NEUROSYMBOLIC PROGRAMMING IN SCALLOP: DESIGN, IMPLEMENTATION, AND APPLICATIONS

Ziyang Li

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2025

Supervisor of Dissertation

Mayur Naik, Misra Family Professor of Computer and Information Science

Graduate Group Chairperson

Anindya De, Associate Professor of Computer and Information Science

Dissertation Committee

Rajeev Alur, Zisman Family Professor of Computer and Information Science
Chris Callison-Burch, Professor of Computer and Information Science
Armando Solar-Lezama, Distinguished Professor of Computing, MIT
Val Tannen, Professor of Computer and Information Science

NEUROSYMBOLIC PROGRAMMING IN SCALLOP:

DESIGN, IMPLEMENTATION, AND APPLICATIONS

To my parents, for their endless support. To Jiani, for walking every step with me.

ACKNOWLEDGEMENT

This dissertation is a culmination of not just years of research, engineering, and reflection, but also a deeply personal journey shaped by the wisdom, support, laughter, music, and love shared with many wonderful people. I owe each of you a note in this symphony.

First and foremost, I am endlessly grateful to my advisor, Mayur Naik. Mayur, your mastery of our field and your unwavering support have been a compass throughout my PhD. Your attention to detail and relentless pursuit of clarity have sharpened my thinking. Yet, it is your trust, your willingness to let me spend hours tinkering with engineering, and your pride in giving me the freedom to explore, that made all the difference. Thank you for giving me the opportunity to be your PhD student, even though we first met by chance, in an unplanned half-hour conversation. Thank you for giving me the stage at summer schools, letting me stand alongside accomplished professors to teach the Scallop language. And thank you for every false alarm about my papers being doomed—I've never felt so prepared for heartbreak, only to be blindsided by surprises of acceptance emails. Those moments became milestones. I feel truly blessed to have had you as my advisor.

To my partner, my love, and my comrade-in-research, Jiani Huang. There are no words that can fully capture what you've given me. From the very first walk through the campus of Penn to the final days of writing this dissertation, you've been with me. Our research conversations flowed like music, ideas bouncing in rhythm with scribbles on scratch paper. You are not just my intellectual partner, but also the heart of our band, the brightest star in every outfit, and the chef behind our most comforting dinners. Also, thank you for every dead joke you told, followed by my confused stare and your patient (and occasionally smug) explanation. I may never understand half of them, but I laughed anyway because they came from you.

I owe a lifelong debt of gratitude to my parents, Jixian Li and Hong Wang, and grandparents, Chengfang Wang, Shuhua Teng, Kedi Li, and Yuane Li, for their unwavering emotional and financial support through every chapter of my academic life. To my grandfather and grandmother, thank you for planting the seed. I still remember living in university faculty housing as a child, surrounded by

chalkboards, books, and the quiet hum of thought. The light of academia has been with me since the beginning, long before I knew the word "research." It is you who passed to me that passion to search for light, to find and share knowledge, and to chase down ideas like fireflies, no matter how late it gets.

To Rajeev Alur, thank you for being not just a committee chair, but a steadfast guide. Your insights both on research and the larger rhythm of academic life have helped me grow into the researcher and person I hoped to become. To my committee members Val Tannen, Chris Callison-Burch, and Armando Solar-Lezama, thank you for your thoughtful feedback, inspiring questions, and for shaping this work with your expertise and care. I extend my gratitude to the mentors who generously shared their guidance and belief in me: Baishakhi Ray and Isil Dillig. Thank you for your encouragement and support.

To the teachers who first set me on this path: Deian Stefan, thank you for introducing me to the beauty of programming languages and security; Ranjit Jhala, for letting me TA and discover the joy of logic programming; and Ravi Ramamoorthi, who opened my eyes to the magic of computer graphics and the wonder of research during my undergraduate days.

To my brilliant collaborators: thank you not only for your expertise but also for your friendship. I've been lucky to work with and learn from: Xujie Si, Elizabeth Dinella, Saikat Dutta, Aravind Machiry, Ke Wang, Yinjun Wu, Xin Zhang, Sernam-Lim, Hanjun Dai, Hanlin Zhang, Kexin Pei, Alex Gu, Binghong Chen, Anton Xue, Paul Biberstein, Claire Wang, Neelay Velingker, Mayank Keoliya, Alaia Solko-Breslin, Aaditya Naik, Adam Stein, Avishree Khare, Harsh Parekh, Shu Yang, Li Zhang, Qing Lyu, Pengyuan Lu, Fengjun Yang, Junyao Shi, Zhiyang Wang, Mingyuan Zhang, and Yishuai Li. To my mentees, especially Jason Liu, thank you for trusting me to walk alongside you, even as I was learning myself.

To my bandmates, who brought music into my research life every other week: Li Zhang, Zhenglong Zhou, Yilong Huang, Muyang Cheng, Chenjie Song, Colin Ly, and of course, Jiani Huang again. Thank you for the joy, the jamming, and the shared passion that transcended the work. Special

thanks to all my labmates who came out to cheer us on. You made every performance feel like home.

And most importantly, to my inner circle, who supported me through every challenge and triumph. A special note of gratitude to my high school classmates who grew into fellow PhD travelers: Weichen Liu, Yizhe Huang, Qiwei Dong, Risheng Tan, and Siyang Zhang. You kept me grounded, lifted me up when I faltered, and made this long journey feel less solitary. This dissertation is not mine alone; it is a mosaic of all the people who believed in me, piece by piece, moment by moment.

With all my heart, thank you.

