formal logic system extending propositional logic with concepts about time. It is commonly used for formally describing temporal events, with applications in software verification [10, 44] and control [24, 79]. As we operate on prerecorded, finite-length videos, our language is developed using LTL_f [22], which supports LTL reasoning over finite traces.

Thus, we use LTL_f as a framework for specifying events and their temporal relationships.

Our STSL (Figure 6.3) further extends LTL_f by introducing relational predicates and variables. It starts from the specification ψ which existentially quantifies variables in an STSL formula. The formula φ is inductively defined, with basic elements as relational atoms α of the form $a(t_1, \ldots, t_n)$. Note that the terms $\bar{t} = \{t_1, \ldots, t_n\}$ can contain quantified variables to be later grounded into concrete entities based on context Γ, noted by $\text{subst}_\Gamma(\bar{t})$. From here, φ can be constructed using basic propositional logic components $\wedge$ (and), $\vee$ (or), and $\neg$ (not). The system additionally includes temporal unary operators $\Box$ (always), $\Diamond$ (finally), $\bigcirc$ (next), and a binary operator $\mathbf{U}$ (until) [2]. For example, the description "A hand continues to touch the box until it drops." can be represented as an STSL formula

$$\psi = \texttt{touch(h, b)}\ \mathbf{U}\ \texttt{drop(b, \_)}. \tag{6.2}$$

Note that an argument to the predicate `drop` is a wildcard (`_`), since we do not specify where does the box drops from. This formula might seem too strict since it requires the two events to be consecutive. To make the specification more natural, one can change the above formula to "$\Diamond$(`touch(h, b)` $\wedge$ $\Diamond$`drop(b, _)`)". Here, the two events, `touch` and `drop`, need to happen in chronological order but are not required to be consecutive.

In Listing 6.1 we define the STSL using algebraic data type (ADT). We stratify STSL formula into temporal formula and logical formula so that the semantics can be implemented more succinctly later. In temporal section (line 6-10), we cover all the core temporal operations supported in STSL. In the logical section (line 14-18), we can have conjunction as the only logical connective; negation can only be applied to unary or binary atoms, allowing us to define `NegUnary` and `NegBinary` atomic formulas. Note that if we allow arbitrary negation, then our semantics might suffer from unstratified negation, prohibiting our program to be compiled by Scallop compiler. On line 21-23, we show one example specification representing Eqn. 6.2.

6.1.3 Natural Language to Spatio-Temporal Specification

To leverage the abundance of video captions as weak supervision signals, we employ a large language model (LLM) to automatically extract a programmatic specification ψ from each video caption c. Directly converting captions into a formal program is particularly challenging for an LLM, especially in a low-data language like STSL. We hence use a few-shot learning approach with an

```
// Term: either constant string or variable or wildcard
type Term = Const(String) | Var(Var) | Wildcard()

// TForm: temporal formula, containing `Global`, `Finally`,
// `Until`, and `Next`
type TForm = Global(TForm)
           | Finally(TForm)
           | Until(TForm, TForm)
           | Next(TForm)
           | Logic(LForm)

// LForm: logical formula, containing logical conjunction
// and atomic queries
type LForm = And(LForm, LForm)
           | Unary(String, Term)
           | Binary(String, Term, Term)
           | NegUnary(String, Term)
           | NegBinary(String, Term, Term)

// an example specification: touch(h,b) U drop(b,_)
const MY_SPEC = Until(
    Binary("touch", Var("h"), Var("b")),
    Binary("drop", Var("b"), Wildcard()))
```

Listing 6.1: STSL defined in Scallop.

LLM to generate an intermediate structured representation of the caption in JSON format. For each caption c, our goal is to convert it into a series of events $\bar{e} = \{e_1, e_2, \ldots, e_n\}$. Each event includes (a) a detailed natural language description of the event, which guides the generation of subsequent details, (b) a series of unary, binary, positive, and negative predicates describing the semantics of the scenario, (c) the location of the event, $\text{loc}(e_i)$, in the video where the event occurs, represented as a fraction of the video length, and (d) the duration of the event, $\text{dur}(e_i)$, also expressed as a fraction.

To extract such structured representations from the caption, we designed a generic prompt

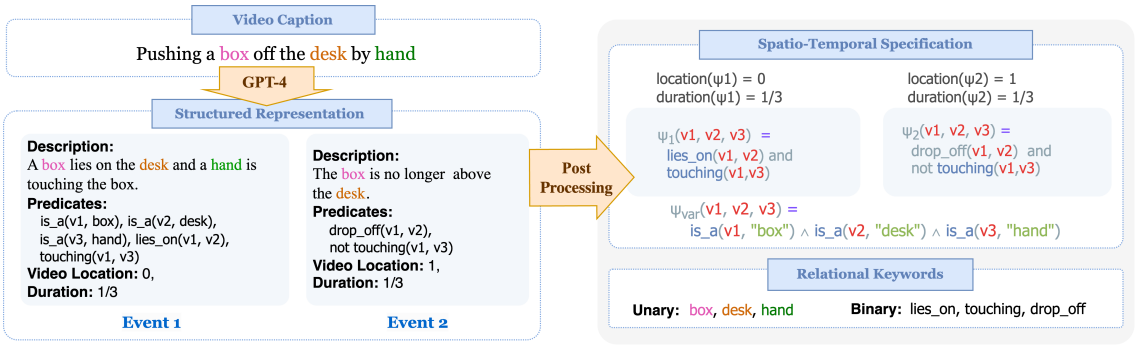


Figure 6.4: An illustration of our pipeline for natural language caption to programmatic spatio-temporal specification.

