Foundations and Trends® in Programming Languages

# Neurosymbolic Programming in Scallop: Principles and Practice

**Suggested Citation:** Ziyang Li, Jiani Huang, Jason Liu and Mayur Naik (2024), "Neurosymbolic Programming in Scallop: Principles and Practice", Foundations and Trends® in Programming Languages: Vol. 8, No. 2, pp 118–249. DOI: 10.1561/2500000059.

**Ziyang Li**
University of Pennsylvania
liby99@seas.upenn.edu

**Jiani Huang**
University of Pennsylvania
jianih@seas.upenn.edu

**Jason Liu**
University of Pennsylvania
jasonhl@seas.upenn.edu

**Mayur Naik**
University of Pennsylvania
mhnaik@seas.upenn.edu



now
the essence of knowledge
Boston — Delft

# Contents

# Neurosymbolic Programming in Scallop: Principles and Practice

Ziyang Li, Jiani Huang, Jason Liu and Mayur Naik

*University of Pennsylvania, USA; liby99@seas.upenn.edu, jianih@seas.upenn.edu, jasonhl@seas.upenn.edu, mhnaik@seas.upenn.edu*

ABSTRACT

Neurosymbolic programming combines the otherwise complementary worlds of deep learning and symbolic reasoning. It thereby enables more accurate, interpretable, and domain-aware solutions to AI tasks. We introduce Scallop, a general-purpose language and compiler toolchain for developing neurosymbolic applications. A Scallop program specifies a suitable decomposition of an AI task's computation into separate learning and reasoning modules. Learning modules are built using existing machine learning frameworks and range from custom neural models to foundation models for language, vision, and multi-modal data. Reasoning modules are specified in a declarative logic programming language based on Datalog which supports expressive features such as recursion, aggregation, negation, and probabilistic programming over structured relations.

Scallop's compiler enables to automatically train neurosymbolic programs in a data- and compute-efficient manner using an end-to-end differentiable reasoning framework. Scallop also supports features useful for building real-world applications such as user-defined data types, and foreign interfaces.

Ziyang Li, Jiani Huang, Jason Liu and Mayur Naik (2024), "Neurosymbolic Programming in Scallop: Principles and Practice", Foundations and Trends® in Programming Languages: Vol. 8, No. 2, pp 118–249. DOI: 10.1561/2500000059.

We demonstrate programming in Scallop for applications that span the domains of image and video processing, natural language processing, planning, and information retrieval in a variety of learning settings such as supervised learning, reinforcement learning, rule learning, contrastive learning, and in-context learning.

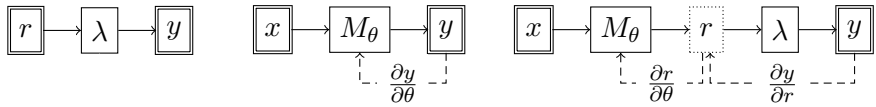
# 1

# 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 $\lambda$ shown in Figure 1.1a is interpretable but operates on limited, structured input $r$. On the other hand, a deep neural network such as $M_\theta$ 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 (Rajpurkar *et al.*, 2016), code completion (Chen *et al.*, 2021), and mathematical problem solving (Lewkowycz *et al.*, 2022), among many others. For instance, code completion requires




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

**Figure 1.1:** Comparison of different paradigms. Logic program $\lambda$ accepts only structured input $r$ whereas neural model $M_\theta$ with parameter $\theta$ can operate on unstructured input $x$. Supervision is provided on data indicated in double boxes. Under *algorithmic supervision*, a neurosymbolic program must learn $\theta$ without supervision on $r$.

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

Neurosymbolic programming is an emerging paradigm that aims to fulfill this goal (Chaudhuri *et al.*, 2021). It seeks to integrate symbolic knowledge and reasoning with neural architectures for better efficiency, interpretability, and generalizability than the neural or symbolic counterparts alone. Consider the task of handwritten formula evaluation (Li *et al.*, 2020), which takes as input a formula as an image, and outputs a number corresponding to the result of evaluating it. An input-output example for this task is $\langle x = \text{1+3÷5}, y = 1.6 \rangle$. A neurosymbolic program for this task, such as the one shown in Figure 1.1c, might first apply a convolutional neural network $M_\theta$ to the input image to obtain a structured intermediate form $r$ as a sequence of symbols [‘1’, ‘+’, ‘3’, ‘/’, ‘5’], followed by a classical algorithm $\lambda$ to parse the sequence, evaluate the parsed formula, and output the final result 1.6.

Despite significant strides in individual neurosymbolic applications (Yi *et al.*, 2018; Mao *et al.*, 2019; Chen *et al.*, 2020; Li *et al.*, 2020; Minervini *et al.*, 2020a; Wang *et al.*, 2019), 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 $\lambda$ 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 $M_\theta$, 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 various advanced features needed for neurosymbolic programming, such as query planning, hardware 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 of Neurosymbolic Solutions

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.

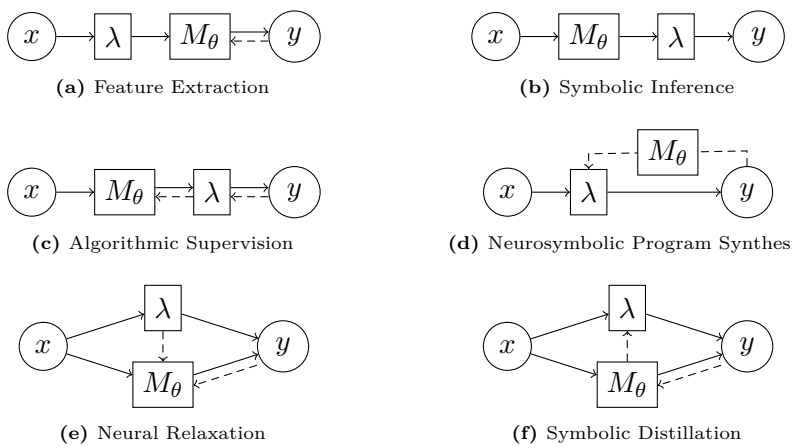



**Figure 1.2:** Neurosymbolic compositions of neural component ($M_\theta$) and symbolic component ($\lambda$), which serve as building-blocks for more complex neurosymbolic applications. We use solid arrows to denote forward data-flows, and dashed arrows to denote backward data-flows used to supervise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

**Feature Extraction** The feature extraction process involves deriving symbolic features from an input $x$ through a symbolic component, denoted here as $\lambda$, before passing these features to a neural model $M_\theta$ 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.

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 (Dinella *et al.*, 2020; Li *et al.*, 2021; Zhu *et al.*, 2024). 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 $M_\theta$ using a symbolic component $\lambda$ 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 $\lambda$ may handle probabilistic information, deriving a distribution rather than just the most likely output.

For instance, in the task of handwritten formula recognition $\langle x = \text{  }, y = 1.6 \rangle$, 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 Section 5, we cover many symbolic inference solutions where the $M_\theta$ are foundation models.

**Algorithmic Supervision** Algorithmic supervision extends symbolic inference by enabling the symbolic component $\lambda$ to propagate learning signals to the neural network $M_\theta$. As before, we assume that $\lambda$ is provided by the programmer. While Figure 1.1 demonstrates one example

of algorithmic supervision through differentiability in $\lambda$, it generally suffices for $\lambda$ to propagate the learning signal. In this way, the symbolic "algorithm" $\lambda$ serves as a guiding supervisor for the neural network $M_\theta$.

Algorithmic supervision also functions as a form of weak supervision, as it does not require direct, fully supervised labels for $M_\theta$; 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 (Li *et al.*, 2020; Li *et al.*, 2023). This tutorial explores additional, advanced examples of algorithmic weak supervision in Section 6.

**Neurosymbolic Program Synthesis** Neurosymbolic program synthesis involves learning the symbolic program $\lambda$ with the support of neural networks. This paradigm resembles the classical syntax-guided synthesis task (Alur *et al.*, 2013), but replaces the traditional algorithmic synthesis procedure with a neural network $M_\theta$. Here, the symbolic program $\lambda$ 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 (Ellis *et al.*, 2022). Traditionally, synthesizing $\lambda$ requires defining a limited domain-specific language (Ellis *et al.*, 2020) 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 $\lambda$ can now be achieved more efficiently (Ma *et al.*, 2024).

**Neural Relaxation** Neural relaxation involves relaxing a deterministic and discrete symbolic reasoning component $\lambda$ by replacing certain components in the pipeline with neural networks $M_\theta$. 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 $M_\theta$, the system gains greater flexibility and responsiveness in handling such scenarios, while being able to learn rapidly (O'Connell *et al.*, 2022; Csomay-Shanklin *et al.*, 2024).

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

This technique has been applied to scientific discovery in fields such as animal behavior analysis (Sun *et al.*, 2022). 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 (Wu *et al.*, 2024). 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 (Silver *et al.*, 2016) 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 (OpenAI, 2023a) 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 Intended Audience

Scallop is built on the logic programming paradigm and integrates seamlessly with machine learning frameworks like PyTorch through Python bindings. As such, we assume readers are familiar with foundational concepts in logic, machine learning, basic calculus (specifically differentiation), and the Python programming language. This tutorial covers topics including programming language syntax and semantics, probabilistic theories and approximations, and the design and implementation of machine learning systems. While it also explores applications in natural language processing and computer vision, we provide accessible introductions to each task. Overall, this tutorial is designed for readers seeking a practical, foundational understanding of neurosymbolic programming with Scallop, covering both theoretical concepts and real-world applications.

## 1.6 Outline

We cover the core Scallop language in Section 2 starting from the basics of relational programming. We then describe our core reasoning module in Section 3 which dives deeper into the internals of Scallop and our provenance framework. We show the core programming constructs in Scallop that allow for scalable differentiable reasoning. Next, Section 4 presents a few motivating tasks showcasing Scallop's ability to concisely and effectively define neurosymbolic applications. Section 5 connects Scallop with foundation models. We present a few more advanced neurosymbolic applications in Section 6. Finally, Section 7 concludes with a discussion of limitations and future directions.

# 2

# Basics of Programming in Scallop

In this section, 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 lines 5 and 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.

```
1 type mother(m: String, c: String),
2      father(f: String, c: String)
3
4 // Christine is Bob's mother
5 rel mother("Christine", "Bob")
6
7 // Bob is father of two kids, Alice and John
8 rel father = {("Bob", "Alice"), ("Bob", "John")}
```

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

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

| 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` |

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 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 pos-

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

**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.

```
// 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.

sible 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".

**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.

```
rel greeting("hello world!")
// or
rel greeting = {("hello world!",)}
// or
rel greeting = {"hello world!"}
```

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

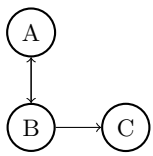



**Figure 2.1:** A graph with three nodes.

```
// An enum type Node
type Node = A | B | C
// Graph relation
rel node = {A, B, C}
rel edge = {(A, B), (B, A), (B, C)}
```

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

**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.

```
> 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.

## 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 (lines 13–14 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

```
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."

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 6) as an 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 (lines 13–14) 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

```
1  // E1: Computing body mass index (BMI) by arithmetic
2  type person(name: String, mass_kg: f32, height_m: f32)
3  rel bmi(name, m / (h * h)) = person(name, m, 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}          // 3 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.

fact is omitted. In the last example shown in Listing 2.8 (lines 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` (lines 4–6) and `path` itself (line 6). Based on this information, we can draw a dependency graph for the

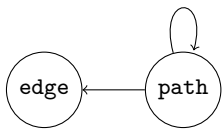



**Figure 2.2:** The dependency graph of the `edge-path` program.

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

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

```
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.

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` $= \text{fib}(x-1)$ and `y2` $= \text{fib}(x-2)$ and compute the sum `y1 + y2`. This almost literally translates to the recursive rule on lines 3–4. 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 (lines 9–10). 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")`, since according to the facts defined above, `"Alice"` is not a parent of anyone.

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

A relation $r$ is *negatively* dependent on $s$ if a negated atom $s$ appears in the body of a rule with head atom $r$. In the 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.

```
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.

```
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 // 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.

### 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 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

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

group-by variable. However, depending on whether we explicitly bound the group-by variable `p`, we get different results. On line 12, 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 1–4.

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

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

// 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.

## 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—Alice cannot be both the mother and father of Alice.

Scallop also supports operators to sample from probability distributions. They share the same surface syntax as aggregations, allowing sampling with group-by. The following rule 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.

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

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

**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.

```
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.

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

```
1  // Grandmother's daughter is 90% likely one's mother
2  // Note: she could also be one's aunt
3  rel 0.9::mother(a,c) =
4      grandmother(a,b) and daughter(b,c)
5
6  // The above rule is desugared to ...
7  rel 0.9::prob_of_rule() // auxiliary nullary relation
8  rel mother(a,c) =
9      grandmother(a,b) and daughter(b,c)
10                      and prob_of_rule()
```

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

cases for 0 and 1, it is expected that the computation for all Fibonacci numbers will never terminate. Such a Scallop program is shown in Listing 2.19. However, often times we have a specific query for these infinite relations. As shown on line 6 in Listing 2.20, we are querying for the 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 every time 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 as 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 rules with on-demand relations as the head atom, their well-formedness is slightly different than regular rules: variables in the positive body atoms as well as the variables bounded by the on-demand head atom are considered bounded by that rule.

On-demand relations can be used to optimize execution of queries. Consider the `edge-path` example shown in Listing 2.21. Suppose we

```
1 type fib(x: i32, y: i32)
2 rel fib = {(0, 1), (1, 1)}
3 rel fib(x, y1 + y2) =
4     fib(x - 1, y1) and fib(x - 2, y2)
5 // 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) =
5     fib(x - 1, y1) and fib(x - 2, y2)
6     and x > 1 // avoid generating infinite demand
7 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 lines 4–6, we also include a constraint `x > 1` in order to bound the recursive generation of demand.

```
1  // Path is declared with on-demand pattern "fb"
2  type Node = usize
3  type edge(x: Node, y: Node), path(x: Node, bound y:
       Node)
4
5  // Dense graph with thousands of edges
6  rel edge = { /* (0, 1), lots of tuples..., (T, S) */ }
7
8  rel path(x, y) = edge(x, y)
9  rel path(x, y) = edge(x, z) and path(z, y)
10 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.

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 11 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). 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 the 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.

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

**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.

```
1  type Expr = Int(i32)          // a simple integer
2            | Add(Expr, Expr) // binary addition
3            | Sub(Expr, Expr) // binary subtraction
4
5  // an expression representing 1 + (3 - 2)
6  const MY_EXPR: Expr = Add(Int(1), Sub(Int(3), Int(2)))
7
8  // a unary relation storing expressions
9  type target_expr(e: Expr)
10 rel target_expr = { MY_EXPR }
```

**Listing 2.22:** A custom algebraic data type `Expr` (lines 1–3) that represents the small language shown in Figure 2.3. Line 6 shows one expression $1 + (3 - 2)$ expressed using the `Expr` type.