ABSTRACT

NEUROSYMBOLIC PROGRAMMING IN SCALLOP: DESIGN, IMPLEMENTATION, AND APPLICATIONS

Ziyang Li

Mayur Naik

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

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

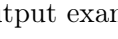
CHAPTER 1

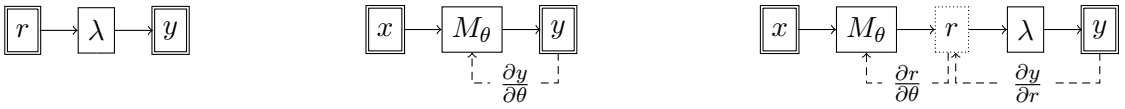
INTRODUCTION

1.1 Neurosymbolic Programming

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

The two paradigms are complementary in nature. For instance, a classical algorithm such as the logic program λ shown in Figure 1.1a is interpretable but operates on limited, structured input r. On the other hand, a deep neural network such as M_θ shown in Figure 1.1b can operate on rich, unstructured input x but is not interpretable. Modern applications demand the capabilities of both paradigms. Examples include question answering (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 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 hand-written formula evaluation (Li et al., 2020a), 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 $\langle x = \text{  }, y = 1.6 \rangle$. A neurosymbolic program for this



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

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

task, such as the one shown in Figure 1.1c, might first apply a convolutional neural network M_θ to the input image to obtain a structured intermediate form r as a sequence of symbols [‘1’, ‘+’, ‘3’, ‘/’, ‘5’], followed by a classical algorithm λ to parse the sequence, evaluate the parsed formula, and output the final result 1.6.

Despite significant strides in individual neurosymbolic applications (Yi et al., 2018; Mao et al., 2019; Chen et al., 2020c; Li et al., 2020a; Minervini et al., 2020; 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 λ that expresses common reasoning patterns such as recursion, negation, and aggregation.

3. An automatic and efficient differentiable reasoning engine for learning ($\frac{\partial y}{\partial r}$) under *algorithmic supervision*, i.e., supervision on observable input-output data (x, y) but not r.

4. The ability to tailor learning ($\frac{\partial y}{\partial r}$) to individual applications’ characteristics, since non-continuous loss landscapes of symbolic programs hinder learning using a one-size-fits-all method.

5. A mechanism to leverage and integrate with existing training pipelines ($\frac{\partial r}{\partial \theta}$), implementations of

neural architectures and models M_θ, and hardware (e.g. GPU) optimizations.

In addition to the above criteria, we identified three general challenges that any practical neurosymbolic system must address:

Programmability Many neurosymbolic approaches are hard-coded or specialized to specific tasks, requiring deep expertise in both symbolic systems and machine learning. This leads to brittle pipelines and poor developer ergonomics. A major challenge is to design an abstraction that enables high-level, concise, and maintainable programs while preserving expressivity. Programmers should be able to declaratively specify reasoning tasks without managing low-level numerical details or system-specific glue code.

Scalability Neurosymbolic systems must reason over large symbolic structures (e.g., knowledge graphs, programs, scene graphs) while integrating with high-dimensional neural representations. However, symbolic reasoning engines traditionally lack the performance characteristics—such as parallelism and efficient memory usage—needed to scale to real-world data sizes. At the same time, differentiable execution over symbolic rules introduces additional computational burdens that strain typical inference engines. Thus, achieving efficient and scalable reasoning across multiple modalities remains a significant barrier.

Adaptability Applications of neurosymbolic methods vary widely from vision and language tasks to planning, security, and biology. Yet most existing tools are tightly coupled to specific representations or reasoning semantics. A general-purpose system must support multiple reasoning modes (e.g., discrete, probabilistic, differentiable), flexible integration with neural components, and extensibility for domain-specific customization. Achieving this adaptability without sacrificing coherence or performance is a key technical challenge.

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 and even molecular structures. Second, a declarative language for symbolic reasoning allows computation to be expressed concisely via high-level rules, thereby alleviating programmer effort. Finally, the relational paradigm offers a suitable abstraction for advanced features needed for neurosymbolic programming, such as query planning, hardware (GPU) acceleration, and probabilistic and differentiable reasoning.

Scallop is a Datalog-based programming language (Abiteboul et al., 1994) that allows rules to be written as Horn clauses (Robinson, 1965). It is strongly typed, and symbolic data is represented uniformly as predicate-named tuples. Operations over symbolic values—such as strings, integers, and booleans—are supported through a general foreign function interface. Scallop also supports stratified negation and stratified aggregation, enabling reasoning over multiple tuples with well-defined semantics.

To support probabilistic and differentiable reasoning, Scallop employs a provenance framework based on abstract tags. This design abstracts away the underlying algorithmic complexity, allowing programmers to focus purely on symbolic reasoning. The framework unifies discrete, probabilistic, and differentiable reasoning modes under a single abstraction. Users can select from a built-in library of provenances tailored to different reasoning needs, or implement custom provenance modules to extend the system's capabilities.

To bridge symbolic values and provenance tags, Scallop provides extensible systems for foreign predicates and aggregations, further enhancing its expressiveness. These features make Scallop a

versatile platform for developing neurosymbolic programs that combine high-level symbolic logic with low-level numerical computation. 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.

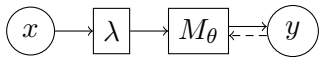
Moreover, as a rapidly evolving area, neurosymbolic research continues to generate new building blocks and design patterns. A key challenge is to build a system that can adapt to these emerging paradigms, offering a unified interface that enables their composition and evaluation. In this dissertation, we will characterize several of these building blocks and show how the Scallop system can be used to instantiate and experiment with them across a wide range of applications.