$$
\begin{array}{rll}
\langle w, s\rangle \models \psi & \text{iff } \exists \Gamma, \langle \Gamma, w, s\rangle \models \varphi \\
\langle \Gamma, w, s\rangle \models a(\bar{t}) & \text{iff } a(\bar{c}) \in w[s] \ \wedge \bar{c} = \text{substitute}_\Gamma(\bar{t}) \\
\langle \Gamma, w, s\rangle \models \varphi_1 \wedge \varphi_2 & \text{iff } \langle \Gamma, w, s\rangle \models \varphi_1 \ \wedge \langle \Gamma, w, s\rangle \models \varphi_2 \\
\langle \Gamma, w, s\rangle \models \neg\varphi & \text{iff } \langle \Gamma, w, s\rangle \not\models \varphi \\
\langle \Gamma, w, s\rangle \models \bigcirc\varphi & \text{iff } \langle \Gamma, w, s+1\rangle \models \varphi \\
\langle \Gamma, w, s\rangle \models \varphi_1 \mathbf{U} \varphi_2 & \text{iff } \exists i.s \leq i \wedge \langle \Gamma, w, i\rangle \models \varphi_2, \\
 & \qquad \forall k.s \leq k < i, \langle \Gamma, w, k\rangle \models \varphi_1
\end{array}
$$

Figure 6.5: Formal semantics of STSL. $\langle w, s\rangle \models \psi$ means the STSL specification ψ is *aligned* with the ST-SG w starting from time s. We use $w \models \psi$ as an abbreviation for $\langle w, 1\rangle \models \psi$.

template, which consists of the following components: (a) examples for temporal specification in fraction numbers: "0", "1/2", "2/3", "1". (b) scene graph keywords, such as object names and relations. (c) few-shot examples of caption and JSON structured representations pairs. We illustrate a caption and its structured representation in Figure 6.4.

The programmatic spatio-temporal specification is then generated by postprocessing the events in sequential order. Consequently, we can generate the programmatic spatio-temporal specification ψ for the caption as a sequence of events in chronological order:

$$\psi = \Diamond_{e_i \in \bar{e}} \psi_i, \quad \psi_i = \bigwedge_{\phi_j \in \psi_i} \phi_j \tag{6.3}$$

6.1.4 Spatio-Temporal Alignment Checking

Given a probabilistic database $\mathbf{r}$ that encodes a distribution of STSGs (§6.1.1), and a specification ψ in STSL, we aim to measure the alignment score $\Pr(\mathbf{r} \models \psi)$ in an end-to-end and differentiable manner. Conceptually, each probabilistic fact f in the database can be toggled on or off, resulting in $2^{|\mathbf{r}|}$ distinct *worlds*. Denoting each world (i.e. a concrete STSG) as $w \in \mathcal{P}(\mathbf{r})$ [1], we can check whether the world w satisfies the specification ψ or not. From here, the alignment score can be computed as the sum of the probabilities of worlds satisfying ψ:

$$\Pr(\mathbf{r} \models \psi) = \sum_{\substack{w \in \mathcal{P}(\mathbf{r}) \\ w \models \psi}} \Pr(w), \tag{6.4}$$

$$\Pr(w) = \textstyle\prod_{f \in w} \Pr(f) \prod_{f' \in \mathbf{r} \setminus w} (1 - \Pr(f')) \tag{6.5}$$

[1] $\mathcal{P}$ represents power-set.

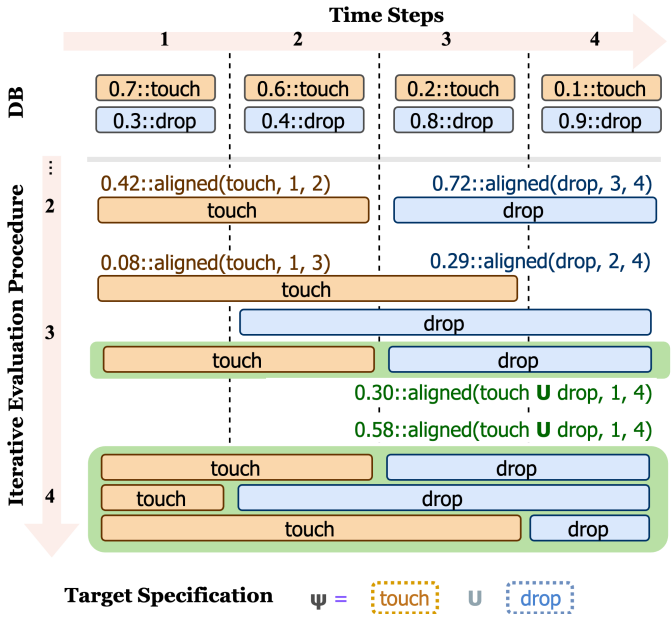


Figure 6.6: The evaluation process aligning a spatio-temporal scene graph (DB) with a specification `climb` **U** `walk`. This figure elides showing the arguments of the relational predicates and focuses only on matching sequential events.

Enumerating all possible worlds is intractable due to its exponential complexity. Since Scallop employ scalable algorithms, we can approximate this probability and greatly reduce the probabilistic reasoning time. We also note that some of the STSG w sampled from $\mathcal{P}(\mathbf{r})$ might be infeasible due to involving conflicting facts (e.g., a box is above and below a desk at the same time). To further enhance the logic deduction efficiency, we extend Scallop's "top-k proofs" provenance to support general disjunctive constraints and early removal of infeasible STSGs that do not satisfy the specification.