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.

```
1  // eval relation evaluates the expr, 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) =
7      eval(e1, i1) and eval(e2, i2)
8  rel eval(Sub(e1,e2), i1-i2) =
9      eval(e1, i1) and eval(e2, i2)
10
11 // query the result of MY_EXPR
12 query eval(MY_EXPR, y)
```

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

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 12 (`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 Section 5. 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 declaration of externally defined items. However, during normal use of Scallop, such declarations are not necessary 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 a 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.

```
// 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 an
// integer range. Note the end index `e` is optional;
// if not provided, we retrieve the part of the 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 "..."
extern type $string_concat(strs: String...) -> String

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

**Listing 2.24:** Type declarations of foreign functions from Scallop's standard library.

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 values of any type to be passed into the function.

When using a generic function, it is not necessary 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"`.

**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 knobs 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 information 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.

```
1 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 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 computes the probability of the two numbers being equal based on their distance. Formally, it is defined as follows:

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

```
// Given a string, produce 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`.

```
// Given two floating point numbers, compute the
// probability that 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.

Essentially, we have a parameter $\beta$ 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 a 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 Section 5.

### 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 start with an `@` sign, and can accept arbitrary arguments, both positional and keyworded. Conceptually, one may think of attributes as taking in the annotated item, and returning another item.

The following example in Listing 2.28 shows the use of a `@file` attribute to annotate a relation named `edge`. Specifically, it instructs 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.

```
1 // [edge.csv]
2 from,to
3 0,1
4 1,2
```

**Listing 2.27:** A CSV file storing edges.

```
1 // [edge_path.scl]
2 @file("edge.csv", header=true)
3 type edge(from: u32, to: u32)
4 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.

In the standard library of Scallop, there are many existing predefined 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. As another example, `@cmd_arg` retrieves command-line arguments (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 Section 5.

# 3

# Core Reasoning Framework

The preceding section 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 section, we formally define the semantics of the language by means of a provenance framework. We show how Scallop uniformly supports different reasoning modes—discrete, probabilistic, and differentiable—simply by defining different provenances.

We start by presenting the basics of our provenance framework (Section 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* (Green *et al.*, 2007). Figure 3.1 defines Scallop's algebraic interface for

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

**Figure 3.1:** Core algebraic interface for provenance $T$.

provenance. We call the additional information a *tag* $t$ from a *tag space* $T$. There are two distinguished tags, $\mathbf{0}$ and $\mathbf{1}$, representing unconditionally *false* and *true* tags. Tags are propagated through operations of binary *disjunction* $\oplus$, binary *conjunction* $\otimes$, and unary *negation* $\ominus$ resembling logical *or*, *and*, and *not*. Lastly, a *saturation* check $\Subset$ serves as a customizable stopping mechanism for fixed-point iteration. 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 3.1.** `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 section.

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., $t_1 \oplus (t_1 \otimes t_2) = t_1$ (Dannert *et al.*, 2021). 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.

## 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 ($\gamma_g$ with aggregator $g$), projection ($\pi_\alpha$ with mapping $\alpha$), and selection ($\sigma_\beta$ with condition $\beta$), 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.

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

**Figure 3.2:** Abstract syntax of core fragment of SclRam.

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 ($\gamma$) 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

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

**Figure 3.3:** Semantic domains for SclRam.

under a relational predicate $p$. Tuples and facts can be tagged, forming *tagged tuples* ($t :: u$) and *tagged facts* ($t :: p(u)$). Given a set of tagged tuples $U_T$, we say $U_T \vDash u$ iff there exists a $t$ such that $t :: u \in U_T$. A set of tagged facts form a database $F_T$. We use bracket notation $F_T[p]$ to denote the set of tagged facts in $F_T$ under predicate $p$.

## 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 $\beta$, and projection transforms tuples according to mapping $\alpha$. The mapping function $\alpha$ 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

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

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

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

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

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

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

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

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

(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** $\mathbf{lfp}^\circ : (\mathcal{F}_T \to \mathcal{F}_T) \to (\mathcal{F}_T \to \mathcal{F}_T), \quad [\![s]\!], [\![\overline{s}]\!] : \mathcal{F}_T \to \mathcal{F}_T$

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

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

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

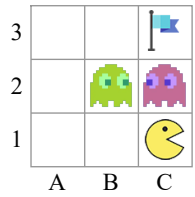
**Figure 3.4:** Operational semantics of core fragment of SclRam.

(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.

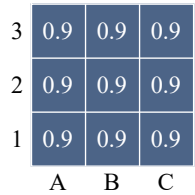
**Difference and Negation.** To evaluate a difference expression $e_1 - e_2$, there are two cases depending on whether a tuple $u$ evaluated from $e_1$ appears in the result of $e_2$. If it does not, we simply propagate the tuple and its tag to the result (DIFF-1); otherwise, we get $t_1 :: u$ from $e_1$ 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 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




(a) Maze illustration




(b) `grid_cell`

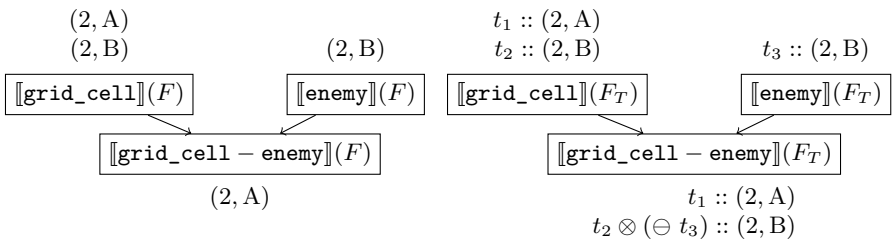

(c) `enemy`

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

↓

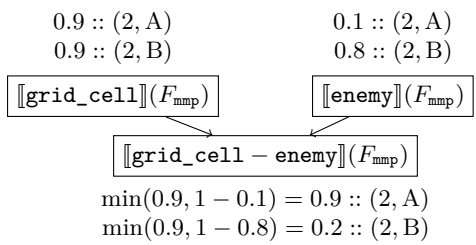
ScLRam Code: `safe_cell` ← `grid_cell` − `enemy`

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



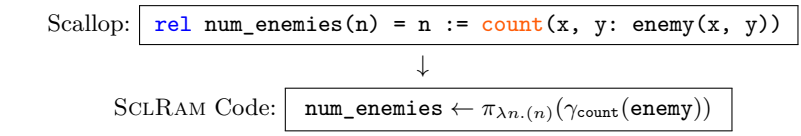

(e) Untagged semantics

(f) ScLRam tagged semantics

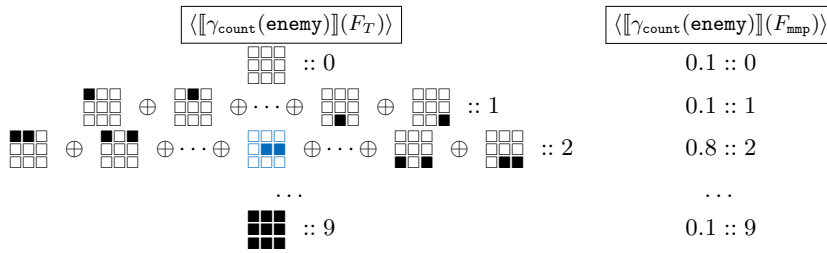



(g) ScLRam with `max-min-prob`

**Figure 3.5:** An example maze configuration is shown in (a), where each cell is represented by a tuple like $(1, \mathrm{A})$. 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.

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 (Abiteboul *et al.*, 1995). The iteration happens on a per-stratum basis to enforce stratified negation and aggregation. Evaluating a single step of stratum $s$ means evaluating all the rules in $s$ and returning the updated database. Note that we define a specialized

```
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 $F^{(i)}_T$ | – | $\boxed{\uparrow}$ | = | $\boxed{\uparrow} \oplus \boxed{\uparrow} \oplus \cdots \oplus \boxed{\uparrow}$ | = | $\boxed{\uparrow} \oplus \boxed{\uparrow} \oplus \cdots \oplus \boxed{\uparrow}$ | = |
| $t^{(i)}_{\texttt{1C-3C}}$ 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 $\boxed{\uparrow}$ to represent a conjunction of negated tags of `enemy` along the illustrated path, e.g. $\boxed{\uparrow} = (\ominus t_{\texttt{2C}}) \otimes (\ominus t_{\texttt{3C}})$. 2nd iter is the first time $t_{\texttt{1C-3C}}$ is derived, but the path $\boxed{\uparrow}$ is blocked by an enemy. On 6th iter, the best path $\boxed{\uparrow}$ 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.

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 $(I, O, \tau, \rho)$ the *external interface* for a provenance $T$. The whole execution pipeline is then illustrated in Figure 3.8.


**Figure 3.8:** Execution pipeline with external interface.

In the context of a Scallop application, an EDB is provided in the form $F_{\texttt{option<I>}}$. During the *tagging phase*, $\tau$ 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*, $\rho$ 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 $\rho$ is only applied to tags under such relations to avoid redundant computations.

**Example 3.2.** The external interface of the `max-min-prob` provenance from Example 3.1 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 $id := \lambda x.x$.

## 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 3.1 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.

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

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

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 $\phi$. 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, whereas $\bot$ is an empty DNF formula. As such, `proofs-prob` is formally defined as follows:

**Definition 3.1.** 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 $([0,1], [0,1], \tau_{\texttt{pp}}, \rho_{\texttt{pp}})$ where:

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

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

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 ($p_i$).

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 $\tau_{\texttt{pp}}$ transforms the input probabilities into internal tags. After the derivation of boolean formulas, we apply recovery function $\rho_{\texttt{pp}}$ 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.

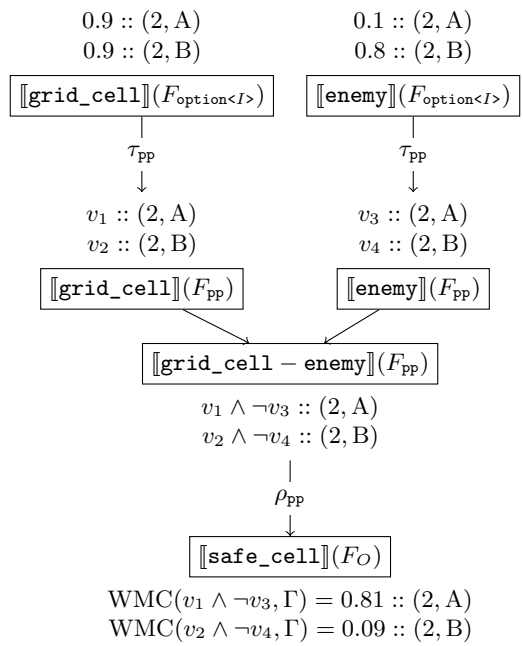



**Figure 3.10:** The ScLRam evaluation result with `proofs-prob` on the rule shown in Figure 3.5. As shown in the first row, we assume that facts in `grid_cell` and `enemy` are provided as base facts with given probabilities. Therefore, each of the 4 shown facts on the second row is assigned a unique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

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.

## 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 $\phi$ 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:

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

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

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.

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

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

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

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

// R2: query objects that are 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.

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 as follows:

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

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

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.

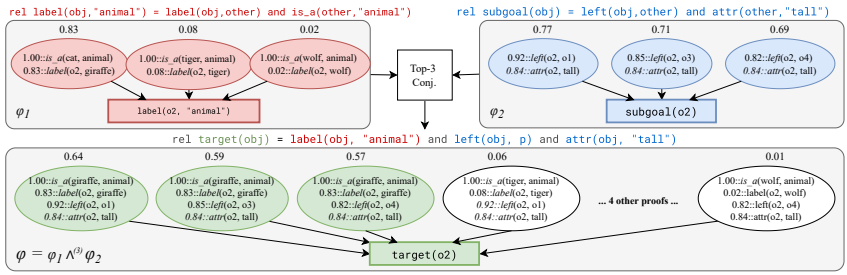



**Figure 3.12:** Illustration of top-$k$ conjunction using $k = 3$. Each ellipse represents a proof of the fact shown in the box. Given the top 3 proofs for each of "$\texttt{label}(o_2, \texttt{"animal"})$" and "$\text{subgoal}(o_2)$", we wish to derive the top 3 proofs for their conjunction, "$\text{target}(o_2)$". 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).

To take negation $\neg^{(k)}$ on DNF $\varphi$, 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 follows:

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

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 `proofs-prob` 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 $\phi$ 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, $\nabla y_j$. From here, we can obtain our expected output $\vec{y}$ and $\nabla \vec{y}$ by stacking together $y_j$'s and $\nabla y_j$'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.

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

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

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

**Figure 3.13:** Operations on dual-number $\mathbb{D} \triangleq [0,1] \times \mathbb{R}^n$, where $n$ is the number of input probabilities. The function $\text{clamp}_0^1$ clamps its input into the range $[0,1]$.

| **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` | $\Phi$ | $\bot$ | $\top$ | $t_1 \vee^{(k)} t_2$ | $t_1 \wedge^{(k)} t_2$ | $\neg^{(k)}\, t$ | $t_1$ == $t_2$ | $v_i$ | $\text{WMC}(t, \Gamma)$ |

**Figure 3.14:** Definitions of three differentiable provenances.

### 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* semiring originally proposed in Huang *et al.* (2021) 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 Manhaeve *et al.*, 2021. Other than the boolean formula $\varphi$, 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) (Darwiche, 2011) 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.

### 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 remove such facts early. 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 a mutually exclusive set of probabilistic facts (Listing 2.16). Mutual exclusivity of facts is *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 `proofs-prob` 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 `proofs-prob` as an example to show how it can be extended to handle mutual exclusivity. In Scallop, `proofs-prob` (along with others like top-$k$ proofs) are already extended with this functionality. But for presentation purpose, we consider a new provenance, named `proofs-prob-me`, where `me` stands for *mutual exclusion*. In `proofs-prob-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, $I_{\texttt{proofs-prob-me}} = [0, 1] \times \texttt{option<}\mathbb{N}\texttt{>}$.

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 prob-

```
1 rel color = {0.9::(OBJ_A, "red");
2              0.1::(OBJ_A, "green")}
3 rel color = {0.2::(OBJ_B, "red");
4              0.8::(OBJ_B, "green")}
5
6 rel should_not_exist(obj) =
7         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 lines 6–7 should result in an empty relation, if the mutual exclusions are properly handled.

ability, while the mapping from this fact ID to the mutual exclusion ID is stored for future reference. When executing the rule (lines 6–7), 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. `proofs-prob` provenance will reject such a proof, rendering the result tag to be **0**. Thus, combined with the early removal feature, the `should_not_exist` relation is computed to be empty, as desired.

### 3.8.3 Specializing for Provenances

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.

**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))                                        */
```

$\mathbf{t}^{\text{pos}} = sorted([t_i \mid i = 1 \ldots n])$;

$\mathbf{t}^{\text{neg}} = [1 - t^{\text{pos}}_{n-i+1} \mid i = 1 \ldots n]$;

```
/* Iterate through all possible partitions between positive and
   negative tags. O(n)                                         */
```

$U'_{\texttt{mmp}} = \{t^{\text{neg}}_n :: 0, t^{\text{pos}}_1 :: n\}$ ;

**for** $i = 1 \ldots (n-1)$ **do**

  Add $\min(t^{\text{pos}}_{i+1}, t^{\text{neg}}_i) :: (n-i)$ to $U'_{\texttt{mmp}}$;

**return** $U'_{\texttt{mmp}}$

### 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.

# 4

# Scallop in Practice: End-to-End Examples

In this section, we present end-to-end examples showcasing how to use Scallop to do Neurosymbolic learning on relatively simple tasks. For each case study we present the comprehensive problem setup, Scallop code, and the end result.