1.3 Building Blocks for Neurosymbolic Methods

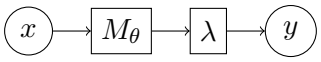
A language that integrates neural and symbolic components can be applied to construct diverse and adaptable solutions. Broadly, a neurosymbolic solution to any given task involves the flexible interplay of neural and symbolic components, each serving distinct yet complementary roles in problem-solving.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

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

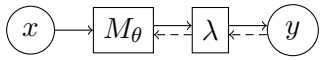
Notably, advanced feature extraction goes beyond simple tabular data and often incorporates sophisticated reasoning mechanisms to construct complex data structures. For instance, in program analysis, source code can be pre-processed into intricate structures such as abstract syntax trees



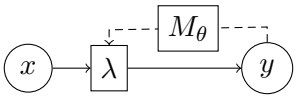
(a) Feature extraction



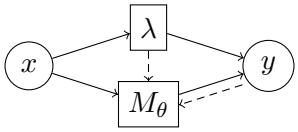
(b) Symbolic inference



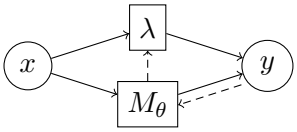
(c) Algorithmic supervision



(d) Neurosymbolic program synthesis



(e) Neural relaxation



(f) Symbolic distillation

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

(ASTs), data-flow graphs, symbolic constraints, or relational databases (Dinella et al., 2020; Li et al., 2021d; 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_θ using a symbolic component λ provided by a programmer. This analysis can serve various purposes, such as filtering nonsensical outputs, verifying output integrity, or combining multiple information sources symbolically to derive additional insights. Though straightforward in concept, an advanced symbolic inference component λ may handle probabilistic information, deriving a distribution rather than just the most likely output.

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

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

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

Algorithmic supervision also functions as a form of weak supervision, as it does not require direct, fully supervised labels for M_θ; only the end label y is needed. This reduces the need for extensive data labeling or feature engineering, simplifying the training process. Numerous applications in Scallop leverage this approach, including the previously mentioned task of learning to evaluate handwritten formulas (Li et al., 2020a, 2023b). This dissertation explores additional, advanced examples of algorithmic weak supervision in Chapter 6.

Neurosymbolic program synthesis Neurosymbolic program synthesis involves learning the symbolic program λ with the support of neural networks. This paradigm resembles the classical syntax-guided synthesis task (Alur et al., 2013), but replaces the traditional algorithmic synthesis procedure with a neural network M_θ. Here, the symbolic program λ is responsible for generating the expected outputs, and it may be iteratively refined to better align with a dataset.

This approach offers the advantage of interpretability, as the learned symbolic component is a white-box program that can be inspected and verified by humans (Ellis et al., 2022). Traditionally, synthesizing λ 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 λ can now be achieved more efficiently (Ma et al., 2024).

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

For instance, consider the challenge of designing a neurosymbolic controller for drones: while effective deterministic controllers exist for standard maneuvers, they may struggle to adapt to out-of-domain scenarios, such as operating near the ground, in strong winds, or in proximity to other drones. By relaxing certain aspects of the controller into a neural network M_θ, the system gains greater flexibility and responsiveness in handling such scenarios, while being able to learn rapidly (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 λ. Although this process involves generating and refining λ, similar to neurosymbolic program synthesis, symbolic distillation focuses on translating otherwise uninterpretable weights from a well-trained neural network M_θ into an interpretable form.

This technique has been applied to scientific discovery in fields such as animal behavior analysis (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 Contributions

This dissertation makes contributions along the dimensions of language design, system implementation, and application-driven evaluation. The key contributions are as follows:

Design of the Scallop language for neurosymbolic programming

1. Designed the core Scallop language, a Datalog-based, declarative, and strongly typed language for symbolic reasoning.
2. Developed a foreign interface for integrating external modules and databases.
3. Introduced a general relational interface for invoking and coordinating foundation models.
4. Developed a plugin library for Scallop, supporting large language models and vision-language models.

Unified semantics via provenance framework

1. Introduced a compiler from Scallop to SclRam, a low-level intermediate language based on relational algebra.
2. Designed a provenance framework for SclRam, enabling tagged symbolic computation.
3. Formalized the semantics of SclRam with provenance support.
4. Proposed practically useful provenances for descrete, probabilistic, and differentiable reasoning.
5. Introduced top-k-proofs, top-bottom-k-proofs, and optimal-k-proofs provenance for scalable approximated probabilistic inference.

Tooling and system integration

1. Developed an end-to-end implementation of the Scallop programming language, including (a) a compiler supporting type inference, syntax desugaring, and query planning, and (b) a runtime

with a low-level in-memory database and support for GPU execution.

2. Implemented native support for PyTorch and seamless integration with foundation models.

3. Designed extensible system for foreign predicates and aggregations to bridge symbolic and numerical computation.

Application to diverse domains and neurosymbolic paradigms

1. Applied Scallop to synthetic neurosymbolic reasoning tasks, including MNIST-R, Hand-Written Formula, and PathFinder.

2. Proposed and evaluated a new synthetic planning benchmark, PacMan-Maze, using Scallop for neurosymbolic planning.

3. Developed neurosymbolic solutions for natural language rasoning, including CLUTRR (kinship reasoning).

4. Applied Scallop to visual question answering benchmarks: CLEVR and VQAR.

5. Demonstrated Scallop's use with foundation models on a wide range of benchmarks: GSM8K, Date Reasoning, Tracking Shuffled Objects, Amazon Product Search, among others.