We implement the alignment checker with Scallop, as partially shown in Listing 6.3. This helps us to succinctly and precisely encode the formal semantics of STSL. It inductively computes the alignment between a temporal slice of $\mathbf{r}$ and an STSL formula. The whole specification ψ is aligned if the full $\mathbf{r}$ (from 1 to m) satisfies ψ with a concrete variable grounding Γ, which maps variables to concrete entities. We illustrate one simplified evaluation process in Figure 6.6. The checker iteratively aligns the predicted probabilistic events (simplified to just climb and walk) with the specification. At the 4th iteration, 3 different satisfying alignments are derived, yielding a final alignment score of 0.58.

As for our Scallop implementation of the alignment checker, we start by defining the relations

```
1  // Type definitions
2  type Var = String
3  type Obj = u32
4  type Time = u32
5
6  // Spatial-temporal scene graph
7  type time(time: Time)
8  type unary_atom(pred: String, fid: Time, o1: Obj)
9  type binary_atom(pred: String, fid: Time, o1: Obj, o2: Obj)
10
11 // Variable assignments
12 type name(o: Obj, name: Var)
13 type variable(var: Var), object(o: Obj)
14 rel var_obj = disjunct[v](o: variable(v) and object(o))
```

Listing 6.2: Setting up the Spatio-Temporal Scene Graph (STSG) as well as the variable assignment solving context.

storing our STSG (line 7-9 of Listing 6.2). Since in STSG we only deal with unary and binary relations, we hardcode the two relations `unary_atom` and `binary_atom`. One important difference between our alignment checker and the semantics of other DSLs shown before is that, alignment checker needs the *constraint solving* capability due to specifications having variables. Specifically, we need to solve each variable v to a concrete object o in the scene. This means that each variable can only be assigned to one object, forming a *mutual exclusion*. For this, we use an aggregator in Scallop, `disjunct`, that constructs the mutual exclusion in the tag space, as shown on line 5 of Listing 6.2.

We now present the Scallop code for the alignment checker for STSL (Listing 6.3). Starting from line 1-5, we define the relation used to ground each term into concrete objects. Specifically, when the term is a variable (`Var`), we use the `var_obj` relation defined in Listing 6.2 to ground it into an object `o`. Note that `var_obj` has mutual exclusion within it, meaning that if two facts where a single variable is assigned to two objects present in a single proof, then the proof will be rejected by Scallop's provenance system. We continue to define the rules for aligning logical formula, which require the grounding of all terms appearing in the atoms. As for aligning temporal formula, we deal with temporal relations. For instance, on line 21, aligning `Global` formula at time step `s` means that the subformula `p1` needs to be aligned for all time steps between `s` and `n`. This is exactly what we define in the formal semantics of STSL (Figure 6.5).

```
// grounding a term into an object
type ground_term(t: Term, o: Obj)
rel ground_term(Var(v), o) = var_obj(v, o)
rel ground_term(Const(c), o) = name(o, c)
rel ground_term(Wildcard(), o) = object(o)

// aligning logical formula
type align_lform(phi: LForm, s: Time)
rel align_lform(Binary(pred, t1, t2), s) =
  ground_term(t1, o1) and ground_term(t2, o2)
  and binary_atom(pred, s, o1, o2) and time(s)
rel align_lform(NegBinary(pred, t1, t2), s) =
  ground_term(t1, o1) and ground_term(t2, o2)
  and not binary_atom(pred, s, o1, o2) and time(s)
// handling other logical formulas...

// aligning temporal formula: the formula `psi` is aligned
// with the scene graph starting from time `s`, given the
// variable assignment context
type align_tform(psi: TForm, s: Time)
rel align_tform(Global(p1), s) =
  max_time(n) and align_all_tform(p1, s, n)
rel align_tform(Finally(p1), s) = align_once_tform(p1, s)
rel align_tform(Until(p1, p2), s) =
  time(t + 1) and s < (t + 1)
  and align_all_tform(p1, s, t) and align_tform(p2, t + 1)
// handling other temporal formulas...
```

Listing 6.3: The (partial) alignment checker for STSL, implemented in Scallop.

6.1.5 Preliminary Results and Proposed Tasks

We have implemented the Scallop neurosymbolic pipeline for the task of video scene graph generation, and obtained preliminary results on the dataset of OpenPVSG, outperforming many existing baselines. I propose to continue expanding the STSL so that more systematic specifications can be generated from natural language, thus enabling larger scale training.

6.2 Security Vulnerability Detection

Software is prone to security vulnerabilities. Program analysis tools to detect them have limited effectiveness in practice due to their reliance on human labeled specifications. Large language models (or LLMs) have shown impressive code generation capabilities but they cannot do complex reasoning over code to detect such vulnerabilities especially since this task requires whole-repository analysis. We propose to use Scallop to implement a neuro-symbolic approach that systematically combines LLMs with static analysis to perform whole-repository reasoning for security vulnerability detection. Specifically, Scallop leverages LLMs to infer taint specifications and perform contextual analysis, alleviating needs for human specifications and inspection.