## 4.1 Summing Two MNIST Digits

The `MNIST-Sum2` task from Manhaeve *et al.* (2021) concerns classifying sums from pairs of handwritten digits, e.g.,  +  = 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 4.1, we specify this task using a neural and a symbolic component, following the style of DeepProbLog (Manhaeve *et al.*, 2021). The neural component is a perception model that takes in an image of a handwritten digit (Lecun *et al.*, 1998) 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 $0.85 :: \texttt{d}(\texttt{3}, 3)$ denotes

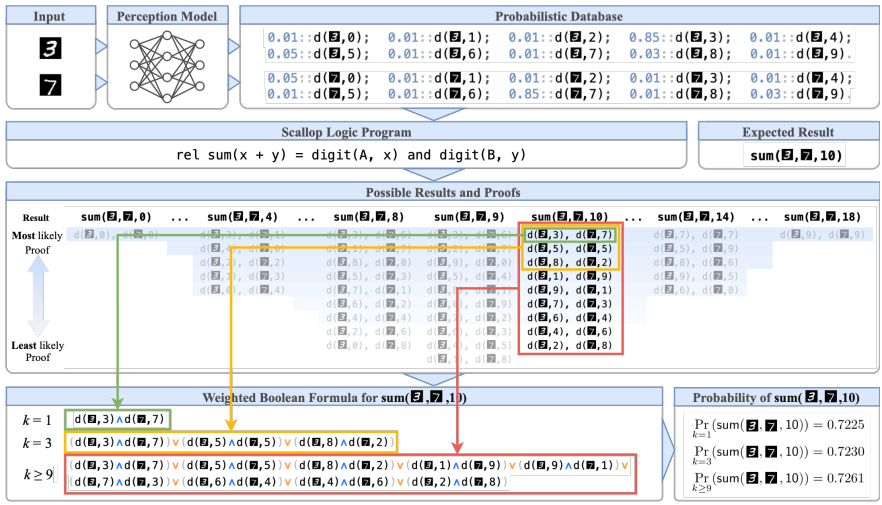



**Figure 4.1:** Illustration of our approach on the task 3 + 7 = 10 using different values of parameter $k$.

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 of Figure 4.1 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 :: `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 in Section 3.6, 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 4.1. The `mnist_net` (line 5) is the neural network that classifies individual MNIST image into a distribution of 10 classes, while `sum_2` (lines 7–15) 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 4.1. Lines 12–14 tells how to turn input distributions into relational symbols (and vice versa for the output). Line 15 configures the provenance to use for the reasoning, which is the `dtkp` provenance with $k = 1$.

```
1  class MNISTSum2Net(nn.Module):
2    def __init__(self):
3      super(MNISTSum2Net, self).__init__()
4      # Classic MNIST Digit Recognition module
5      self.mnist_net = MNISTNet()
6      # Scallop reasoning module
7      self.sum_2 = scallopy.Module(
8        program="""
9          type digit_1(i32), digit_2(i32)
10         rel sum(a + b) = digit_1(a) and digit_2(b)
11       """,
12       input_mappings={"digit_1": range(10),
13                       "digit_2": range(10)},
14       output_mappings={"sum_2": range(19)},
15       provenance="diff-top-k-proofs", k=1)
16
17   def forward(self, x: Tuple[Tensor, Tensor]):
18     # Input images; shape: (batch_size, 27, 27, 1)
19     (a_imgs, b_imgs) = x
20     # Classify first digit; shape: (batch_size, 10)
21     a_distrs = self.mnist_net(a_imgs)
22     # Classify second digit; shape: (batch_size, 10)
23     b_distrs = self.mnist_net(b_imgs)
24     # Run Scallop; result shape: (batch_size, 19)
25     return self.sum_2(digit_1=a_distrs,
26                       digit_2=b_distrs)
```

**Listing 4.1:** The Scallop code for the `MNIST-Sum2` learning task.

During inference, as shown in the `forward` function (lines 17–26), 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 complete. All the algorithmic and differentiation details of Scallop is hidden from the user, providing a clean programming interface.

## 4.2 Evaluating Handwritten Formulas

In this case study we take the `MNIST-Sum2` one step further by allowing multiple symbols including handwritten digits and also simple arithmetic operators like $+$, $-$, $\times$, and $\div$. This is the task of handwritten formula evaluation (HWF) (Li *et al.*, 2020). The input to the task is a sequence of images of handwritten symbols, forming a handwritten formula. One such example is given in Figure 4.2. 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 simplicity, we assume that the input formulas always are parsed and free of divide-by-zero errors.

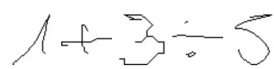


**Figure 4.2:** One handwritten formula $1 + 3 \div 5$ which should evaluate to 1.6.

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* (Petersen, 2022).

The reasoning component is written in Scallop as shown in Listing 4.2. The program uses Datalog-like syntax. It specifies two input relations, `symbol` and `length` (lines 2-5). The former 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 12–14), `term` (lines 17–22), and `expr` (lines 25–30), 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 33), and store the evaluated result in the `result` unary relation.

Next, we may integrate this program into an end-to-end learning pipeline. Listing 4.3 shows the PyTorch module for the HWF task. During initialization, we setup the CNN to process each symbol image (line 7). Then we create a Scallop module to load the program from file `hwf.scl` (lines 9–17). 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 (lines 21–22) and the result distributions to our Scallop module (lines 23–24). 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's reasoning module needs to accept formulas of varying lengths. In PyTorch, this 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.

```
1  // [hwf.scl]
2  // Input: probabilistic symbols
3  type symbol(index: usize, symbol: String)
4  // Input: length of the formula
5  type length(n: usize)
6
7  // Helper relation
8  rel digit = {"0","1","2","3","4","5","6","7","8","9"}
9
10 // Parsing and evaluating the sequence of symbols
11 // A single number
12 type factor(value: f32, begin: usize, end: usize)
13 rel factor(x as f32, b, b + 1) =
14     symbol(b, x) and digit(x)
15
16 // A mult/div expression
17 type term(value: f32, begin: usize, end: usize)
18 rel term(x, b, r) = factor(x, b, r)
19 rel term(x * y, b, e) = term(x, b, m) and
20     symbol(m, "*") and factor(y, m + 1, e)
21 rel term(x / y, b, e) = term(x, b, m) and
22     symbol(m, "/") and factor(y, m + 1, e)
23
24 // An add/minus expression which has higher precedence
25 type expr(value: f32, begin: usize, end: usize)
26 rel expr(x, b, r) = term(x, b, r)
27 rel expr(x + y, b, e) = expr(x, b, m) and
28     symbol(m, "+") and term(y, m + 1, e)
29 rel expr(x - y, b, e) = expr(x, b, m) and
30     symbol(m, "-") and term(y, m + 1, e)
31
32 // Obtain the result
33 rel result(y) = expr(y, 0, l) and length(l)
```

**Listing 4.2:** Formula evaluator for the HWF task in Scallop.

To process this tensor, we setup the `input_mapping` (lines 12–15) 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 17).

```
class HWFNet(nn.Module):
  def __init__(self):
    MAX_LEN = 7
    ALPHABET = ["0", ..., "9", "+", "-", "*", "/"]
    ...
    # Symbol recognition module
    self.symbol_cnn = SymbolNet()
    # 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]
    symbol = self.symbol_cnn(img_seq.flatten(0, 1))
                 .view(len(length), -1)
    (out_symbols, out_distr) = self.eval_formula(
        symbol=symbol, length=length)
    return (out_symbols, out_distr)
```

**Listing 4.3:** PyTorch module for the HWF task with Scallop.

**The need for symbol sampling.** The input space for the HWF task 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 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 the 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 23, 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 the computed distribution over the results in `out_symbols`. We present in Listing 4.4 the loss function that is used to process the resulting output.

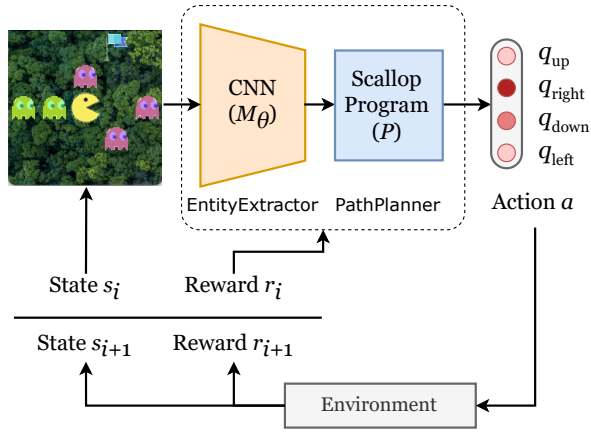
```
1  # calling hwf_net yields the set of outputs as well as
2  # the predicted distributions over the set of outputs
3  (outputs, y_pred) = hwf_net(formula_imgs)
4
5  # construct a ground truth tensor y based on the
6  # labels and the set of produced outputs
7  y = torch.tensor([
8    [1.0 if abs(l-m) < 0.001 else 0.0 for m in outputs]
9    for l in labels])
10
11 # compute the binary cross entropy loss
12 loss = binary_cross_entropy(y_pred, y)
```

**Listing 4.4:** 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.

## 4.3 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 4.3a, concerns an intelligent agent realizing a sequence of actions in a simplified version of the PacMan maze game. The maze is an implicit $5 \times 5$ grid of cells. Each cell is either empty or has an entity,


**(a)** Three states of one gameplay session.




**(b)** Architecture of application with Scallop.

**Figure 4.3:** Illustration of a planning application PacMan-Maze in Scallop.

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

Concretely, the game is modeled as a sequence of interactions between the agent and an environment, as depicted in Figure 4.3b. The game state $s_i \in S$ at step $i$ is a $200 \times 200$ colored image ($S = \mathbb{R}^{200 \times 200 \times 3}$). 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 (Watkins, 1989). Its goal is to learn a function $Q : S \times A \to \mathbb{R}$ that returns the expected reward of taking action $a_i$ in state $s_i$.[1] Since the game states are images, we employ Deep $Q$-Learning (Mnih *et al.*, 2015), which approximates the $Q$ function using a convolutional neural network (CNN) model with learned parameter $\theta$. An end-to-end deep learning based approach for our application involves training the model to predict the $Q$-value of each action for a given game state. This approach takes 50K training episodes to achieve a 84.9% test success rate, where a single episode is one gameplay session from start to end.

In contrast, a neurosymbolic solution using Scallop only needs 50 training episodes to attain a 99.4% test success rate. Scallop enables to realize these benefits of the neurosymbolic paradigm by decomposing the agent's task into separate neural and symbolic components, as shown in Listing 4.3b. 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 **??** 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 Listing **??**. 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

[1] 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 4.5:** Snippet of implementation in Python.

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 Listing 4.6. We next review three key design decisions of Scallop with respect to this program.

**Relational Model.** In Scallop, the primary data structure for representing symbols is a *relation*. In our example, the game state can be symbolically described by the kinds of entities that occur in the discrete cells of a $5 \times 5$ 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 (lines 14–15 of Listing 4.5). Scallop therefore allows to associate tuples with probabilities, e.g. 0.96 ::

```
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) =
11     grid_cell(x, y) and not enemy(x, y)
12 rel edge(x, y, x, yp, UP) = safe_cell(x, y) and
13                             safe_cell(x, yp) and
14                             yp == y + 1
15 // Rules for DOWN, RIGHT, and LEFT edges omitted
16
17 rel next_pos(p, q, a) =
18     actor(x, y) and edge(x, y, p, q, a)
19 rel path(x, y, x, y) = next_pos(x, y, _)
20 rel path(x1, y1, x3, y3) = path(x1, y1, x2, y2) and
21                            edge(x2, y2, x3, y3, _)
22 rel next_action(a) = next_pos(p, q, a) and
23                      path(p, q, r, s) and
24                      goal(r, s)
```

**Listing 4.6:** The logic program of the PacMan-Maze application in Scallop.

`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 4.5), 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 4.6. 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's 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 (lines 22–24) computing all such actions.[2]

The arguably most complex concept is the `path` relation which is recursively defined on lines 19–21. Recursion allows to define the pattern succinctly, enables the trained application to generalize to grids arbitrarily larger than $5 \times 5$ 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 Listing 4.5 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.

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

# 5

# Programming with Foundation Models

## 5.1 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 (Bommasani *et al.*, 2021). Exemplars of foundation models include *language models* (LMs) like GPT (Bubeck *et al.*, 2023), *vision models* like Segment Anything (Kirillov *et al.*, 2023), and *multi-modal models* like CLIP (Radford *et al.*, 2021). 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 (McKenna *et al.*, 2023). 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 (Radford *et al.*, 2021) for visual question answering.

Various mechanisms have been proposed to augment foundation models to overcome these limitations. For example, PAL (Gao *et al.*, 2023), WebGPT (Nakano *et al.*, 2021), and Toolformer (Schick *et al.*, 2023) connect LMs with search engines and external tools, expanding

```
@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 5.1:** Two example programs in Scallop using foundation models.

their information retrieval and structural reasoning capabilities. LMQL (Beurer-Kellner *et al.*, 2022) generalizes pure text prompting in LMs to incorporate scripting. In the domain of computer vision, neurosymbolic visual reasoning frameworks such as VisProg (Gupta and Kembhavi, 2022) compose diverse vision models with LMs and image processing subroutines. Despite these advances, programmers 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 5.1a shows a Scallop program which invokes GPT to extract the height of mountains whose names are specified in a struc-

```
@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 5.1:** 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`.

tured table. Likewise, the program in Figure 5.1b uses the image-text alignment model CLIP to classify images into discrete labels such as `cat` and `dog`. Figure 5.1c shows relational input-output examples for the two programs. Notice that the CLIP model also outputs probabilities that allow for probabilistic reasoning.

## 5.2 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 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 5.1 illustrates one succinct implementation of the FA that enables the use of the CLIP model shown in Figure 5.1b.

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.

## 5.3 Large Language Models

**Text completion.** In Scallop, language models like GPT (OpenAI, 2023b) and LLaMA (Touvron *et al.*, 2023) can be used as basic foreign predicates for text completion (Listing 5.2). 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.