6. Demonstrated the use of Scallop for video scene graph generation, which results in the training of SGClip, a foundation model for multi-modal scene graph generation.

7. Illustrated the use of Scallop on the application of whole-project vulnerability detection, including the construction of a new dataset (CWE-Bench-Java).

8. Proposed a neurosymbolic solution to RNA secondary structure prediction, with an evaluation on the ArchiveII dataset.

1.6 Thesis Structure

The remainder of this dissertation is organized as follows. Chapter 2 introduces the Scallop language and its foundational constructs. Chapter 3 presents the provenance framework that powers Scallop's unified reasoning backend, including several proposed probabilistic and differentiable provenances that enhance its versatility. Chapter 4 describes how Scallop integrates with neural and foundation models via foreign interfaces. Chapter 5 evaluates Scallop on a suite of standard neurosymbolic benchmarks, as well as on tasks involving foundation models. In the chapter, we present end-to-end programs and case studies to illustrate Scallop's expressiveness and practical utility.

We then cover three advanced applications where the neurosymbolic paradigm and Scallop is applied. 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 explore the following three tasks: video scene graph generation in computer vision (Chapter 6), vulnerability detection in cybersecurity (Chapter 7), and RNA secondary structure prediction in bioinformatics (Chapter 8).

Parts of this dissertation are based on previously published or submitted work. In particular, Chapter 2 draws from material presented in Li et al. (2023b) and Li et al. (2024b), Chapter 3 is based on Li et al. (2023b) and Huang et al. (2021), Chapter 4 incorporates content from Li et al. (2024c) and Li et al. (2024b), and Chapter 5 incorporates benchmarks and evaluations from both Li et al. (2023b) and Li et al. (2024c). The advanced applications in Chapters 6, 7, and 8 are partially based on Huang et al. (2025), Li et al. (2025), Biberstein et al. (2025), and ongoing work currently under submission.

CHAPTER 2

BASICS OF PROGRAMMING IN SCALLOP

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

2.1 Relations, Data Types, and Facts

The fundamental data type in Scallop is a relation which comprises a set of tuples of statically-typed primitive values. The primitive data types include signed and unsigned integers of various sizes (e.g. `i32`, `usize`), single- and double-precision floating point numbers (`f32`, `f64`), boolean (`bool`), character (`char`), and string (`String`). A comprehensive list is provided in Table 2.1. For example, Listing 2.1 declares two binary relations, `mother` and `father`. Note that we declare multiple relations with one `type` keyword. Values of relations can be specified via individual tuples or a set of tuples of constant literals, as shown in line 5 and line 8 in Listing 2.1. The type of facts must conform to the statically declared relation type. All the tuples under `mother` and `father` are of arity 2 and both elements are strings. Note that the keyword `rel` is chosen as a shorthand for `relation`, which is used to define relations.

As a shorthand, primitive values can be named and declared as constant variables, as shown in line 2 in Listing 2.2. Type declarations are optional since Scallop supports type inference. The type of the `composition` relation is inferred as `(usize, usize, usize)` since the default type of 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`).

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

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

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

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

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

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

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

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

2.1.1 Nullary, Unary, and Binary Relations

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

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

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

```
1 // Declaration of the type of a 0-arity relation
2 type is_target()
3
4 // Declaring a single fact
5 rel is_target()
6
7 // Declaring a set of facts but with only a single empty tuple
8 rel is_target = {()}
```

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

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

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

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

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

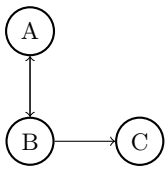


Figure 2.1: A sample graph with three nodes.

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

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

2.1.2 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.

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

```
> 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.

```
// A few facts under the base relations
rel father = {("Bob", "Alice"), ("John", "Harry")}
rel mother = {("Christine", "Bob")}

// Father's mother is grandmother
rel grandmother(c, a) = mother(c, b) and father(b, a)
// Mother's mother is also grandmother
rel grandmother(c, a) = mother(c, b) and mother(b, a)

// == is equivalent to... ==

// Mother or father's mother is grandmother
rel grandmother(c, a) = mother(c, b) and
                        (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."

`c` can be grounded with "Christine", `b` can be grounded with "Bob", while `a` can be grounded with "Alice". Notably, the variable `b` appears in both the `mother(c, b)` atom as well as the `father(b, a)` atom, meaning that the value being used to ground the variable `b` has to appear in both facts.

In a rule, conjunction is specified using `and`-separated atoms within the rule body whereas disjunction can be specified by multiple rules with the same head predicate. Each variable appearing in the head must also appear in some positive atom in the body. Conjunctions and disjunctions can also be expressed using logical connectives like `and`, `or`, and `implies`. For instance, the last rule (line

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

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

9-10 of Listing 2.7) is equivalent to the two rules above combined.

Scallop performs a few compilation checks to ensure that the program is well-formed. First of all, the rules need to type check. In the case of Listing 2.7, all the shown relations are binary `String` relations, and therefore type inference succeeds. Moreover, all the variables appearing in the head atom must be *bounded* by atoms in the body. Consider the first rule (line 2) as an example, in which variable `a` is bounded by the `father` relation, while variable `c` is bounded by `mother`. Therefore, the head atom of the rule is bounded and well-formed. For the last rule (line 9-10) where the body contains disjunctions, head variables need to be bounded for all branches in the body. This is indeed true since `a` is bounded by both atoms in the disjunction.