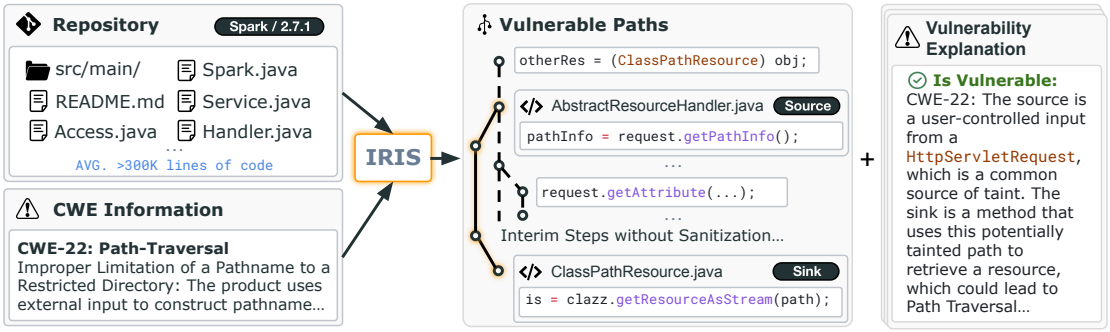


Figure 6.7: Overview of the IRIS neuro-symbolic system. It checks a given whole repository for a given type of vulnerability (CWE) and outputs a set of potential vulnerable paths with explanations.

We have implemented a preliminary version for LLM-assisted security vulnerability detection, in a tool named IRIS. IRIS is a neuro-symbolic approach for vulnerability detection that combines the strengths of static analysis and LLMs. Fig. 6.7 presents an overview of IRIS. Given a project to analyze for a given vulnerability class (or CWE), IRIS applies LLMs for mining CWE-specific taint specifications. IRIS augments such specifications with CodeQL, a tool for static taint analysis. Our intuition here is because LLMs have seen numerous usages of such library APIs, they have an understanding of the relevant APIs for different CWEs. Further, to address the imprecision problem of static analysis, we propose a contextual analysis technique with LLMs that reduces the false positive alarms and minimizes the triaging effort for developers. Our key insight is that encoding the code-context and path-sensitive information in the prompt elicits more reliable reasoning from LLMs. Finally, our neuro-symbolic approach allows LLMs to do more precise whole-repository reasoning and minimizes the human effort involved in using static analysis tools.

At a high level, IRIS takes a Java project P, the vulnerability class C to detect, and a large language model `LLM`, as inputs. IRIS statically analyzes the project P, checks for vulnerabilities specific to C, and returns a set of potential security alerts A. Each alert is accompanied by a unique code path from a taint source to a taint sink that is vulnerable to C (i.e., the path is unsanitized).

As illustrated in Fig. 6.8, IRIS has four main stages: First, IRIS builds the given Java project and uses static analysis to extract all candidate APIs, including invoked external APIs and internal function parameters. Second, IRIS queries an LLM to label these APIs as sources or sinks that are specific to the given vulnerability class C. Third, IRIS transforms the labelled sources and sinks into specifications that can be fed into a static analysis engine, such as CodeQL, and runs a vulnerability class-specific taint analysis query to detect vulnerabilities of that class in the project.

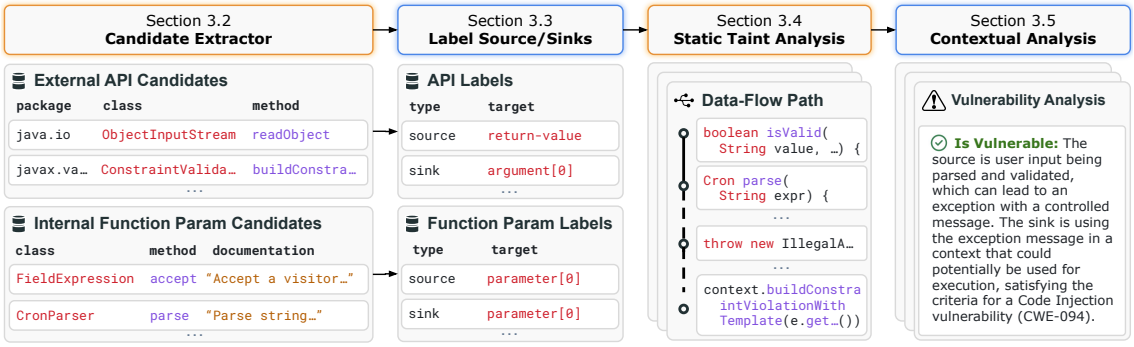


Figure 6.8: An illustration of the IRIS pipeline.

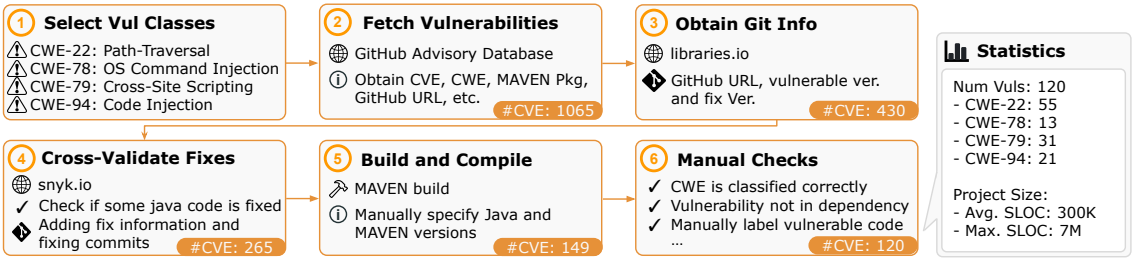


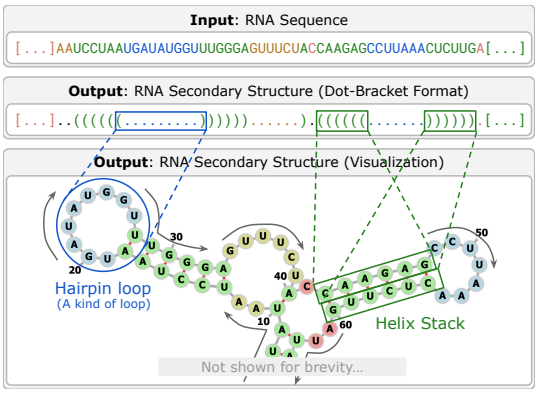
Figure 6.9: Steps for curating CWE-Bench-Java, and dataset statistics.