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

**Listing 5.2:** A snippet of Scallop using `gpt` as a foreign predicate.

To make the interface more relational and structural, we provide an FA for better specification of prompts, as shown in Listing 5.3. 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 prompt with the given location at the *bound* argument `{{loc}}` and invokes GPT to fill in the *free* argument `{{num}}`.

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

**Listing 5.3:** A snippet of Scallop using `gpt` as a foreign attribute.

Consider the Scallop program in Listing 5.4. 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

```
@gpt(
  "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 5.4:** A snippet of Scallop using `@gpt` for querying mountain heights.

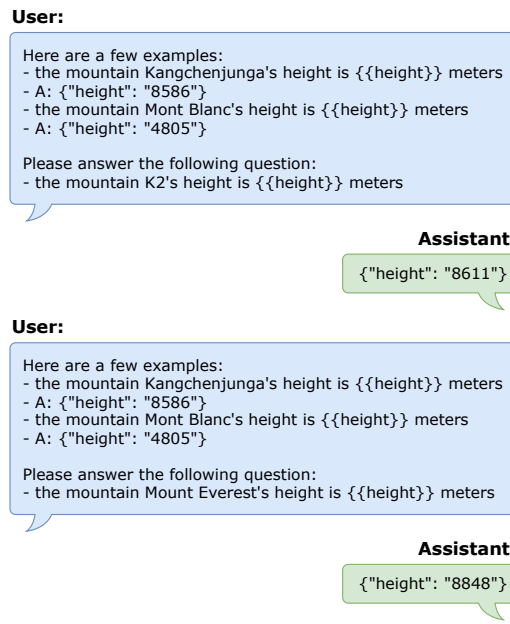



**Figure 5.2:** Conversation history between User (messages generated by `gpt` FA) and GPT-4 (`gpt-4-0613`) via OpenAI API after executing the program in Listing 5.4.

value of `{{height}}` for each mountain. The shots provided in `examples` modify the prompt to GPT-4 as shown in Figure 5.2.

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 5.5 takes in a `context` and produces `(subject, object, relation)` triplets.

```
@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 5.5:** A snippet of Scallop using `gpt_extract_relation` as a foreign attribute.

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 (Srivastava *et al.*, 2023). 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?
>
> **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.

Assuming that the above relations are supplied with complete and accurate facts, the Scallop rules in Listing 5.6 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 5.7 to define the GPT-4 prompt for extracting the three relations `mentioned_date`, `goal`, and `relationship` in Listing 5.8 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.

```
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 5.6:** Scallop logic rules for the date understanding task.

```
@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 5.7:** 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 5.8:** Scallop rules for extracting 3 relations from a question for date understanding via the FA-annotated rules of Listing 5.7.

Referring to Listing 5.7, 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 5.8. The shots provided in `examples` (lines 7–19) are formatted as messages that are prepended to the conversation history given to GPT-4, as seen in Figure 5.3. Finally, the parameter `cot` (line 20) is a Boolean array where `cot[i]` toggles whether the $i$th relation should be extracted using zero-shot chain-of-thought (CoT) prompting (Kojima *et al.*, 2022).

Now suppose the bound argument `question` has value:

> Jane finished her PhD in January 5th, 2008. Today is the 10th anniversary. What is the date 10 days ago?

The GPT-4 conversation history after executing the code in Listing 5.7 and Listing 5.8 is given by Figure 5.3. With a little thought, the reader will find that applying the rules in Listing 5.6 on the relations generated by GPT-4 in Figure 5.3 will yield the correct answer: 12/26/2017.

This example points towards a general pattern for programming neurosymbolically 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 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.

## 5.4 Embedding Models and Vector Databases

Textual embeddings are useful in performing tasks such as information retrieval. In Scallop, embedding models are usually modeled as foreign predicates. Listing 5.9 declares an FP encapsulating a cross-encoder (Nogueira and Cho, 2019).

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"`.

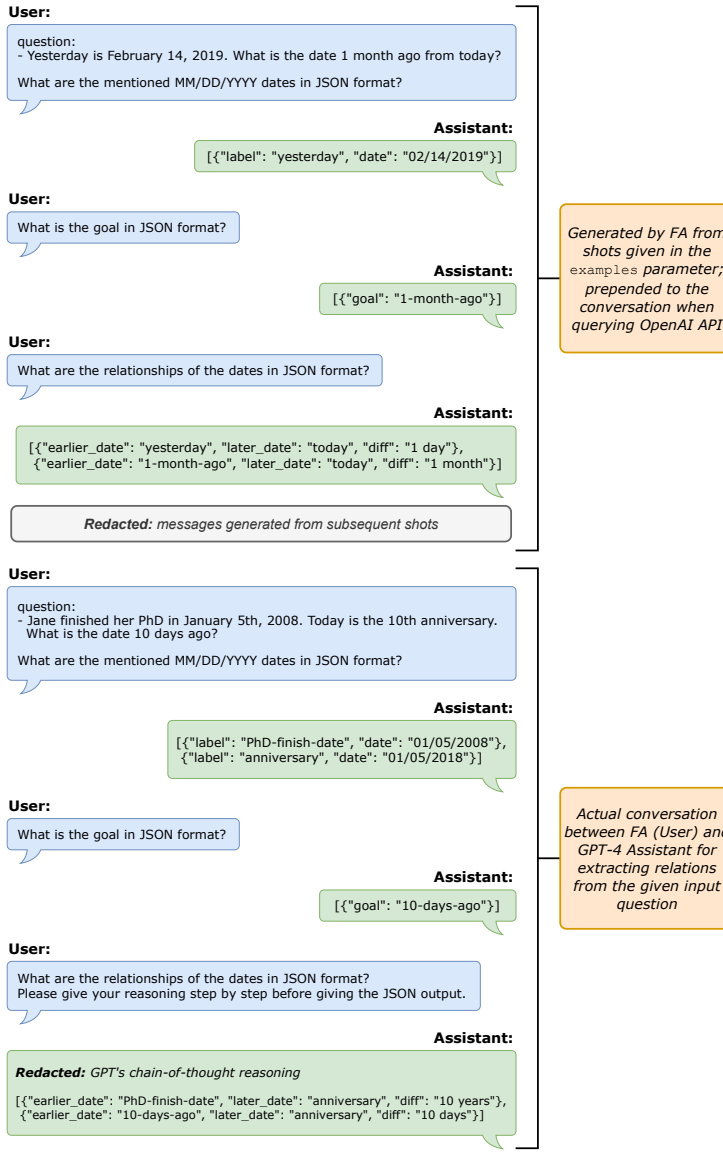



**Figure 5.3:** The GPT-4 conversation history after executing the program in Listing 5.7, with annotations and redactions in italics.

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

**Listing 5.9:** Scallop snippet using `cross_encoder` as a foreign attribute.

One application of these techniques is in information retrieval. For example, consider the task from HotpotQA (Yang *et al.*, 2018). In this Wikipedia-based question answering (QA) dataset, the model takes 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 5.10, we implement an adaptation of FE2H (Li *et al.*, 2022). 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 17–22). By retrieving only 2 documents, the context we generate is inherently less distracting than the naive context of all 10 documents.

## 5.5 Vision and Multi-Modal Models

**Image classification models.** Image-text alignment models, such as CLIP (Radford *et al.*, 2021), can be used off-the-shelf as zero-shot image classification models. Figure 5.1b 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 (Minderer *et al.*, 2022), Segment Anything Model (SAM) (Kirillov *et al.*, 2023), and Dual-Shot Face Detector (DSFD) (Li *et al.*, 2018) 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.

```
@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 5.10:** Scallop program for information retrieval in the HotpotQA task.

For instance, the OWL-ViT model can be used and configured as shown in Listing 5.11. 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, label, and cropped image. Note that each produced fact is tagged with a probability, representing the model's confidence.

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

**Listing 5.11:** Scallop snippet using `owl_vit` as a foreign attribute.

**Image generation models.** Visual generative models such as Stable Diffusion (Rombach *et al.*, 2022) and DALL-E (Ramesh *et al.*, 2021) can be regarded as relations as well.

Listing 5.12 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.

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

**Listing 5.12:** Scallop snippet using `stable_diffusion` as a foreign attribute.

**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 (Gupta and Kembhavi, 2022). 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 5.4.

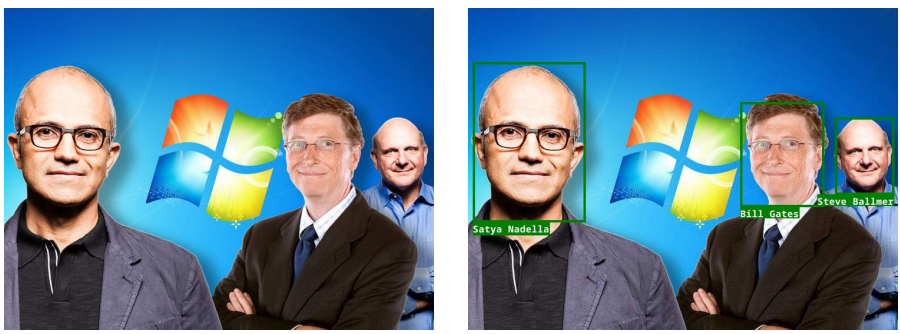



**Figure 5.4:** The face-tagging input (left) and output (right) of the image with descriptive filename `microsoft_ceos.jpeg`.

```
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: labeling identified faces w/ boxes
```

**Listing 5.13:** Scallop program for face-tagging.

The code for face-tagging is provided in Listing 5.13. Our solution obtains a set of possible names from GPT-4 (lines 5–13) and candidate faces from the DSFD face detection model (lines 15–27). These are provided to CLIP for object classification (lines 29–32), after which probabilistic reasoning filters the most relevant face-name pairs (lines 34–38). 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).

# 6

# Advanced Applications

Neurosymbolic methods decompose a problem into two core components: neural perception and logical deduction to enjoy the benefits of both deep learning and traditional algorithms. In real-world scenarios, Scallop users may be particularly interested in understanding when and how to apply this paradigm. This section is designed to address these queries, providing targeted insights into the practical application of neurosymbolic methods by multiple complex tasks.

Determining the optimal division of neural and logical components is fundamental to applying Scallop effectively. When given a task, one may start by envisioning its solution through a purely neural approach, then assess how it would be tackled using only logical methodologies. These two perspectives serve as extreme baselines. Following this, select a suitable intermediate representation that allows for seamless extraction of perceptual inputs and efficient neurosymbolic learning.

In this section, we cover three tasks where the neurosymbolic paradigm improves upon the previous state-of-the-art baseline. Due to the tasks' complexity, we elide the end-to-end Python and Scallop code. Rather, we focus on the conceptual advancements that each application brings, such as unique symbolic representation of unstructured

data, special logical reasoning patterns, and newly adapted learning paradigms. Specifically, we discuss the following topics for each task:

**Understanding the Input Dataset** A detailed examination of the nature and structure of datasets appropriate for neurosymbolic methods, setting the stage for effective processing by Scallop.

**Choosing the Structured Representation** Discussion of the intermediate representations that Scallop utilizes to seamlessly integrate logical reasoning with perceptual data.

**Neural Architecture and Scallop Programs** We provide the neural network architectures utilized for each task and describe how these are integrated into Scallop programs.

**Performance Metrics** Evaluation of the improvements and efficiencies brought by adopting the neurosymbolic paradigm.

## 6.1 Learning Composition Rules for Kinship Reasoning

**CLUTRR** (Sinha *et al.*, 2019) 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 explicitly mentioned in the story.

We showcase one CLUTRR example in Figure 6.1. 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

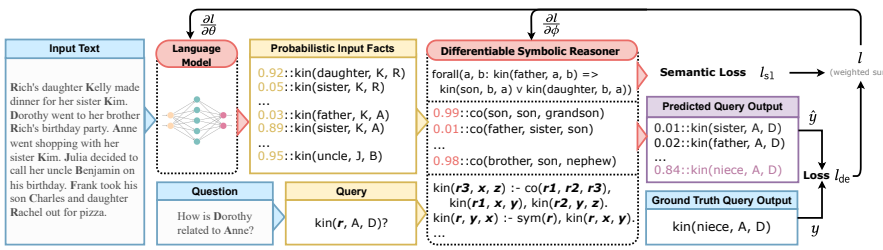



**Figure 6.1:** Overview of kinship reasoning with an example where "Anne is the niece of Dorothy" can be inferred from the context. We abbreviate the names with their first initials in the relational symbols, and the composite relationship with "co".

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 $k \in \{2, \ldots, 10\}$.

### 6.1.1 Structured Representation: Family Graph

One natural representation of kinship is the family graph, as shown in Figure 6.2. 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 with facts.

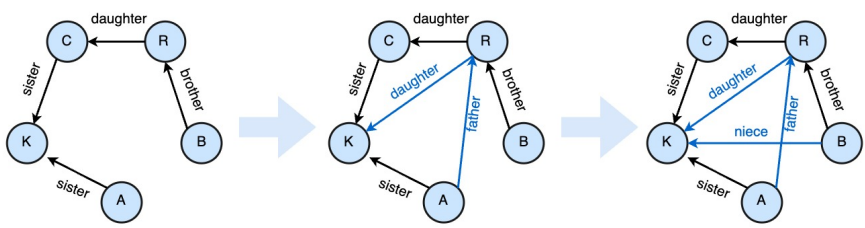



**Figure 6.2:** The family graph corresponding to the story shown in Figure 6.1. Edges representing family relations directly extracted from the story are colored in black, while those requiring derivation using common sense knowledge are colored in blue. Additionally, names are abbreviated using their initials.

Logic rules can be applied to known facts to deduce new ones. For example, below is a Horn 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 6.1.

**Table 6.1:** Higher-order predicate examples.

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

The logic for reasoning over kinship relations is realized in Scallop in Listing 6.1. Line 2 declares 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 declare on lines 7–9 the rule that composes 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 predefined 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

```
// Relation declarations
type kinship(rela: String, sub: String, obj: String)
type composite(r1: String, r2: String, r3: String)
type question(sub: String, obj: String)

// Rules to derive the final answer
rel kinship(r3,a,c) = kinship(r1,a,b) and
                      kinship(r2,b,c) and
                      composite(r1,r2,r3) and a != c
rel answer(r) = question(s, o), kinship(r, s, o)