Scallop supports value creation by means of foreign functions (FFs). FFs are polymorphic and include arithmetic operators such as `+` and `-`, comparison operators such as `!=` and `>=`, type conversions such as `[i32] as String`, and built-in functions like `$hash` and `$string_concat`. They only operate on primitive values but not relational tuples or atoms. Listing 2.8 shows a few examples. Specifically, the first shows that floating point weight and height can be used to compute body mass index (BMI). In the second example, strings are concatenated together using FF, producing the result `full_name("John Doe")`.

Note that FFs can fail due to runtime errors such as division-by-zero and integer overflow, in which

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

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

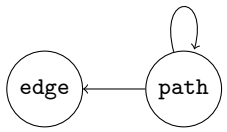


Figure 2.2: The dependency graph associated with the `edge-path` program.

case the computation for that single fact is omitted. In the last example shown in Listing 2.8 (line 10-12), when dividing 6 by `denominator`, the result is not computed for denominator `0` since it causes a FF failure. The purpose of this semantics is to support probabilistic extensions rather than silent suppression of runtime errors. When dealing with floating-point numbers, tuples with `NaN` (not-a-number) are also discarded.

2.3 Recursion, Negation, and Aggregation

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

2.3.1 Recursion

A powerful programming construct in Scallop is to declaratively define recursion. Within a rule, if a relational predicate appearing in the head appears in the body, the rule is recursive. More generally, a relation r is dependent on s if an atom s appears in the body of a rule with head atom r. A *recursive* relation is one that depends on itself, directly or transitively. For instance, Listing 2.9 shows a program with recursion. In the program, `path` depends on `edge` (line 5-6) and `path` itself (line 6). Based on this information, we can draw a dependency graph for the program, shown in Figure 2.2. Since there is a self-loop on the `path` relation, we say that the program is recursive.

Recursion is also very useful in recursive mathematical definitions. For example, the definition of

```
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.

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 sum `y1 + y2` where `y1` $=$ fib$(x-1)$ and `y2` $=$ fib$(x-2)$. This almost literally translates to the recursive rule on line 4. We note that an extra constraint `x > 1` must be added in order for the computation to terminate. At the end, when the atom `fib(5, y)` is queried, Scallop will return that a fact `fib(5, 8)` suggesting that 8 is the result of computing fib(5).

2.3.2 Negation

Scallop supports stratified negation using the `not` operator on atoms in the rule body. Listing 2.11 shows a rule defining the `has_no_children` relation as any person `p` who is neither a father nor a mother (line 7-8). In the rule, the underscore (`_`) stands for *wildcard* which is used to match any value. Note that we need to bound `p` by a positive atom `person` in order for the rule to be well-formed. In the rule that does not compile, the variable `p` can be anything other than `"Bob"` or `"Christine"`, meaning that it is impossible to enumerate the values. Scallop rejects these kinds of programs by ensuring that all variables that occur in the head are bounded by positive atoms in the body. At the end, we have the relation `has_no_children` containing one single tuple `("Alice")`, 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  rel father("Bob", "Alice")      // Bob is Alice's father
4  rel mother("Christine", "Bob") // Christine is Bob's mother
5
6  // Compute the person who has no children
7  rel has_no_children(p) = person(p) and
8      not father(p, _) and not mother(p, _)
9
10 // !! This rule does not compile: p is not bounded !!
11 rel error(p) = not father(p, _) and not mother(p, _)
```

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

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

Listing 2.12: A rule with a negative circular dependency—where the predicate `something_is_true` depends on itself—will lead the compiler to reject the program.

A relation r is *negatively* dependent on s if a negated atom s appears in the body of a rule with head atom r. In the example shown in Listing 2.11, `has_no_children` negatively depends on `father`. A relation cannot be negatively dependent on itself, directly or transitively, as Scallop supports only stratified negation. The rule shown in Listing 2.12 is rejected by the compiler, as the negation is not stratified.

2.3.3 Aggregation

Scallop also supports stratified aggregation. We use the assignment symbol `:=` to retrieve the results obtained from aggregations. The set of built-in aggregators include common ones such as `count`, `sum`, `max`, and first-order quantifiers `forall` and `exists`. Besides the operator, the aggregation construct specifies the binding variables, the aggregation body to bound those variables, and the result variable(s) to assign the result. The rule with aggregation in Listing 2.13 reads, "variable `n` is assigned the count of `p`, such that `p` is a person". Specifically, `n` is the result of the aggregation, `count` is the aggregator, `p` is the qualified variable for aggregation, and `person(p)` is the body of the

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

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

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

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 group-by variable, since it appears in both the aggregation body (`parent(p, c)`) and the head of the rule (`num_child(p, n)`). However, depending on whether we explicitly bound the group-by variable `p`, we get different results. On line 11, we explicitly use a `where` clause to bound the variable `p` with everyone in the `person` relation. As such, we would also find the number of children of `"Alice"`,

```
1 rel father("Bob", "Alice")   // Bob is Alice's father
2 rel daughter("Alice", "Bob") // Alice is Bob's daughter
3
4 // An integrity constraint for kinship graphs
5 rel integrity_constraint(sat) = sat := forall(a, b:
6         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.

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 be presented in the result.

Finally, quantifier aggregators such as `forall` and `exists` return one boolean variable. For instance, for the aggregation shown in Listing 2.15, variable `sat` is assigned the truthfulness (`true` or `false`) of the following statement: "for all `a` and `b`, if `b` is `a`'s father, then `a` is `b`'s son or daughter". At the end, we would obtain a fact `integrity_constraint(true)`, meaning that the constraint is satisfied given the kinship facts shown on line 1-2.

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

2.4 Programming with Probabilities

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

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

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

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

```
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.

between Alice (`A`) and Bob (`B`). As shown in Listing 2.16, we list out all possible kinship relations between Alice and Bob. For each of them, we use the syntax `[PROB]::[TUPLE]` to tag the kinship tuples with probabilities. The semicolon "`;`" separating them specifies that they are mutually exclusive—Bob cannot be both the mother and father of Alice.