This step generates a set of vulnerable code paths (or alerts) in the project. Finally, IRIS triages the generated alerts by automatically filtering false positives, and presents them to the developer.

6.2.1 CWE-Bench-Java Dataset

To evaluate our approach, we require a dataset of vulnerable versions of Java projects with several important characteristics: 1) Each benchmark should have relevant **vulnerability metadata**, such as the CWE ID, CVE ID, fix commit, and vulnerable project version, 2) each project in the dataset must be **compilable**, which is a key requirement for static analysis and data flow graph extraction, 3) the projects must be **real-world**, which are typically more complex and hence challenging to analyze compared to synthetic benchmarks, and 4) finally, each vulnerability and its location (e.g., method) in the project must be **validated** so that this information can be used for robust evaluation of vulnerability detection tools. Unfortunately, no existing dataset satisfies all these requirements.

To address these requirements, we curate our own dataset of vulnerabilities. For this paper, we focus only on vulnerabilities in Java libraries that are available via the widely used Maven package manager. We choose Java because it is commonly used to develop server-side, Android,



(a) Illustration of the RNA folding problem. We visualize the output RNA secondary structure in the bottom. The grey arrow indicates the direction in which the indexes of nucleotides are increasing.

(Structure Token) $\tau ::= H_l \mid H_r \mid L_l \mid L_r \mid L_u \mid E_u$

(b) The set of structure tokens (terminal symbols) for our CFG, including helix (H), loop (L), and external loop (E). The subscripts l (left), r (right), and u (unpaired) denote the structural role of the corresponding nucleotide within each substructure.

$$
\begin{aligned}
\text{(Paired Substr.)} \quad & \mathcal{P} ::= \mathcal{L} \mid \mathcal{H} \\
\text{(Helix Stack)} \quad & \mathcal{H} ::= H_l \cdot \mathcal{P} \cdot H_r \\
\text{(Ext. Unpaired)} \quad & \mathcal{E} ::= E_u \mid \mathcal{E} \cdot E_u \\
\text{(Int. Unpaired)} \quad & \mathcal{U} ::= L_u \mid \mathcal{U} \cdot L_u \\
\text{(Loop Body)} \quad & \mathcal{B} ::= \mathcal{U} \mid \mathcal{P} \cdot \mathcal{U} \mid \mathcal{B} \cdot \mathcal{P} \mid \mathcal{B} \cdot \mathcal{U} \\
\text{(Loop)} \quad & \mathcal{L} ::= L_l \cdot \mathcal{B} \cdot L_r \\
\text{(RNA SS)} \quad & \mathcal{R} ::= \mathcal{P} \mid \mathcal{E} \mid \mathcal{R} \cdot \mathcal{R}
\end{aligned}
$$

(c) The complete CFG used to parse RNA secondary structures ($\mathcal{R}$).

Figure 6.10: RNA folding problem and the CFG that can parse RNA structures.

and web applications, which are prone to security risks. Further, due to Java's long history, there are many existing CVEs in numerous Java projects that are available for analysis. We initially use the GitHub Advisory database [30, 31] to obtain such vulnerabilities, and further filter it with cross-validated information from multiple sources, including manual verification. Fig. 6.9 illustrates the complete set of steps for curating CWE-Bench-Java.

As shown in the statistics (Fig. 6.9), the sheer size of these projects make them challenging to analyze for any static analysis tool or ML-based tool. Each project in CWE-Bench-Java comes with GitHub information, vulnerable and fix version, CVE metadata, a script that automatically fetches and builds, and the set of program locations that involve the vulnerability.

6.2.2 Perliminary Results and Proposed Task

Initial results shows that when combined with GPT-4, IRIS is capable of detecting 55 out of the 120 vulnerabilities in our CWE-Bench-Java dataset. While the effectiveness of the neurosymbolic approach is well justified, the technique is still lacking in generality, not to mention that the program is unneccessarily long and difficult to manage. I propose to implement the symbolic components of IRIS in Scallop, with the primary focus on relaxing the otherwise stubborn CodeQL semantics.

6.3 RNA Secondary Structure Prediction

6.3.1 Problem Definition

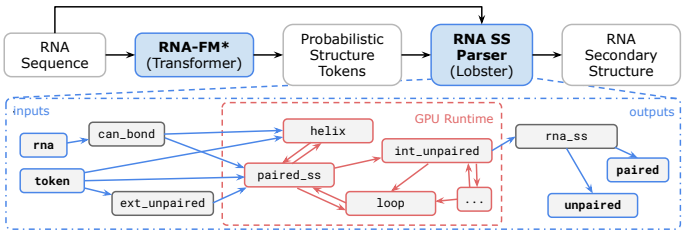
Given an RNA sequence of nucleotides ($\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$), the goal is to determine its secondary structure, which specifies whether each nucleotide at position i pairs with a nucleotide at position j, or remains unpaired. A primary way to represent this output is through "dot-bracket" notation, where unpaired nucleotides appear as dots, and paired nucleotides as matched brackets (Figure 6.10a). For the structure to be valid, all "brackets" must be properly matched, forming a well-structured output.