// Integrity constraints:
// (6 for kinship and 2 for rule learning)
rel violate(!r) = r :=
  forall(a,b : kinship("mother",a,b)
    => kinship("son",b,a) or kinship("daughter",b,a))
// Other constraints are omitted...
```

**Listing 6.1:** The Scallop program for reasoning over kinship graphs in CLUTRR.

be directly used for semantic constraint loss, which we will introduce later in Section 6.1.2.

### 6.1.2 Learning Pipeline

The learning pipeline concerns tightly integrating a perceptive model for relation extraction with Scallop's symbolic engine 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 $\phi$ represents the weights of the learned logic rules. It takes as input the probabilistic relational symbols and the query $q$ and outputs

a distribution $\hat{y}$ over the set of all possible relations $\mathcal{R}$. Overall, the deductive model is written as

$$\hat{y} = \mathcal{P}_\phi(\mathcal{M}_\theta(x), q). \tag{6.1}$$

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

$$l_{\texttt{sl}} = \mathcal{P}_{\texttt{sl}}(\mathcal{M}_\theta(x), \phi) \tag{6.2}$$

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

$$J(\theta, \phi) = \frac{1}{|\mathcal{D}|} \sum_{(x,q,y)\in\mathcal{D}} w_1 \mathcal{L}(\mathcal{P}_\phi(\mathcal{M}_\theta(x), q), y) + w_2 \mathcal{P}_{\texttt{sl}}(\mathcal{M}_\theta(x), \phi), \tag{6.3}$$

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 6.1. We only need to additionally configure the Scallop module to output two relations, `answer` (for kinship prediction) and `violation` (for semantic loss).

### 6.1.3 Relation Extraction

Since pretrained LMs have strong pattern recognition capabilities for tasks like Named-Entity-Recognition (NER) and Relation Extraction (RE) (Tenney *et al.*, 2019; Soares *et al.*, 2019), 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 pretrained 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 $\theta$ contains the parameters of both the LM and the classifier.

We assume the relations to be classified come from a finite set of relations $\mathcal{R}$. For example in CLUTRR (Sinha *et al.*, 2019), we have 20 kinship relations including mother, son, uncle, father-in-law, etc. In practice, we perform $(|\mathcal{R}|+1)$-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 and $n$ entities in the context $x$, we extract $mn(n-1)(|\mathcal{R}|+1)$ probabilistic relational symbols. Each symbol is denoted as an atom of the form $p(s,o)$, where $p \in \mathcal{R} \cup \{\text{n/a}\}$ 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 $\mathbf{r} = \mathcal{M}_\theta(x) \in \mathbb{R}^{mn(n-1)(|\mathcal{R}|+1)}$.

**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 has an attached a weight, initialized by prompting an underlying LM. Thus, let a composition rule be $prob :: \text{comp}(r,p,q)$, which means one's $r$'s $p$ is their $q$, with probability weight $prob$. For example, the facts listed in Listing 6.2 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 6.2:** A few probabilistic composite rules that are learned.

Given that the relations $r,p,q \in \mathcal{R}$, Scallop automatically enumerates $r$ and $p$ from $\mathcal{R}$ while querying for LM to unmask the value of $q$.

LM then returns a distribution of words, which we take an intersection with $\mathcal{R}$. The probabilities combined form the initial rule weights $\phi$. 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 prompts. In fact, with the above prompt, even GPT-3 can only produce 62% of composition rules correctly. While we can edit the prompt to include few-shot examples, in this work we consider fine-tuning such rule weights $\phi$ 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 $|\mathcal{R}|^3 = 20^3$ candidate rules.

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

**Semantic loss and integrity constraints.** In general, learning with weak supervision labels is hard, not to mention that the deductive rules are learned 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 learned 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 extraction—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 formula is encoded in Scallop on lines 14–16 of Listing 6.1. Through differentiable reasoning, we evaluate the probability of such constraints being violated, yielding our expected *semantic loss*. In practice, an arbitrary number of constraints can be included, though too many interleaved ones could hinder learning.

### 6.1.4 Experiment

**Setup.** The CLUTRR dataset is divided into different difficulties measured by $k$, the number of facts used in the reasoning chain. For training, we only have 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 $k \in \{2, \ldots, 10\}$. We first initialize all possible kinship composition rules with GPT-3 provided probabilities, and extract the relationship from the given story.

**Baselines.** We compare Scallop with a spectrum of baselines from purely neural to logically structured. The baselines include pretrained large language models (BERT (Kenton and Toutanova, 2019) and RoBERTa (Liu *et al.*, 2019), non-LM counterparts (BiLSTM (Hochreiter and Schmidhuber, 1997; Cho *et al.*, 2014) and BERT-LSTM), structured models (GAT (Veličković *et al.*, 2018), RN (Santoro *et al.*, 2017), and MAC (Hudson and Manning, 2018)), and other neurosymbolic models (CTP (Minervini *et al.*, 2020b) and RuleBert (Saeed *et al.*, 2021)). The structured models include those models with relational inductive biases, while the neurosymbolic models use logic constraints.

**Performance.** We compare the performance of Scallop against multiple baselines in Figure 6.3, which shows that our neurosymbolic solution outperforms all compared neural baselines by a large margin. We also show the top 20 learned rules from the CLUTRR experiment in Figure 6.2. All top-20 learned rules match our expected real-life kinship relations.

## 6.2 Visual Question Answering on Scene Images

In this section, we describe the Scallop program for one of our benchmark applications, CLEVR (Johnson *et al.*, 2016). In Figure 6.4, we illustrate a concrete example from the CLEVR dataset. The program is given two inputs, namely the image (left) and the question (top-right), and it is expected to produce an answer (bottom-right). In general, the images

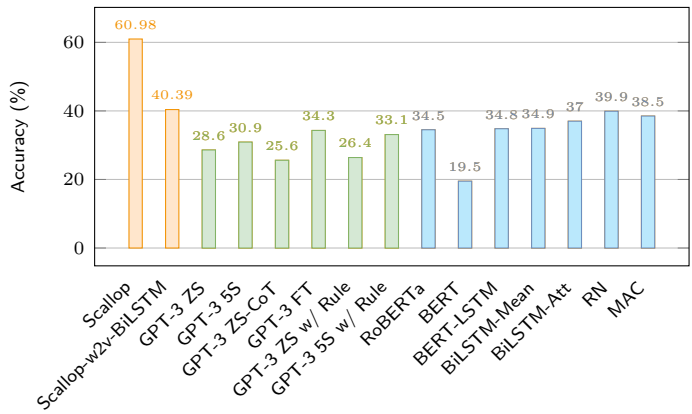
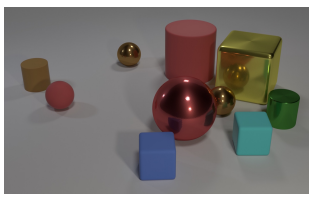



**Figure 6.3:** Scallop's performance on CLUTRR compared with various baselines.

**Table 6.2:** The learned logic rules (expressed as Horn rules) with top@20 confidence in CLUTRR rule learning.

| Confidence | Rule |
|---|---|
| 1.154 | mother(A,B) ← sister(A,C) ∧ mother(C,B) |
| 1.152 | daughter(A,B) ← daughter(A,C) ∧ sister(C,B) |
| 1.125 | sister(A,B) ← daughter(A,C) ∧ aunt(C,B) |
| 1.125 | father(A,B) ← brother(A,C) ∧ father(C,B) |
| 1.123 | granddaughter(A,B) ← grandson(A,C) ∧ sister(C,B) |
| 1.120 | brother(A,B) ← sister(A,C) ∧ brother(C,B) |
| 1.117 | brother(A,B) ← son(A,C) ∧ uncle(C,B) |
| 1.105 | brother(A,B) ← daughter(A,C) ∧ uncle(C,B) |
| 1.104 | daughter(A,B) ← wife(A,C) ∧ daughter(C,B) |
| 1.102 | mother(A,B) ← brother(A,C) ∧ mother(C,B) |
| 1.102 | brother(A,B) ← father(A,C) ∧ son(C,B) |
| 1.096 | sister(A,B) ← mother(A,C) ∧ daughter(C,B) |
| 1.071 | sister(A,B) ← father(A,C) ∧ daughter(C,B) |
| 1.071 | son(A,B) ← son(A,C) ∧ brother(C,B) |
| 1.070 | uncle(A,B) ← father(A,C) ∧ brother(C,B) |
| 1.066 | daughter(A,B) ← son(A,C) ∧ sister(C,B) |
| 1.061 | brother(A,B) ← brother(A,C) ∧ brother(C,B) |
| 1.056 | grandson(A,B) ← husband(A,C) ∧ grandson(C,B) |
| 1.055 | sister(A,B) ← son(A,C) ∧ aunt(C,B) |
| 1.053 | grandmother(A,B) ← sister(A,C) ∧ grandmother(C,B) |

in the CLEVR dataset may contain up to 10 primitive objects, each with a predefined set of shapes, colors, materials, and sizes. There is a range of questions whose answer may be numbers (counting questions), true/false (existence or comparing or assertive questions), or properties (querying color).



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

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

```
Answer: true
```

**Figure 6.4:** An example problem in CLEVR. Our model is supposed to answer the given question based on the image shown on the left.

We decompose our solution to this application into three sub-tasks: (a) extracting a structured 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 programmed 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 (OpenAI, 2023b) and CLIP (Radford *et al.*, 2021), in this section we present an off-the-shelf no-training solution. We next describe how we solve each of these sub-tasks.

### 6.2.1 Image to structured scene graph

To convert an image to a structured scene graph, we use two vision models: OWL-ViT (Minderer *et al.*, 2022) for obtaining object segments, and CLIP (Radford *et al.*, 2021) for classifying object properties. The goal is to construct a scene graph which contains the shape, color,

material, and size for each object, as well as the spatial relationships between each pair of objects.

Our object detection predicate is defined in Listing 6.3. We use the `@owl_vit` foreign attribute to decorate a predicate `segment_image`. Here, the image has one bounded argument which 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 6.5.

```
@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 6.3:** Definition of the relation used for image segmentation, using OWL-ViT.

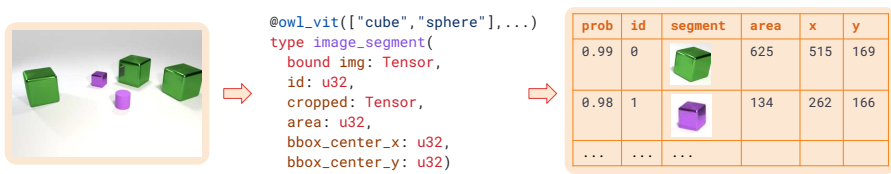



**Figure 6.5:** An illustration of the `segment_image` relation.

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 the foreign function `$load_image`, and then segment the image using the `segment_image` predicate defined previously. Our code is illustrated in Listing 6.4.

```
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) = image(img) and
10   segment_image(img, id, seg, area, x, y)
11 rel obj(id) = obj_seg(id, _, _, _, _)
```

**Listing 6.4:** The Scallop program that loads and segments a CLEVR image.

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 6.5). 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 6.5.

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

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

**Listing 6.5:** Classifier relations using CLIP.

The spatial relationship (`left`, `right`, `front`, and `behind`) is derived from object coordinates (Listing 6.6). 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.

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

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

**Listing 6.6:** Scallop rules for deriving spatial relations between pairs of objects.

### 6.2.2 NL question to programmatic query

We use the GPT-4 model (OpenAI, 2023b) 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 6.7. 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.

```
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 6.7:** The DSL for representing the NL questions in the CLEVR dataset, defined in Scallop.

We then create the semantic parser for the DSL by configuring a relation to parse the natural language question into a programmatic `Query`, shown in Listing 6.8. 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`. `header` constitutes the system prompt, while the structured `examples` are expanded with the `prompt` into the few-shot examples. In Figure 6.6 we show one specific example of "conversation" with LLM to precisely parse the NL question into `Query`.

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

**Listing 6.8:** A semantic parser relation `parse_query`.

### 6.2.3 Putting it all together

The last part which brings everything together is the semantics of our `Query` DSL, shown in Listing 6.7. We can start by treating each variant of our DSL as a *function*. Assume $O_{\text{all}} = \{o_1, o_2, \ldots, o_n\}$ represents 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. Denoting $\mathcal{O} = \mathcal{P}(O_{\text{all}})$, we assign function types and functional semantics to the variants in our DSL (Figure 6.7). 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`

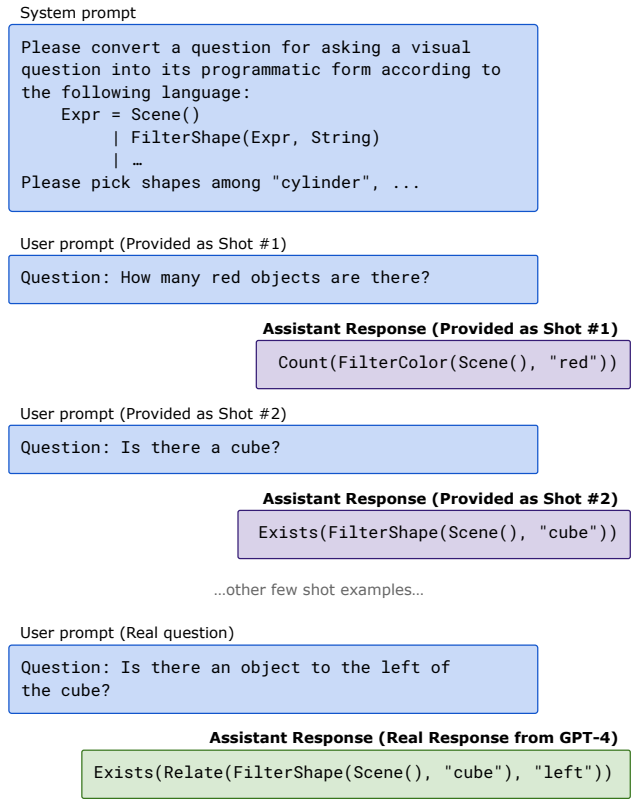



**Figure 6.6:** A "conversation" between user and the LLM for semantically parsing the NL question into programmatic query in our domain specific language (Listing 6.7). We use few-shot prompting in order to generate accurate programmatic query. Everything except the last bubble (green) is generated 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.

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 6.9. Specifically for `eval_obj`, even though the original functions return sets of objects, we may define

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

**Figure 6.7:** 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.

```
1  // for functions like Count that return numbers
2  type eval_num(q: Query, n: usize)
3
4  // for functions like Exists that return booleans
5  type eval_bool(q: Query, b: bool)
6
7  // for functions like Scene that return object sets
8  // note: we use `u32` to represent Object IDs
9  type eval_obj(q: Query, obj_id: u32)
10
11 // for functions like QueryColor that return
12 // stringified attributes, such as
13 // "red", "cube", "left", "large", etc.
14 type eval_str(q: Query, attr: String)
```

**Listing 6.9:** The type declarations of `eval_*` relations in Scallop.

a relation whose facts relate a function with one of the objects in its output set. Such a representation is natural (and unique) in relational programming—it allows us to tag each output with probabilities, which might be very hard to do in traditional functional semantics.

We can now start defining the semantics of our DSL in Scallop (Listing 6.10). 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 lines 3-4 handles the `FilterShape(e1, s)` query: it evaluates the subquery `e1` to obtain object `o`, and further quantifies it using the `shape(o,`

```
// 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 6.10:** The semantics of CLEVR DSL defined in Scallop.

`s)` atom to make sure that it has the desired shape. For the rules 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 (for `count`) or `false` (for `exists`).

The rules are thus defined relatively concisely. While the rules look like their functional counterparts, they actually have 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.