Scallop also supports operators to sample from probability distributions. They share the same surface syntax as aggregations, allowing sampling with group-by. The following rule shown in Listing 2.17 deterministically picks the most likely kinship relation between a given pair of people `a` and `b`, which are implicit group-by variables in this aggregation. As the end, only one fact, `0.95::top_1_kinship(FATHER, A, B)`, will be derived according to the above probabilities. Other types of sampling are also supported, including categorical sampling (`categorical<K>`) and uniform sampling (`uniform<K>`), where a static constant `K` denotes the number of trials.

Finally, rules can also be tagged by probabilities which can reflect their confidence. The rule shown in Listing 2.18 states that a grandmother's daughter is one's mother with 90% confidence. Probabilistic rules are syntactic sugar. They are implemented by introducing in the rule's body an auxiliary

```
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) = grandmother(a,b) and daughter(b,c)
4
5 // == the above rule is desugared to... ==
6 rel 0.9::prob_of_rule() // one auxilliary nullary relation
7 rel mother(a,c) = grandmother(a,b) and daughter(b,c) and
8                    prob_of_rule()
```

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

nullary (i.e., boolean) fact that is regarded true with the tagged probability.

2.5 On-Demand Computations

In normal Scallop, facts are computed in a bottom-up fashion. That is, for each rule, we start from grounding the body with existing facts, and derive the fact in the head. Typically, this would derive all possible outcomes for a relation, which may be costly. Worse, it may even be impossible to derive fully due to the derived relation being infinite. One example is the computation of Fibonacci number (also shown previously in Listing 2.10). Fibonacci number itself is infinite, so given the base cases for 0 and 1, it is expected that the computation for all Fibonacci numbers will never terminate. Such Scallop program is shown in Listing 2.19. However, often times we have a specific query for these inifinite relations. As shown on line 6 in Listing 2.20, we are querying for the 5th Fibonacci number, and nothing else is expected. For such cases, we might use *on-demand computation* to answer those queries, without computing the full infinite relation. Specifically, the number `5` is the *demand* for the `fib` relation.

We achieve on-demand computation in Scallop by doing the following (Listing 2.20). First, as shown on line 2, we add a `bound` keyword to the `x` variable when defining the `fib` relation. This is called an *adornment*, meaning that everytime the relation `fib` is computed, we are treating the first argument `x` as the input. For the second variable `y` that is not adorned by the `bound` keyword, it means that the value will be derived by rules. A variable not adorned by `bound` is treated a *free* variable. We say that `bound-free` (or *bf* in short) is the on-demand pattern for the `fib` relation.

```
type fib(x: i32, y: i32)
rel fib = {(0, 1), (1, 1)}
rel fib(x, y1 + y2) = fib(x - 1, y1) and fib(x - 2, y2)
// 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.

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

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

Without specification, normal relations have an *all-free* on-demand pattern, which means they are *not* on-demand relations.

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

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

```
1 // Type defs; 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 // Suppose we have a 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) or (edge(x, z) and path(z, y))
9 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.

2.6 Algebraic Data Types

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

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

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

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) // adding two expressions
3            | Sub(Expr, Expr) // subtracting two expressions
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 defined in Scallop that represents the small language shown in Figure 2.3 (line 1-3). Line 4 shows one expression $1 + (3 - 2)$ expressed using `Expr` type.

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

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

the respective results to form the final result.

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

2.7 Foreign Interface

Scallop supports a foreign interface which allows external definition of functions, predicates, and attributes. These constructs allow Scallop to be effective in diverse applications, including a tight integration of foundation models, which we describe in detail in Chapter 4. In this section we describe such constructs and a selection of standard library containing interfaced items. We note that the code snippets in this section may show the use of `extern` keyword, suggesting the declaration of externally defined items. However, during normal use of Scallop, such declarations are not neccessary and most foreign constructs are imported automatically.

2.7.1 Foreign Functions

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

Generic functions When defining the type of a function, we may use an additional angle brackets `<...>` after the function name, to specify the generic type parameters. Each type parameter may be followed by a type family to give additional constraint on the type. For instance, the `$abs` function shown in Listing 2.24 is a generic function with one type parameter, `T`, that needs to be a `Number`. Signed or unsigned integers as well as floating point numbers are types under the family `Number`. The absolute value function is properly defined on any of such data types.

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

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

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

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

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

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

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

value of any type to be passed into the function.

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

Optional argument When specifying the type of an argument to the function, we may add a question mark (`?`) at the end of the type annotation to denote that the argument is optional (e.g., `usize?`). 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 nobs to allow the report of errors.

2.7.2 Foreign Predicates

Foreign predicate is a generalized interface of foreign function, which can now produce multiple outputs associated with additional informations such as probabilities. Predicates are mostly declared just like other relations in Scallop, where inputs should be associated with `bound` keywords while

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

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

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

Listing 2.25: An example foreign predicate `string_chars`.

outputs may be associated with `free` keywords. Here, we add the `extern` keyword to denote that the `PREDICATE` should be defined externally.

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

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

In Listing 2.25 we showcase one foreign predicate `string_chars`, that could help in obtaining the nucleotides ($\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$) in an RNA sequence string. Taking the string `s` as an input, `string_chars` produces `(s, i, n)` triplets where `i` is the index of a character in the string, and `n` is the character itself. It is clear that `string_chars` returns multiple facts as the output, whereas foreign functions introduced in the previous section can only return one output.