A neurosymbolic approach is a natural fit due to the symbolic yet non-deterministic nature of the problem. RNA molecules in nature can fold into multiple valid structures depending on environmental conditions. However, certain principles remain consistent, such as structural constraints that enforce matching pairs and pairing rules that allow only specific nucleotide pairs (`A-U`, `C-G`, and `A-G`). In our solution, the neural component makes data-driven predictions about the potential structural roles of nucleotides—such as whether a nucleotide resides within a helix-stack or an internal loop. The symbolic component, on the other hand, uses these predictions to parse the RNA sequence into a well-formed secondary structure. Without neurosymbolic, a purely neural approach would struggle to infer long-range dependencies, and a purely symbolic approach would have trouble adapting to specific datasets and distributions.

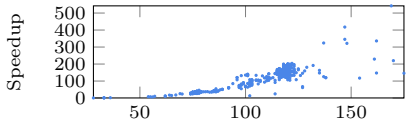
Our solution employs a CFG with its terminal symbols, or structure tokens, illustrated in Figure 6.10b. The neural network produces a probability distribution over these structure tokens for each nucleotide. The symbolic component then processes this sequence of distributions to identify the "most probable parse" that satisfies the CFG (Figure 6.10c). To generate the final dot-bracket notation, we extract all paired substructures $\mathcal{P}$ from the parsed syntax tree represented by $\mathcal{R}$.

6.3.2 Scalability and Programmability Challenges

Our neurosymbolic solution poses a significant scalability challenge. While the neural component can utilize modern hardware accelerators like GPUs and TPUs, the symbolic component currently runs on CPUs alone. The symbolic engine must derive a set of all possible predicted structures and their associated probabilities. As the length of the input sequence increases, the number of possible parses and the size of their associated weights also grows exponentially, leading to a combinatorial



(a) Our neurosymbolic pipeline for predicting RNA secondary structure. The main parsing loop is computed on the GPU.



Overall (475 Seqs)	Scallop	Scallop GPU
Time	13 hours	**6 minutes**
Speedup	1×	**146**×

(b) The speedup of Scallop GPU across different RNA sequence lengths relative to Scallop.

```
1  type Nucleotide = A | C | G | U                                  // enum type for nucleotides
2  type StructureToken = Hl | Hr | Ll | Lr | Lu | Eu          // enum type for structure tokens
3
4  type rna(idx: usize, nuc: Nucleotide)          // input RNA seq (index mapped to nucleotide)
5  type token(idx: usize, tok: StructureToken)     // probabilistic tokens extracted by RNA-FM
6
7  rel can_bond = {(A, U), (U, A), ...}       // facts for nucleotide pairs that can be bonded
8  rel paired_ss(i, j) = loop(i, j) or helix(i, j)            // Rule for paired-substructures
9  rel helix(i, j) =                                                  // Rule for helix stack
10   rna(i, x_i) and rna(j, x_j) and can_bond(x_i, x_j) and           // i,j must be bondable
11   token(i, Hl) and paired_ss(i + 1, j - 1) and token(j, Hr)    // production rule for helix
12 // ... other rules for parsing RNA are omitted for brevity
```

(c) Scallop program snippet that parses the RNA secondary structure according to the CFG in Figure 6.10c.

Figure 6.11: The overall pipeline, symbolic program, and the acceleration result of our solution.

explosion in the number of required computations.

While custom GPU implementations have been developed for these algorithms [104], it demands specialized knowledge (e.g., parallel programming in CUDA) and hinders domain experts from focusing on functionality. Even for the problem of RNA folding, such experts have designed diverse CFGs tailored to specific biological contexts, making it impractical to implement custom GPU solutions for each CFG variant. A general purpose framework is therefore desirable for making accelerated GPU computation widely accessible.

6.3.3 Our Approach

We present the high-level pipeline of our approach in Figure 6.11a. We specify the symbolic component as a Datalog program which is partially shown in Figure 6.11c. Datalog offers an intuitive interface for defining data types (lines 1-2) and relation types (lines 4-5) which we use to specify the inputs. Notice that the RNA sequence of nucleotides is encoded as a binary relation between indices (`usize`, unsigned integers) and the nucleotides.

Our program contains recursive rules, which are key for modeling the hierarchical structure of our CFG. Line 8 defines the rule for paired substructures ($\mathcal{P}$), which could be either an internal

loop ($\mathcal{L}$) or a helix-stack ($\mathcal{H}$). Lines 9-11 define the rule for helix-stacks ($\mathcal{H}$). Assuming that i-th and j-th nucleotides can be bonded (line 10), the subsequence $[i, j]$ can be parsed as a helix stack ($\mathcal{H}$) if the i-th and j-th nucleotide represent the left (L_l) and right (L_r) side of a helix-stack, while the enclosed subsequence $[i + 1, j - 1]$ forms a paired-substructure (line 11). Notice how closely these declarative rules mirror the corresponding CFG rules depicted in Figure 6.10c.

With the declarative high-level language, Scallop offers a convenient abstraction to hide the underlying GPU runtime from users. Figure 6.11a shows the dependency graph of the relations needed to compute RNA secondary structures. Further, Scallop-GPU off-loads the main recursion computation involving the inter-dependent relations to the GPU runtime for maximum acceleration.

The probabilistic reasoning semantics is also abstracted away. The `token` relation in Figure 6.11c (line 5) contains probabilistic facts extracted by the underlying neural network. For example, a fact `0.97::token(16, Hl)` represents that the 16-th nucleotide in the RNA is predicted to be the left element of a helix-stack, with the probability of 0.97. All the derived facts, such as `helix(i, j)`, will carry probabilities computed from a concrete subsequence of structure tokens. This is manifested by the underlying provenance framework within Scallop. In our solution, we use a probabilistic provenance called `top-1-proof` which instruments the program to carry the most-likely parse for any given subsequence. We propose to speed up the probabilistic reasoning semantics to its full potential by Scallop-GPU.