### 6.2.4 Evaluation

We evaluate our method against another no-fine-tuning VQA model, ViLT-VQA (Kim *et al.*, 2021) as a baseline. As shown in Table 6.3, our method significantly outperforms the baseline model on the CLEVR dataset under the no-fine-tuning setting. We note that the CLEVR dataset requires systematic compositional reasoning, meaning that pure data-driven neural models may never reach the generalizability of models

**Table 6.3:** Quantitative results on CLEVR dataset.

| Method | CLEVR | |
|---|---|---|
| | Recall@1 | Recall@3 |
| ViLT-VQA | 0.241 | 0.523 |
| Ours | **0.463** | **0.638** |

with symbolic components. Additionally, we have attempted to solve CLEVR under other settings, such as with the training of simple unary attribute and binary relation classifiers from scratch. There, Scallop is used as the differentiable module in an end-to-end training loop, with which we obtain an accuracy of 95.4% (Li *et al.*, 2023). In general, we conclude that a neurosymbolic approach is very suitable for solving VQA tasks that require both perception and reasoning.

## 6.3 Aligning Texts and Videos for 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 6.8 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) (Shang *et al.*, 2017; Zhu *et al.*, 2022) has been proposed to represent entity relations throughout a video. Existing approaches for learning STSGs from video data are typically fully-supervised (Nag *et al.*, 2023; Cong *et al.*, 2021). They can potentially learn high-fidelity STSGs but are greatly hindered in practice due to the complexity of low-level annotations that are laborious to obtain (Yang *et al.*, 2023).

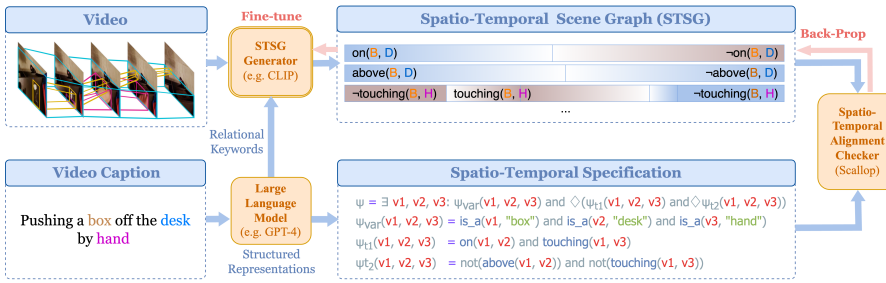



**Figure 6.8:** An example from 20BN demonstrating the end-to-end learning pipeline. The model $M_\theta$ processes a video to generate a probabilistic STSG. With 3-shot GPT-4, an STSL specification is derived from the video caption, which describes a temporal sequence of two events: "the box is on the 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, resulting in a differentiable alignment score of 0.95.

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*

($\text{LTL}_f$) (De Giacomo and Vardi, 2013) 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 (Chang *et al.*, 2019) 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 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 (OpenAI *et al.*, 2024) 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. We illustrate the full learning pipeline in Figure 6.8, which the remainder of this section will describe in greater detail.

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 $M_\theta$ 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:

$$J(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(X,C)\in\mathcal{D}} \mathcal{L}(\Pr(M_\theta(X) \models \texttt{LLM}(C)), 1), \tag{6.4}$$

where $\texttt{LLM}(C)$ is an STSL formula $\psi$ 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.

### 6.3.1 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 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, $M_\theta$, parameterized by $\theta$, takes in pixel-based raw video data $X$, and generates a distribution of STSGs. This distribution is then encoded as a predicted probabilistic relational database, $\hat{\mathbf{r}} = M_\theta(X)$.

### 6.3.2 Spatio-Temporal Specification Language

Linear Temporal Logic (LTL) (Pnueli, 1977) 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 (Chaki *et al.*, 2005; Kesten *et al.*, 1998) and control (Ding *et al.*, 2014; Sadigh *et al.*, 2014). As we operate on prerecorded, finite-length videos, our language is developed using $\text{LTL}_f$ (De Giacomo and Vardi, 2013), which supports LTL reasoning over finite traces. Thus, we use $\text{LTL}_f$ as a framework for specifying events and their temporal relationships.

Our STSL (Figure 6.9) further extends $\text{LTL}_f$ by introducing relational predicates and variables. It starts from the specification $\psi$ which

existentially quantifies variables in an STSL formula. The formula $\varphi$ is inductively defined, with basic elements as relational atoms $\alpha$ 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 $\Gamma$, denoted by $\text{subst}_\Gamma(\bar{t})$. From here, $\varphi$ can be constructed using basic propositional logic components $\wedge$ (and) and $\neg$ (not). The system additionally includes temporal unary operators $\bigcirc$ (next), $\Box$ (always), and $\Diamond$ (finally), and a binary operator $\mathbf{U}$ (until) (Albers *et al.*, 2009). For example, the description "A hand continues to touch the box until it drops." can be represented as an STSL formula

$$\psi = \texttt{touch(h, b)}\ \mathbf{U}\ \texttt{drop(b, \_)}. \tag{6.5}$$

Note that an argument to the predicate `drop` is a wildcard (`_`), since we do not specify where the box drops from. This formula might seem too strict since it requires the two events to be consecutive. To make the specification more general, we alter Eqn. 6.5 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.

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

**Figure 6.9:** The formal syntax of STSL, where $a$ represents relational predicates, $c$ represents constants, and $v$ represents variables. Here, $\wedge$ and $\neg$ represent logical "and" and "not" respectively. Formula may also contain temporal operators $\bigcirc$ (next), $\Box$ (global), $\Diamond$ (finally), and $\mathbf{U}$ (until).

In Listing 6.11 we define the STSL using algebraic data types (ADTs). We stratify an STSL formula into a temporal formula and a logical formula so that the semantics can be implemented more succinctly later. In the temporal section (lines 6-10), we cover all the core temporal operations supported in STSL. In the logical section (lines 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,

```
1  // Term: constant string or variable or wildcard
2  type Term = Const(String) | Var(Var) | Wildcard()
3
4  // TForm: temporal formula, containing
5  // `Global`, `Finally`, `Until`, and `Next`
6  type TForm = Global(TForm)
7             | Finally(TForm)
8             | Until(TForm, TForm)
9             | Next(TForm)
10            | Logic(LForm)
11
12 // LForm: logical formula, containing logical
13 // conjunction and atomic queries
14 type LForm = And(LForm, LForm)
15            | Unary(String, Term)
16            | Binary(String, Term, Term)
17            | NegUnary(String, Term)
18            | NegBinary(String, Term, Term)
19
20 // an example specification: touch(h,b) U drop(b,_)
21 const MY_SPEC = Until(
22     Binary("touch", Var("h"), Var("b")),
23     Binary("drop", Var("b"), Wildcard()))
```

**Listing 6.11:** STSL defined in Scallop.

then our semantics might suffer from unstratified negation, prohibiting our program to be compiled by the Scallop compiler. On lines 21–23, we show one example specification representing Equation 6.5.

### 6.3.3 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 $\psi$ 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 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. In Section 6.3.5, we explain how these structured representations are incorporated into the loss function.

To extract such structured representations from the caption, we designed a generic prompt 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; and (c) few-shot examples of caption and JSON structured representations pairs. We illustrate a caption and its structured representation in Figure 6.10.

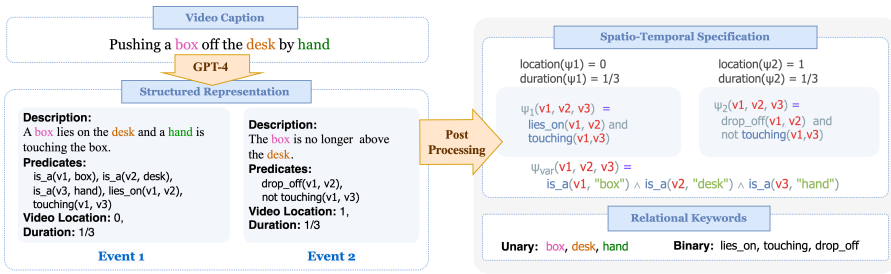



**Figure 6.10:** An illustration of our pipeline for natural language caption to programmatic spatio-temporal specification.

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

$$\psi = \Diamond_{e_i \in \bar{e}} \psi_i, \quad \psi_i = \bigwedge_{\phi_j \in \psi_i} \phi_j \tag{6.6}$$

### 6.3.4 Spatio-Temporal Alignment Checking

Given a probabilistic database $\mathbf{r}$ that encodes a distribution of STSGs (Section 6.3.1), and a specification $\psi$ 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 $\psi$ or not. From here, the alignment score can be computed as the sum of the probabilities of worlds satisfying $\psi$:

$$\Pr(\mathbf{r} \models \psi) = \sum_{\substack{w \in \mathcal{P}(\mathbf{r}) \\ w \models \psi}} \Pr(w), \tag{6.7}$$

$$\Pr(w) = \prod_{f \in w} \Pr(f) \prod_{f' \in \mathbf{r} \setminus w} (1 - \Pr(f')) \tag{6.8}$$

Enumerating all possible worlds is intractable due to its exponential complexity. Since Scallop employs 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 6.12. 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 $\psi$ is aligned if the full $\mathbf{r}$ (from 1 to $m$) satisfies $\psi$ with a concrete variable grounding $\Gamma$, which maps variables to concrete entities. We illustrate one simplified evaluation process in Figure 6.11. 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 storing our STSG (lines 7–9 of Listing 6.13). Since in STSG we only deal with unary and binary relations, we hard-code the two relations `unary_atom` and `binary_atom`. One important difference between our alignment checker and the semantics of other

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

```
1  // grounding a term into an object
2  type ground_term(t: Term, o: Obj)
3  rel ground_term(Var(v), o) = var_obj(v, o)
4  rel ground_term(Const(c), o) = name(o, c)
5  rel ground_term(Wildcard(), o) = object(o)
6
7  // aligning logical formula
8  type align_lform(phi: LForm, s: Time)
9  rel align_lform(Binary(pred, t1, t2), s) =
10   ground_term(t1, o1) and ground_term(t2, o2)
11   and binary_atom(pred, s, o1, o2) and time(s)
12 rel align_lform(NegBinary(pred, t1, t2), s) =
13   ground_term(t1, o1) and ground_term(t2, o2)
14   and not binary_atom(pred, s, o1, o2) and time(s)
15 // handling other logical formulas...
16
17 // aligning temporal formula: the formula `psi` is
18 // aligned with the scene graph starting from time
19 // `s`, given the variable assignment context
20 type align_tform(psi: TForm, s: Time)
21 rel align_tform(Global(p1), s) =
22   max_time(n) and align_all_tform(p1, s, n)
23 rel align_tform(Finally(p1), s) =
24   align_once_tform(p1, s)
25 rel align_tform(Until(p1, p2), s) =
26   time(t + 1) and s < (t + 1)
27               and align_all_tform(p1, s, t)
28               and align_tform(p2, t + 1)
29 // handling other temporal formulas...
```

**Listing 6.12:** The (partial) alignment checker for STSL, implemented in Scallop.

DSLs shown before is that our 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 15 of Listing 6.13.

We now present the Scallop code for the STSL alignment checker (Listing 6.12). Starting from lines 1–5, we define the relation used to ground each term into concrete objects. Specifically, when the term is

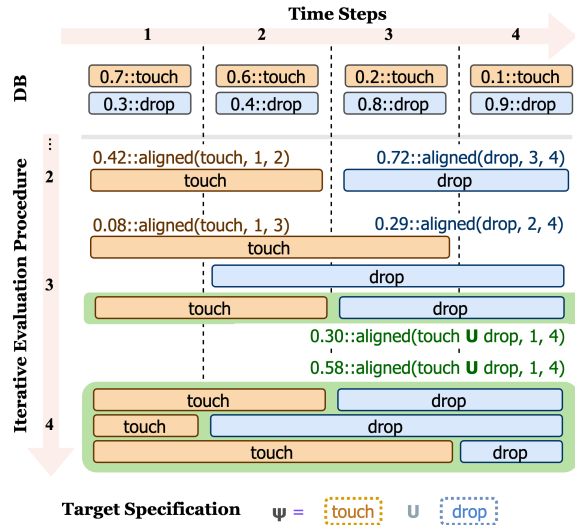



**Figure 6.11:** The evaluation process aligning a spatio-temporal scene graph (DB) with a specification `climb` **U** `walk`. This figure elides showing the arguments of the relational predicates and focuses only on matching sequential events.

```
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,
10                   o1: Obj, o2: Obj)
11
12 // Variable assignments
13 type name(o: Obj, name: Var)
14 type variable(var: Var), object(o: Obj)
15 rel var_obj = disjunct[v](o: variable(v), object(o))
```

**Listing 6.13:** Setting up the Spatio-Temporal Scene Graph (STSG) as well as the variable assignment solving context.

a variable (`Var`), we use the `var_obj` relation defined in Listing 6.13 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 formulas, which require the grounding of all terms appearing in the atoms. As for aligning temporal formulas, 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 6.12).

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

**Figure 6.12:** Formal semantics of STSL. $\langle w, s\rangle \models \psi$ means the STSL specification $\psi$ is *aligned* with the ST-SG $w$ starting from time $s$. We use $w \models \psi$ as an abbreviation for $\langle w, 1\rangle \models \psi$.