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

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

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

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

based on their distance. Formally, it is defined as follows:

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

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

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

2.7.3 Foreign Attributes

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

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

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

Listing 2.27: A CSV file storing edges.

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

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

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

CHAPTER 3

CORE REASONING PROVENANCE FRAMEWORK

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

We start by presenting the basics of our provenance framework (Section 3.1). We then present a low-level representation SclRam, its operational semantics, and its interface to the rest of a Scallop application (Sections 3.2-3.3). We next present how our provenance framework enables scalable 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 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.

A provenance must satisfy a few properties. First, $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ should form a commutative semiring.

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

Figure 3.1: Core algebraic interface for provenance T.

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.

Example 1. *`max-min-prob` (`mmp`)* $\triangleq ([0,1], 0, 1, \mathit{max}, \mathit{min}, \lambda x.(1-x), \texttt{==})$*, 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.*

In particular, the `mmp` provenance extends the standard fuzzy semiring, also known as the Gödel semiring (Cintula et al., 2011). While variants of the fuzzy semiring have been successfully applied in neurosymbolic systems, such as in visual reasoning models like NS-CL (Mao et al., 2019), these basic formulations lack key operations required for our use case. To support more expressive reasoning patterns in our provenance framework, we extend the fuzzy semiring with two additional operations: negation, to model logical complement or absence of belief, and saturation, to handle fixed-point reasoning or convergence under iterative updates. We discuss richer provenances for more accurate probability calculations later in this chapter.

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

Figure 3.2: Abstract syntax of core fragment of SclRam.

3.2 SclRam Intermediate Language

Scallop programs are compiled to a low-level representation called SclRam. Figure 3.2 shows the abstract syntax of a core fragment of SclRam. Expressions resemble queries in an extended relational algebra. They operate over relational predicates (p) using unary operations for aggregation (γ_g with aggregator g), projection (π_α with mapping α), and selection (σ_β with condition β), and binary operations union ($\cup$), product ($\times$), join ($\bowtie$), difference ($-$), and anti-join ($\triangleright$). We note that there are other binary operations such as intersection ($\cap$) which could be expressed by combining the above core operations. Note that the projection function α is able to pick out elements and also apply foreign functions, such as summation + and `$string_concat`, on tuple values. We elide constructs such as natural join and anti-join since they can be encoded by combining other operations.

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

We next define the semantic domains in Figure 3.3 which are used subsequently to define the semantics of SclRam. A tuple u is either a constant or a sequence of tuples. A fact $p(u) \in \mathbb{F}$ is a tuple u named under a relational predicate p. Tuples and facts can be tagged, forming *tagged tuples*

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

Figure 3.3: Annotations of semantic domains for ScLRam.

$(t :: u)$ 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.

Throughout our discussion of ScLRam semantics, we use a motivating example shown in Figure 3.5 to illustrate the reasoning procedure. The example depicts a maze in which a PacMan must reach a flag while avoiding enemies. For illustrative purposes, we assume that the positions of PacMan, enemies, and the flag are uncertain, each associated with a probability in the range $[0, 1]$. Using this example, we walk through the components of ScLRam, demonstrating how a high-level Scallop program is compiled into ScLRam and how the resulting ScLRam program is subsequently evaluated.

Basic relational algebra Evaluating an expression in ScLRam yields a set of tagged tuples according to the rules defined at the top of Figure 3.4. A predicate p evaluates to all facts under that predicate in the database. Selection filters tuples that satisfy condition β, and projection transforms

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

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

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

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

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

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

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

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

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

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

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

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

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

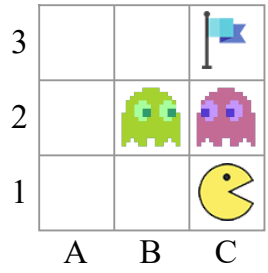
(Saturation) $F_T^{\texttt{old}} \stackrel{\circ}{=} F_T^{\texttt{new}}$ iff $\forall t^{\texttt{new}} :: p(u) \in F_T^{\texttt{new}}, \exists t^{\texttt{old}} :: p(u) \in F_T^{\texttt{old}}$ such that $t^{\texttt{old}} \ominus t^{\texttt{new}}$

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

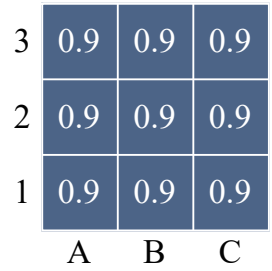
(Stratum) $[\![s]\!] = \textbf{lfp}^\circ(\lambda F_T.(F_T - \bigcup_{p \in P_s} F_T[p]) \cup (\bigcup_{r \in s} [\![r]\!](F_T)))$

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

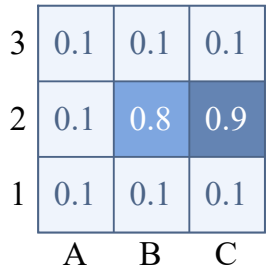
Figure 3.4: Operational semantics of core fragment of SclRam.