6.3.4 Preliminary Results and Proposed Tasks

Preliminary results show that, by employing a fine-tuned RNA-FM model [13]—a foundation model for RNA sequence embedding—as the neural component, this neurosymbolic solution achieves a 92.6% F1 score on the ArchiveII dataset [86], surpassing established methods such as MxFold2 [81] which scores 88.9%. Moreover, as shown by Figure 6.11b, Scallop-GPU is capable of achieving a speedup of 146× on the given datasets containing RNA sequences up to length 175, compared to CPU only Scallop. However, it remains to be seen how well does this neurosymbolic approach generalize to even longer sequences and more diverse datasets. I propose to employ Scallop-GPU to continue scaling the employed symbolic component, while we continue to develop neurosymbolic approach that can help resolving long-term dependency from existing RNA foundation models.

Bibliography

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

[2] Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas. *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555. Springer Science & Business Media, 2009.

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[4] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. In *PLDI*, 2022.

[6] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ B. Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, and et al. On the opportunities and risks of foundation models, 2021.

[7] Tim Brooks, Aleksander Holynski, and Alexei A. Efros. Instructpix2pix: Learning to follow image editing instructions, 2023.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language

models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[9] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[10] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[11] Chien-Yi Chang, De-An Huang, Yanan Sui, Li Fei-Fei, and Juan Carlos Niebles. D3tw: Discriminative differentiable dynamic time warping for weakly supervised action alignment and segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3546–3555, 2019.

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

[13] Jiayang Chen, Zhihang Hu, Siqi Sun, Qingxiong Tan, Yixuan Wang, Qinze Yu, Licheng Zong, Liang Hong, Jin Xiao, Tao Shen, Irwin King, and Yu Li. Interpretable rna foundation model from unannotated data for highly accurate rna structure and function predictions, 2022.

[14] 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. Evaluating large language models trained on code. 2021.

[15] Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V. Le. Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension. In *International Conference on Learning Representations (ICLR)*, 2020.

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

[17] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[18] Yuren Cong, Wentong Liao, Hanno Ackermann, Michael Ying Yang, and Bodo Rosenhahn. Spatial-temporal transformer for dynamic scene graph generation. *CoRR*, abs/2107.12309, 2021.

[19] Noel Csomay-Shanklin, William D. Compton, Ivan Dario Jimenez Rodriguez, Eric R. Ambrose, Yisong Yue, and Aaron D. Ames. Robust agility via learned zero dynamics policies, 2024.

[20] Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. Semiring provenance for fixed-point logic. In *Conference on Computer Science Logic (CSL)*, 2021.

[21] Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[22] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. Association for Computing Machinery, 2013.

[23] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. 2020.

[24] Xuchu Ding, Stephen L Smith, Calin Belta, and Daniela Rus. Optimal control of markov decision processes with linear temporal logic constraints. *IEEE Transactions on Automatic Control*, 59(5):1244–1257, 2014.

[25] Kevin Ellis, Adam Albright, Armando Solar-Lezama, Joshua B. Tenenbaum, and Timothy J. O'Donnell. Synthesizing theories of human language with bayesian program induction. *Nature Communications*, 13, 2022.

[26] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020.

[27] Difei Gao, Ruiping Wang, Shiguang Shan, and Xilin Chen. From two graphs to N questions: A VQA dataset for compositional reasoning on vision and commonsense. 2019.

[28] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023.

[29] Ross B. Girshick. Fast R-CNN. 2015.

[30] GitHub. Github advisory database, 2024. https://github.com/advisories.

[31] GitHub. Github security advisories, 2024. https://github.com/github/advisory-database.

[32] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2013.

[33] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *ACM Symposium on Principles of Database Systems (PODS)*, 2007.

[34] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. 2021.

[35] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized orthogonal basis projections. 2022.

[36] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training, 2022.

[37] Thomas Hayes, Songyang Zhang, Xi Yin, Guan Pang, Sasha Sheng, Harry Yang, Songwei Ge, Qiyuan Hu, and Devi Parikh. Mugen: A playground for video-audio-text multimodal understanding and generation, 2022.

[38] Amir Hertz, Ron Mokady, Jay Tenenbaum, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Prompt-to-prompt image editing with cross attention control, 2022.

[39] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[40] Drew A. Hudson and Christopher D. Manning. GQA: a new dataset for compositional question answering over real-world images, 2019.

[41] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

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

[43] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning, 2016.

[44] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming: 25th International Colloquium, ICALP'98 Aalborg, Denmark, July 13–17, 1998 Proceedings 25*, pages 1–16. Springer, 1998.

[45] Wonjae Kim, Bokyung Son, and Ildoo Kim. Vilt: Vision-and-language transformer without convolution or region supervision, 2021.

[46] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything, 2023.

[47] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *NeurIPS*, 2022.

[48] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[49] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. 2022.

[50] Jian Li, Yabiao Wang, Changan Wang, Ying Tai, Jianjun Qian, Jian Yang, Chengjie Wang, Ji-Lin Li, and Feiyue Huang. DSFD: dual shot face detector, 2018.

[51] Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning. In *ICML*, 2020.

[52] Xin-Yi Li, Wei-Jun Lei, and Yu-Bin Yang. From easy to hard: Two-stage selector and reader for multi-hop question answering, 2022.

[53] Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. What makes convolutional models great on long sequence modeling? 2022.

[54] Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. In *PLDI*, jun 2023.