### 6.3.5 Loss Function

**Contrastive Learning.** Unavoidable dataset biases exist in the specification. Contrastive learning can effectively reduce the bias and generate explanations of better quality. Let $(X_i, \psi_i)$ and $(X_j, \psi_j)$ be two datapoints in a mini-batch $B$, where $\psi_i$ and $\psi_j$ are the specifications for video $X_i$ and $X_j$ correspondingly, already extracted by LLMs. If $X_i \models \psi_j$, then it is an extra positive sample to the video $X_i$, else it is a negative sample to $X_i$. Let the relational database representing the STSG predicted by the model be $\mathbf{r} = M_\theta(X_i)$. We can thus define our contrastive loss $\mathcal{L}_c$:

$$l(B, w, \psi) = \frac{1}{|B|} \sum_{(X_j, \psi_j) \in B} \mathcal{L}(Pr(w \models \psi), \mathbb{1}[\psi = \psi_j]) \tag{6.9}$$

$$\mathcal{L}_c = \frac{1}{|B|} \sum_{(X,\psi) \in B} \sum_{w \in \mathcal{P}(M_\theta(X))} l(B, w, \psi) \tag{6.10}$$

**Time-Span Supervision.** A video caption is expanded into a sequence of events using an LLM, with each event assigned a specific temporal target, detailing its location and duration within the video, as illustrated in Section 6.3.3. By aligning the spatio-temporal specification $\psi$ with the video, we can identify when its sub-specifications, $\psi_1, \psi_2, \ldots, \psi_n$, are met. This alignment facilitates weak supervision across the entire time span. Formally, we define $\sigma(s, l, d) \in [0, 1]$, the time span alignment score, as a function on actual time stamp $s$, expected time stamp $l$, and expected event duration $d$. In particular, $\sigma(s, l, d)$ should peak at 1 when the event happens exactly at the expected locations ($s = l$). In practice, we embed $\sigma$ into the computation of probabilistic alignment between STSG $w$ and an atomic specification $a(\bar{t})$, where we utilize the expected location ($\text{loc}(a)$) and the duration ($\text{dur}(a)$) extracted by the LLM:

$$\Pr(\langle \Gamma, w, s \rangle \models a(\bar{t})) = \rho(\Gamma, w, s, a(\bar{t})) \cdot \sigma(s, \text{loc}(a), \text{dur}(a)) \tag{6.11}$$

$$\rho(\Gamma, w, s, a(\bar{t})) = \Pr(a(\bar{c}) \mid a(\bar{c}) \in w[s] \ \wedge \ \bar{c} = \text{subst}_\Gamma(\bar{t})) \tag{6.12}$$

$$\sigma(s, l, d) = \max(0, 1 - \frac{2|s - l|}{d}) \tag{6.13}$$

**Semantic Loss.** To provide further supervision, we resort to human knowledge encoded in the form of integrity constraints. We introduce semantic loss reflecting the probability of violating the integrity constraints. For example, an entity in a video cannot be `open` and `closed` at the same time; an entity that is not `bendable` cannot be `deformed`. These integrity constraints may interweave so heavily that it is hard to use a simple disjoint multi-class classifier to enforce. We encode all integrity constraints in the form of first-order logic rules, and our reasoning engine generates the probability that these constraints are violated. We thus have the semantic loss as an extra-weighted term

after calculating the other loss components. Let $n$ be the number of integrity constraints and $IC_i$ be the $i$-th integrity constraint; we have:

$$\mathcal{L}_s = \sum_{i=1}^{n} \sum_{w \in \mathcal{P}(M_\theta(X))} \mathcal{L}(\Pr(w \not\models \psi_{IC_i}), 0). \tag{6.14}$$

### 6.3.6 Evaluating on OpenPVSG

**Dataset.** The OpenPVSG (Yang *et al.*, 2023) dataset comprises 400 videos sourced from Ego4D (Grauman *et al.*, 2021), VidOr (Shang *et al.*, 2019; Thomee *et al.*, 2016), and EpicKitchen (Damen *et al.*, 2022; Damen *et al.*, 2018). This dataset offers fine-grained ground truth annotations of STSGs for $150K$ frames, encompassing 126 object classes and 57 relation classes. We train on $1,832$ video-caption pairs, and evaluate on 438 video-STSG pairs.

**Experimental Setup.** As illustrated in Figure 6.13, our objective is to train an STSG generator capable of taking a video clip with object bounding boxes as input and predicting the properties, attributes, and relationships between objects. We leverage our learning pipeline to fine-tune three vision-language models—VIOLET (Fu *et al.*, 2021), SigLIP (Zhai *et al.*, 2023), and CLIP (Radford *et al.*, 2021)—using weak supervision from captions. These models predict both similarity scores between cropped objects and unary predicate keywords, as well as between object pairs and binary predicate keywords, resulting in a probabilistic STSG. All backbone models support open-world vocabularies and are thus robust to the fuzziness present in GPT-4-generated structured representations.

**Evaluation Metric.** We evaluate model performance using Recall@k (R@k) which estimates whether the ground truth label is within the top-$k$ prediction of a given model. During evaluation, the model processes (a) the full vocabulary of object and relation classes and (b) preprocessed cropped objects and object pairs, predicting the corresponding probabilistic STSG. In particular, unary R@k assesses object category prediction capability, while binary R@k evaluates pair-wise prediction of binary relations. Unlike the original VIOU metric from the OpenPVSG, which combines object segmentation, object classification, and relational classification into a single score, R@k provides a

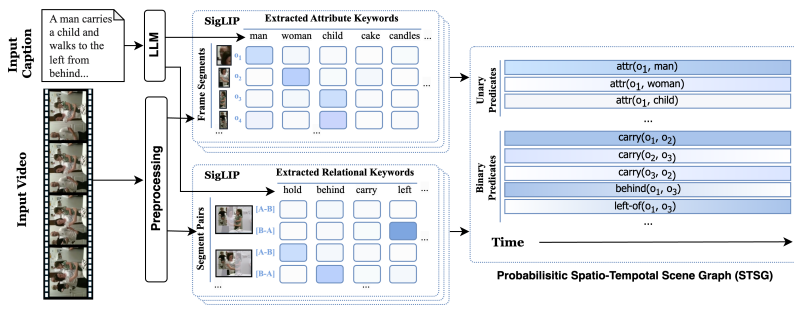



**Figure 6.13:** Pipeline illustration with CLIP as the backbone model for probabilistic STSG generation.

**Table 6.4:** We show the performance improvements of base backbone models and their fine-tuned version, on the R@k metrics of unary and binary predicate prediction. As shown by the increments, Scallop's weak supervisory learning framework significantly enhances all three models' performance on the STSG extraction tasks.

| Backbone | | Unary | | | Binary | | |
|---|---|---|---|---|---|---|---|
| | | R@1 | R@5 | R@10 | R@1 | R@5 | R@10 |
| VIOLET | Base | 0.0660 | 0.1855 | 0.2983 | 0.0460 | 0.1307 | 0.2636 |
| | FT | 0.0878 | 0.2574 | 0.3463 | 0.0501 | 0.2028 | 0.3451 |
| | Incr. | ↑ 0.0218 | ↑ 0.0719 | ↑ 0.0480 | ↑ 0.0041 | ↑ 0.0721 | ↑ 0.0815 |
| SigLIP | Base | 0.0000 | 0.0179 | 0.0483 | 0.0000 | 0.0362 | 0.1667 |
| | FT | 0.1467 | 0.2627 | 0.3152 | 0.0347 | 0.1624 | 0.3012 |
| | Incr. | ↑ 0.1467 | ↑ 0.2448 | ↑ 0.2669 | ↑ 0.0347 | ↑ 0.1262 | ↑ 0.1345 |
| CLIP | Base | 0.1633 | 0.3381 | 0.4404 | 0.0197 | 0.0673 | 0.0988 |
| | FT | 0.2778 | 0.5231 | 0.6402 | 0.1482 | 0.4214 | 0.5398 |
| | Incr. | ↑ 0.1145 | ↑ 0.1850 | ↑ 0.1998 | ↑ 0.1284 | ↑ 0.3540 | ↑ 0.4410 |

more detailed assessment of a model's ability to recognize objects and relations separately.

**Backbone models significantly improve after algorithmic weak supervision.** We validate Scallop' effectiveness in learning STSGs with weak supervision by comparing the performance improvements of the backbone models after fine-tuning. As shown in Table 6.4, our method significantly enhances backbone performance using only captions for weak supervision on the OpenPVSG dataset.

**Data efficiency.** To further assess our method's data efficiency, we train the model on 10% and 50% of the training dataset. As illustrated in

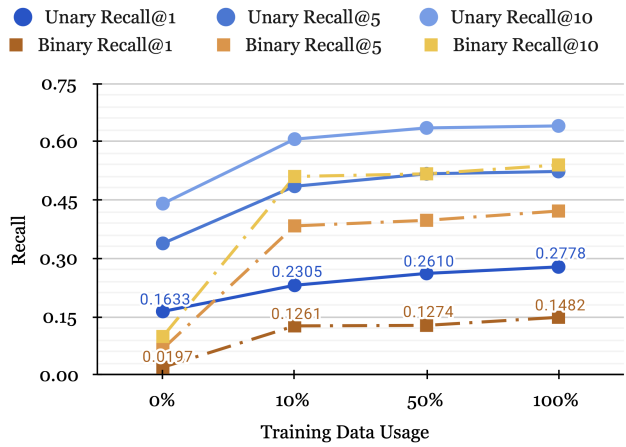



**Figure 6.14:** Data-efficient fine-tuning on OpenPVSG dataset: Providing only 10%, 50%, and 100% of the training dataset significantly enhances the performance of CLIP model.

**Table 6.5:** Comparison between *weakly supervised* Scallop-CLIP and *fully supervised* IPS and VPS methods on various backbones trained on the full OpenPVSG. Scallop-CLIP significantly outperforms all baselines, except on Binary R@1, despite using *weak supervision*.

| Method | Unary | Binary | | |
|---|---|---|---|---|
| | Acc. (%) | R@1 | R@5 | R@10 |
| IPS-Vanilla | 15.13 | 0.0741 | 0.1081 | 0.1109 |
| IPS-Filter | 13.14 | 0.0777 | 0.1040 | 0.1133 |
| IPS-Conv | 15.13 | 0.0861 | 0.1143 | 0.1218 |
| IPS-Trans | 14.67 | 0.1419 | 0.2032 | 0.2207 |
| VPS-Vanilla | 5.49 | 0.0374 | 0.0517 | 0.0531 |
| VPS-Filter | 5.46 | 0.0405 | 0.0480 | 0.0488 |
| VPS-Conv | 7.46 | **0.1616** | 0.1781 | 0.2343 |
| VPS-Trans | 5.46 | 0.1019 | 0.1499 | 0.1562 |
| Scallop-CLIP | **27.78** | 0.1482 | **0.4214** | **0.5398** |

Figure 6.14, even with just 10% of the training data (183 video-caption pairs), our method significantly enhances the unary R@1 from 0.1633 to 0.2305 and the binary R@1 from 0.0197 to 0.1261. On average, using just 10% of the data achieves 70.75% of the performance obtained with the full dataset, highlighting our pipeline's data efficiency.

**Weak-supervision may outperform full-supervision.** To better understand the efficacy of weak supervision, we also compare them against fully supervised methods. We study 8 fully supervised baselines, which employ two different video panoptic segmentation strategies: Image Panoptic Segmentation with Tracker (IPS) and Video Panoptic Segmentation (VPS). For relation extraction, the baselines employ 4 model architectures: (1) Vanilla: fully-connected layers, (2) Filter: handcrafted filters, (3) Conv: 1D-convolutional layers, and (4) Trans: transformer encoders. As shown in Table 6.5, Scallop-CLIP significantly outperforms the best fully supervised methods in all metrics except for binary R@1, where it ranks just after the top-performing VPS-Conv.

# 7

# Conclusion

Logical reasoning and deep learning embody two prevalent paradigms of modern programming. The two paradigms are complementary in nature. For instance, the task of code completion requires deep learning to comprehend programmer intent from the code context, and logical reasoning to ensure that the generated code is correct. A central question in AI then is how to program such tasks by integrating the two paradigms. Neurosymbolic programming (Chaudhuri *et al.*, 2021) is an emerging approach to integrate symbolic knowledge and reasoning with neural models for better efficiency, interpretability, and generalizability than the neural or symbolic counterparts alone.

In this article, we introduce Scallop, a general-purpose programming language and framework designed for developing neurosymbolic applications. Scallop's main contributions are threefold: a rich and intuitive core language, a scalable and configurable provenance framework, and seamless integration with neural networks, including foundation models. Scallop's relational and declarative programming paradigm simplifies the specification of logical reasoning components. Its differentiable reasoning module, built on a specialized provenance framework, supports efficient neurosymbolic learning through algorithmic supervision. Fur-

thermore, Scallop's flexible interface provides access to state-of-the-art foundation models, enhancing its capability to tackle complex tasks that require both perceptual understanding and reasoning.

## 7.1 Limitations

While Scallop excels in scenarios (a), (b), and (c) as outlined in Figure 1.2, where we can assume the symbolic component is provided, there are still challenges in learning, synthesizing, or refining a Scallop program within the learning loop. Additionally, Scallop is well-suited for defining discrete and logical rules but is less effective for specifying numerical properties, limiting its application for low-level control tasks in robotics. Furthermore, the Datalog-based syntax of Scallop poses difficulties in specifying programs that require planning or causal reasoning. The current provenance framework is optimized for probabilistic and differentiable reasoning components; however, how it could propagate learning signals for large language models remains an open question.

## 7.2 Future Work

To address these limitations, we plan to advance Scallop along three primary directions:

1. *Efficiency*. We aim to enhance Scallop's performance by developing more optimized provenance frameworks and runtime environments. Specifically, we intend to improve compiler optimizations and introduce advanced runtime storage systems for greater efficiency. Additionally, implementing a GPU-based runtime for Scallop will allow the inference engine to run on GPUs, similar to neural components, enhancing scalability and speed.

2. *Expressiveness*. Expanding Scallop's language capabilities to support more complex probabilistic specifications, as well as planning, causal, and counterfactual programs, will greatly increase its expressiveness. This will require extensions to both Scallop's surface syntax and underlying semantics, broadening the range of tasks Scallop can effectively handle.

3. *Applicability*. To expand Scallop's applicability, we plan to incorporate it into diverse machine learning scenarios and benchmark its performance in these contexts. Through focused research in machine learning and neurosymbolic learning, we aim to adapt Scallop to a broader set of tasks. Finally, we intend to explore applications in safety-critical domains such as robotics, healthcare, and scientific research, which will not only demonstrate Scallop's versatility but also advance the field of neurosymbolic AI.

# References


Abiteboul, S., R. Hull, and V. Vianu. (1995). *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc.

Albers, S., A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas. (2009). *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*. Vol. 5555. Springer Science & Business Media.

Alur, R., R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. (2013). "Syntax-guided synthesis". *2013 Formal Methods in Computer-Aided Design*: 1–8. URL: https://api.semanticscholar.org/CorpusID:6705760.

Beurer-Kellner, L., M. Fischer, and M. Vechev. (2022). "Prompting Is Programming: A Query Language For Large Language Models". In: *PLDI*.

Bommasani, R., D. A. Hudson, E. Adeli, R. B. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, and et al. (2021). "On the Opportunities and Risks of Foundation Models". arXiv: 2108.07258.

Bubeck, S., V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, *et al.* (2023). "Sparks of artificial general intelligence: Early experiments with GPT-4". arXiv: 2303.12712.

Chaki, S., E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. (2005). "Concurrent software verification with states, events, and deadlocks". *Formal Aspects of Computing*. 17(4): 461–483.

Chang, C.-Y., D.-A. Huang, Y. Sui, L. Fei-Fei, and J. C. Niebles. (2019). "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*. 3546–3555.

Chaudhuri, S., K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, Y. Yue, *et al.* (2021). "Neurosymbolic Programming". *Foundations and Trends in Programming Languages*. 7(3). DOI: 10.1561/2500000049.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.* (2021). "Evaluating Large Language Models Trained on Code". arXiv: 2107.03374.

Chen, X., C. Liang, A. W. Yu, D. Zhou, D. Song, and Q. V. Le. (2020). "Neural Symbolic Reader: Scalable Integration of Distributed and Symbolic Representations for Reading Comprehension". In: *International Conference on Learning Representations (ICLR)*.

Cho, K., B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *EMNLP*.

Cong, Y., W. Liao, H. Ackermann, M. Y. Yang, and B. Rosenhahn. (2021). "Spatial-Temporal Transformer for Dynamic Scene Graph Generation". *CoRR*. abs/2107.12309. URL: https://arxiv.org/abs/2107.12309.