[55] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1400–1415, 2021.

[56] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 2019.

[57] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning, 2023.

[58] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2024.

[59] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298, 2021.

[60] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. 2019.

[61] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. Sources of hallucination by large language models on inference tasks, 2023.

[62] Matthias Minderer, Alexey Gritsenko, Austin Stone, Maxim Neumann, Dirk Weissenborn, Alexey Dosovitskiy, Aravindh Mahendran, Anurag Arnab, Mostafa Dehghani, Zhuoran Shen, Xiao Wang, Xiaohua Zhai, Thomas Kipf, and Neil Houlsby. Simple open-vocabulary object detection with vision transformers, 2022.

[63] Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. Learning reasoning strategies in end-to-end differentiable proving. In *ICML*, 2020.

[64] 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. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

[65] Sayak Nag, Kyle Min, Subarna Tripathi, and Amit K. Roy Chowdhury. Unbiased scene graph generation in videos, 2023.

[66] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2021.

[67] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with bert, 2019.

[68] Michael O'Connell, Guanya Shi, Xichen Shi, Kamyar Azizzadenesheli, Anima Anandkumar, Yisong Yue, and Soon-Jo Chung. Neural-fly enables rapid learning for agile flight in strong winds. *Science Robotics*, 7(66), May 2022.

[69] OpenAI. Chatgpt plugins, 2023. Accessed: 2024-10-27.

[70] OpenAI. Gpt-4 technical report, 2023.

[71] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat,

Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass,

Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[72] Felix Petersen. Learning with differentiable algorithms. *arXiv preprint arXiv:2209.00616*, 2022.

[73] Amir Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[74] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[75] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[76] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *ICML*, 2021.

[77] Chandan K. Reddy, Lluís Màrquez, Fran Valero, Nikhil Rao, Hugo Zaragoza, Sambaran Bandyopadhyay, Arnab Biswas, Anlu Xing, and Karthik Subbian. Shopping queries dataset: A large-scale esci benchmark for improving product search, 2022.

[78] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022.

[79] Dorsa Sadigh, Eric S Kim, Samuel Coogan, S Shankar Sastry, and Sanjit A Seshia. A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In *53rd IEEE Conference on Decision and Control*, pages 1091–1096. IEEE, 2014.

[80] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter. 2019.

[81] Kengo Sato, Manato Akiyama, and Yasubumi Sakakibara. Rna secondary structure prediction using deep learning with thermodynamic integration. *Nature Communications*, 12(1):941, 2021.

[82] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[83] Xindi Shang, Tongwei Ren, Jingfan Guo, Hanwang Zhang, and Tat-Seng Chua. Video visual relation detection. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1300–1308, 2017.

[84] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[85] Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. CLUTRR: A diagnostic benchmark for inductive reasoning from text. 2019.

[86] Michael F. Sloma and David H. Mathews. Exact calculation of loop formation probability identifies folding motifs in rna secondary structures. *RNA*, 22:1808 – 1818, 2016.

[87] Livio Baldini Soares, Nicholas Fitzgerald, Jeffrey Ling, and Tom Kwiatkowski. Matching the blanks: Distributional similarity for relation learning. In *ACL*, 2019.

[88] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding, 2020.

[89] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, and et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2023.

[90] Jennifer J. Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla-Reyes. Neurosymbolic programming for science, 2022.

[91] Hao Tan and Mohit Bansal. LXMERT: learning cross-modality encoder representations from transformers. 2019.

[92] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. 2020.

[93] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. In *ACL*, 2019.

[94] Anthony Meng Huat Tiong, Junnan Li, Boyang Li, Silvio Savarese, and Steven C.H. Hoi. Plug-and-play VQA: Zero-shot VQA by conjoining large pretrained models with zero training. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the ACL: EMNLP*, December 2022.

[95] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

[96] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. 2017.

[97] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning (ICML)*, 2019.

[98] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[99] Yinjun Wu, Mayank Keoliya, Kan Chen, Neelay Velingker, Ziyang Li, Emily J Getzen, Qi Long, Mayur Naik, Ravi B Parikh, and Eric Wong. Discret: Synthesizing faithful explanations for treatment effect estimation, 2024.

[100] Saining Xie, Chen Sun, Jonathan Huang, Zhuowen Tu, and Kevin Murphy. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *European Conference on Computer Vision (ECCV)*, 2018.

[101] Jingkang Yang, Wenxuan Peng, Xiangtai Li, Zujin Guo, Liangyu Chen, Bo Li, Zheng Ma, Kaiyang Zhou, Wayne Zhang, Chen Change Loy, and Ziwei Liu. Panoptic video scene graph generation, 2023.

[102] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018.

[103] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[104] Youngmin Yi, Chao-Yue Lai, and Slav Petrov. Efficient parallel cky parsing using gpus. *Journal of Logic and Computation*, 24(2):375–393, 2014.

[105] Zhangyue Yin, Yuxin Wang, Yiguang Wu, Hang Yan, Xiannian Hu, Xinyu Zhang, Zhao Cao, Xuanjing Huang, and Xipeng Qiu. Rethinking label smoothing on multi-hop question answering, 2022.

[106] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and

Aravind Machiry. TYGR: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, Philadelphia, PA, August 2024. USENIX Association.

[107] Guangming Zhu, Liang Zhang, Youliang Jiang, Yixuan Dang, Haoran Hou, Peiyi Shen, Mingtao Feng, Xia Zhao, Qiguang Miao, Syed Afaq Ali Shah, et al. Scene graph generation: A comprehensive survey. *arXiv preprint arXiv:2201.00443*, 2022.