Csomay-Shanklin, N., W. D. Compton, I. D. J. Rodriguez, E. R. Ambrose, Y. Yue, and A. D. Ames. (2024). "Robust Agility via Learned Zero Dynamics Policies". URL: https://arxiv.org/abs/2409.06125.

Damen, D., H. Doughty, G. M. Farinella, S. Fidler, A. Furnari, E. Kazakos, D. Moltisanti, J. Munro, T. Perrett, W. Price, and M. Wray. (2018). "Scaling Egocentric Vision: The EPIC-KITCHENS Dataset". In: *European Conference on Computer Vision (ECCV)*.

Damen, D., H. Doughty, G. M. Farinella, A. Furnari, J. Ma, E. Kazakos, D. Moltisanti, J. Munro, T. Perrett, W. Price, and M. Wray. (2022). "Rescaling Egocentric Vision: Collection, Pipeline and Challenges for EPIC-KITCHENS-100". *International Journal of Computer Vision (IJCV)*. 130: 33–55. URL: https://doi.org/10.1007/s11263-021-01531-2.

Dannert, K. M., E. Grädel, M. Naaf, and V. Tannen. (2021). "Semiring Provenance for Fixed-Point Logic". In: *Conference on Computer Science Logic (CSL)*. DOI: 10.4230/LIPIcs.CSL.2021.17.

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

De Giacomo, G. and M. Y. Vardi. (2013). "Linear temporal logic and linear dynamic logic on finite traces". In: *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. Association for Computing Machinery. 854–860.

Dinella, E., H. Dai, Z. Li, M. Naik, L. Song, and K. Wang. (2020). "HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS".

Ding, X., S. L. Smith, C. Belta, and D. Rus. (2014). "Optimal control of Markov decision processes with linear temporal logic constraints". *IEEE Transactions on Automatic Control*. 59(5): 1244–1257.

Ellis, K., A. Albright, A. Solar-Lezama, J. B. Tenenbaum, and T. J. O'Donnell. (2022). "Synthesizing theories of human language with Bayesian program induction". *Nature Communications*. 13. URL: https://api.semanticscholar.org/CorpusID:251951680.

Ellis, K., C. Wong, M. I. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. B. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. (2020). "DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning". *CoRR*. abs/2006.08381. URL: https://arxiv.org/abs/2006.08381.

Fu, T.-J., L. Li, Z. Gan, K. Lin, W. Y. Wang, L. Wang, and Z. Liu. (2021). "Violet: End-to-end video-language transformers with masked visual-token modeling". *arXiv preprint arXiv:2111.12681*.

Gao, L., A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. (2023). "PAL: Program-aided Language Models". arXiv: 2211.10435 [cs.CL].

Grauman, K., A. Westbury, E. Byrne, Z. Chavis, A. Furnari, R. Girdhar, J. Hamburger, H. Jiang, M. Liu, X. Liu, M. Martin, T. Nagarajan, I. Radosavovic, S. K. Ramakrishnan, F. Ryan, J. Sharma, M. Wray, M. Xu, E. Z. Xu, C. Zhao, S. Bansal, D. Batra, V. Cartillier, S. Crane, T. Do, M. Doulaty, A. Erapalli, C. Feichtenhofer, A. Fragomeni, Q. Fu, C. Fuegen, A. Gebreselasie, C. González, J. Hillis, X. Huang, Y. Huang, W. Jia, W. Khoo, J. Kolár, S. Kottur, A. Kumar, F. Landini, C. Li, Y. Li, Z. Li, K. Mangalam, R. Modhugu, J. Munro, T. Murrell, T. Nishiyasu, W. Price, P. R. Puentes, M. Ramazanova, L. Sari, K. Somasundaram, A. Southerland, Y. Sugano, R. Tao, M. Vo, Y. Wang, X. Wu, T. Yagi, Y. Zhu, P. Arbeláez, D. Crandall, D. Damen, G. M. Farinella, B. Ghanem, V. K. Ithapu, C. V. Jawahar, H. Joo, K. Kitani, H. Li, R. A. Newcombe, A. Oliva, H. S. Park, J. M. Rehg, Y. Sato, J. Shi, M. Z. Shou, A. Torralba, L. Torresani, M. Yan, and J. Malik. (2021). "Ego4D: Around the World in 3, 000 Hours of Egocentric Video". *CoRR*. abs/2110.07058. URL: https://arxiv.org/abs/2110.07058.

Green, T. J., G. Karvounarakis, and V. Tannen. (2007). "Provenance Semirings". In: *ACM Symposium on Principles of Database Systems (PODS)*. DOI: 10.1145/1265530.1265535.

Gupta, T. and A. Kembhavi. (2022). "Visual Programming: Compositional visual reasoning without training". arXiv: 2211.11559.

Hochreiter, S. and J. Schmidhuber. (1997). "Long short-term memory". *Neural computation*.

Huang, J., Z. Li, B. Chen, K. Samel, M. Naik, L. Song, and X. Si. (2021). "Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning". In: *Conference on Neural Information Processing Systems (NeurIPS)*.

Hudson, D. A. and C. D. Manning. (2018). "Compositional Attention Networks for Machine Reasoning". In: *ICLR*.

Johnson, J., B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. B. Girshick. (2016). "CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning". URL: http://arxiv.org/abs/1612.06890.

Kenton, J. D. M.-W. C. and L. K. Toutanova. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL*.

Kesten, Y., A. Pnueli, and L.-o. Raviv. (1998). "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*. Springer. 1–16.

Kim, W., B. Son, and I. Kim. (2021). "ViLT: Vision-and-Language Transformer Without Convolution or Region Supervision". arXiv: 2102.03334 [stat.ML].

Kirillov, A., E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, *et al.* (2023). "Segment Anything". arXiv: 2304.02643.

Kojima, T., S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. (2022). "Large language models are zero-shot reasoners". In: *NeurIPS*.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. (1998). "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE*. 86(11). DOI: 10.1109/5.726791.

Lewkowycz, A., A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, *et al.* (2022). "Solving Quantitative Reasoning Problems with Language Models".

Li, J., Y. Wang, C. Wang, Y. Tai, J. Qian, J. Yang, C. Wang, J. Li, and F. Huang. (2018). "DSFD: Dual Shot Face Detector". URL: http://arxiv.org/abs/1810.10220.

Li, Q., S. Huang, Y. Hong, Y. Chen, Y. N. Wu, and S.-C. Zhu. (2020). "Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning". In: *ICML*. DOI: 10.48550/arXiv.2006.06649.

Li, X.-Y., W.-J. Lei, and Y.-B. Yang. (2022). "From Easy to Hard: Two-stage Selector and Reader for Multi-hop Question Answering". arXiv: 2205.11729 [cs.CL].

Li, Z., J. Huang, and M. Naik. (2023). "Scallop: A Language for Neurosymbolic Programming". In: *PLDI*. DOI: 10.1145/3591280.

Li, Z., A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song. (2021). "ARBITRAR: User-Guided API Misuse Detection". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 1400–1415. DOI: 10.1109/SP40001.2021.00090.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. (2019). "RoBERTa: A Robustly Optimized BERT Pretraining Approach". arXiv: 1907.11692.

Ma, Y. J., W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. (2024). "Eureka: Human-Level Reward Design via Coding Large Language Models". URL: https://arxiv.org/abs/2310.12931.

Manhaeve, R., S. Dumancic, A. Kimmig, T. Demeester, and L. D. Raedt. (2021). "Neural Probabilistic Logic Programming in DeepProbLog". *Artificial Intelligence*. 298. DOI: 10.1016/j.artint.2021.103504.

Mao, J., C. Gan, P. Kohli, J. B. Tenenbaum, and J. Wu. (2019). "The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision". arXiv: 1904.12584.

McKenna, N., T. Li, L. Cheng, M. J. Hosseini, M. Johnson, and M. Steedman. (2023). "Sources of Hallucination by Large Language Models on Inference Tasks". arXiv: 2305.14552.

Minderer, M., A. Gritsenko, A. Stone, M. Neumann, D. Weissenborn, A. Dosovitskiy, A. Mahendran, A. Arnab, M. Dehghani, Z. Shen, X. Wang, X. Zhai, T. Kipf, and N. Houlsby. (2022). "Simple Open-Vocabulary Object Detection with Vision Transformers". arXiv: 2205.06230 [cs.CV].

Minervini, P., S. Riedel, P. Stenetorp, E. Grefenstette, and T. Rocktäschel. (2020a). "Learning Reasoning Strategies in End-to-End Differentiable Proving". In: *ICML*. arXiv: 2007.06477.

Minervini, P., S. Riedel, P. Stenetorp, E. Grefenstette, and T. Rocktäschel. (2020b). "Learning reasoning strategies in end-to-end differentiable proving". In: *ICML*.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.* (2015). "Human-level Control Through Deep Reinforcement Learning". *Nature*. 518(7540). DOI: 10.1038/nature14236.

Nag, S., K. Min, S. Tripathi, and A. K. R. Chowdhury. (2023). "Unbiased Scene Graph Generation in Videos". arXiv: 2304.00733 [cs.CV].

Nakano, R., J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman. (2021). "WebGPT: Browser-assisted question-answering with human feedback". URL: https://arxiv.org/abs/2112.09332.

Nogueira, R. and K. Cho. (2019). "Passage Re-ranking with BERT". arXiv: 1901.04085.

O'Connell, M., G. Shi, X. Shi, K. Azizzadenesheli, A. Anandkumar, Y. Yue, and S.-J. Chung. (2022). "Neural-Fly enables rapid learning for agile flight in strong winds". *Science Robotics*. 7(66). DOI: 10.1126/scirobotics.abm6597.

OpenAI. (2023a). "ChatGPT Plugins". URL: https://openai.com/index/chatgpt-plugins/.

OpenAI. (2023b). "GPT-4 Technical Report". arXiv: 2303.08774 [cs.CL].

OpenAI *et al.* (2024). "GPT-4 Technical Report". arXiv: 2303.08774 [cs.CL].

Petersen, F. (2022). "Learning with Differentiable Algorithms". *arXiv preprint arXiv:2209.00616*.

Pnueli, A. (1977). "The temporal logic of programs". *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*: 46–57.

Radford, A., J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. (2021). "Learning Transferable Visual Models From Natural Language Supervision". URL: https://arxiv.org/abs/2103.00020.

Rajpurkar, P., J. Zhang, K. Lopyrev, and P. Liang. (2016). "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. DOI: 10.18653/v1/D16-1264.

Ramesh, A., M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. (2021). "Zero-shot text-to-image generation". In: *ICML*.

Rombach, R., A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. (2022). "High-Resolution Image Synthesis With Latent Diffusion Models". In: *CVPR*.

Sadigh, D., E. S. Kim, S. Coogan, S. S. Sastry, and S. A. Seshia. (2014). "A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications". In: *53rd IEEE Conference on Decision and Control*. IEEE. 1091–1096.

Saeed, M., N. Ahmadi, P. Nakov, and P. Papotti. (2021). "RuleBERT: Teaching Soft Rules to Pre-Trained Language Models". In: *EMNLP*.

Santoro, A., D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap. (2017). "A simple neural network module for relational reasoning". *NeurIPS*.

Schick, T., J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. (2023). "Toolformer: Language Models Can Teach Themselves to Use Tools". arXiv: 2302.04761 [cs.CL].

Shang, X., D. Di, J. Xiao, Y. Cao, X. Yang, and T.-S. Chua. (2019). "Annotating Objects and Relations in User-Generated Videos". In: *Proceedings of the 2019 on International Conference on Multimedia Retrieval*. ACM. 279–287.

Shang, X., T. Ren, J. Guo, H. Zhang, and T.-S. Chua. (2017). "Video visual relation detection". In: *Proceedings of the 25th ACM international conference on Multimedia*. 1300–1308.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. (2016). "Mastering the game of Go with deep neural networks and tree search". *Nature*. 529: 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Sinha, K., S. Sodhani, J. Dong, J. Pineau, and W. L. Hamilton. (2019). "CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text". arXiv: 1908.06177.

Soares, L. B., N. Fitzgerald, J. Ling, and T. Kwiatkowski. (2019). "Matching the Blanks: Distributional Similarity for Relation Learning". In: *ACL*.

Srivastava, A., A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, and et al. (2023). "Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models". arXiv: 2206.04615.

Sun, J. J., M. Tjandrasuwita, A. Sehgal, A. Solar-Lezama, S. Chaudhuri, Y. Yue, and O. Costilla-Reyes. (2022). "Neurosymbolic Programming for Science". URL: https://arxiv.org/abs/2210.05050.

Tenney, I., D. Das, and E. Pavlick. (2019). "BERT Rediscovers the Classical NLP Pipeline". In: *ACL*.

Thomee, B., D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. (2016). "YFCC100M: The New Data in Multimedia Research". *Communications of the ACM*. 59(2): 64–73.

Touvron, H., L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.* (2023). "Llama 2: Open Foundation and Fine-Tuned Chat Models". arXiv: 2307.09288.

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. (2018). "Graph Attention Networks". In: *ICLR*.

Wang, P.-W., P. L. Donti, B. Wilder, and Z. Kolter. (2019). "SATNet: Bridging Deep Learning and Logical Reasoning Using a Differentiable Satisfiability Solver". In: *International Conference on Machine Learning (ICML)*. arXiv: 1905.12149.

Watkins, C. J. C. H. (1989). "Learning from delayed rewards".

Wu, Y., M. Keoliya, K. Chen, N. Velingker, Z. Li, E. J. Getzen, Q. Long, M. Naik, R. B. Parikh, and E. Wong. (2024). "DISCRET: Synthesizing Faithful Explanations For Treatment Effect Estimation". URL: https://arxiv.org/abs/2406.00611.

Yang, J., W. Peng, X. Li, Z. Guo, L. Chen, B. Li, Z. Ma, K. Zhou, W. Zhang, C. C. Loy, and Z. Liu. (2023). "Panoptic Video Scene Graph Generation". arXiv: 2311.17058 [cs.CV].

Yang, Z., P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning. (2018). "HotpotQA: A dataset for diverse, explainable multi-hop question answering". arXiv: 1809.09600.

Yi, K., J. Wu, C. Gan, A. Torralba, P. Kohli, and J. Tenenbaum. (2018). "Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding". In: *Conference on Neural Information Processing Systems (NeurIPS)*.

Zhai, X., B. Mustafa, A. Kolesnikov, and L. Beyer. (2023). "Sigmoid Loss for Language Image Pre-Training". arXiv: 2303.15343 [cs.CV].

Zhu, C., Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé, M. Naik, Y. Shoshitaishvili, R. Wang, and A. Machiry. (2024). "TYGR: Type Inference on Stripped Binaries using Graph Neural Networks". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association. 4283–4300. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/zhu-chang.

Zhu, G., L. Zhang, Y. Jiang, Y. Dang, H. Hou, P. Shen, M. Feng, X. Zhao, Q. Miao, S. A. A. Shah, *et al.* (2022). "Scene graph generation: A comprehensive survey". *arXiv preprint arXiv:2201.00443*.