(a) Maze illustration



(b) `grid_cell`



(c) `enemy`

```
1  const A = 1, B = 2, C = 3
2
3  type grid_cell(i: usize, j: usize)
4  rel grid_cell = {0.9::(3, A), 0.9::(3, B), 0.9::(3, C),
5                   0.9::(2, A), 0.9::(2, B), 0.9::(2, C),
6                   0.9::(1, A), 0.9::(1, B), 0.9::(1, C) }
7
8  type enemy(i: usize, j: usize)
9  rel enemy     = {0.1::(3, A), 0.1::(3, B), 0.1::(3, C),
10                  0.1::(2, A), 0.8::(2, B), 0.9::(2, C),
11                  0.1::(1, A), 0.1::(1, B), 0.1::(1, C) }
```

(d) The Scallop program representing the maze shown in (a), where grid cells and enemies are represented by `grid_cell` and `enemy` relations.

Figure 3.5: A grid based maze used as a motivating example in this section. Shown in (a), the grid is 3×3 with a PacMan located in location $(1, \mathrm{C})$, two enemies located in locations $(2, \mathrm{B})$ and $(2, \mathrm{C})$, and the goal flag located in location $(3, \mathrm{C})$. For illustration purpose, we assume entities are located in a given cell with a certain probability shown in (b) and (c). The Scallop program that describes the maze configuration is shown in (d).

tuples according to mapping α. The mapping function α is partial: it may fail since it can apply foreign functions to values. A tuple in a union $e_1 \cup e_2$ can come from either e_1 or e_2. In a (Cartesian) product $e_1 \times e_2$, each pair of incoming tuples is combined, and we use the provenance multiplication $\otimes$ to compute their tags.

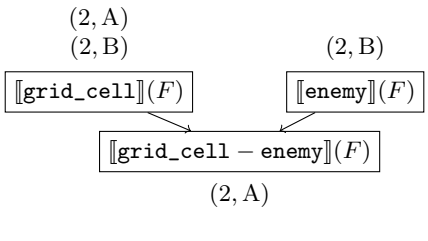
Difference and negation To evaluate a difference expression $e_1 - e_2$, there are two cases depending on whether a tuple u evaluated from e_1 appears in the result of e_2. If it does not, we simply propagate the tuple and its tag to the result (DIFF-1); otherwise, we get $t_1 :: u$ from e_1 and $t_2 :: u$ from e_2. Instead of erasing the tuple u from the result as in untagged semantics, we propagate a tag

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

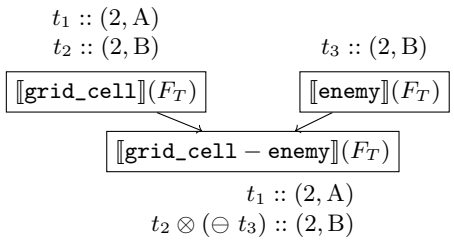
(a) A Scallop program computing the safe cells

$$\texttt{safe\_cell} \leftarrow \texttt{grid\_cell} - \texttt{enemy}$$

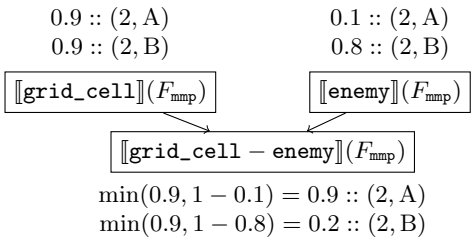
(b) The compiled ScLRam program from (a)



(c) Untagged semantics



(d) ScLRam tagged semantics



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

Figure 3.6: In (a), we demonstrate a Scallop rule computing the `safe_cells`, which are cells that do not contain an enemy. The rule makes use of negation, and the compiled ScLRam code, shown in (b) involves a difference operation ($-$) on `grid_cell` and `enemy` relations. Figures (c), (d), and (e) illustrate evaluation of the ScLRam code under different semantics, where (e) instantiates the tagged semantics with `max-min-prob` provenance.

$t_1 \otimes (\ominus\, t_2)$ with u (Diff-2). In this manner, information is not lost during negation. Figure 3.6c and Figure 3.6d compare the evaluations of a difference expression under different semantics. While the tuple $(2, \text{B})$ is removed from the outcome under untagged semantics, it is preserved under the tagged semantics, albeit with lesser probability.

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

```
rel num_enemies(n) = n := count(x, y: enemy(x, y))
```

(a) A Scallop program counting the number of enemies.

$$\texttt{num\_enemies} \leftarrow \pi_{\lambda n.(n)}(\gamma_{\texttt{count}}(\texttt{enemy}))$$

(b) The compiled ScLRam program utilizing aggregation operator $\gamma_{\texttt{count}}$.

$\langle[\![\gamma_{\texttt{count}}(\texttt{enemy})]\!](F_T)\rangle$	$\langle[\![\gamma_{\texttt{count}}(\texttt{enemy})]\!](F_{\texttt{mmp}})\rangle$
:: 0	0.1 :: 0
$\oplus \cdots \oplus$ :: 1	0.1 :: 1
$\oplus \cdots \oplus \cdots \oplus$ :: 2	0.8 :: 2
...	...
:: 9	0.1 :: 9

(c) Evaluation of the aggregation operator $\gamma_{\texttt{count}}$. Each symbol such as ▪□□ represents a world corresponding to our maze (■: enemy; □: no enemy). A world is a conjunction of 9 tags, e.g., ▪□□ $= t_{\texttt{enemy(3,A)}} \otimes (\ominus t_{\texttt{enemy(3,B)}}) \otimes \cdots \otimes (\ominus t_{\texttt{enemy(1,C)}})$. We mark the correct world □■■ which yields the answer 2.

Figure 3.7: An example counting enemies in the PacMan maze shown in Figure 3.5a. Shown in (a) and (b) are the Scallop rule and compiled ScLRam rule with aggregation. In (c), we show two normalized ($\langle . \rangle$ defined in Figure 3.4) evaluation results under abstract tagged semantics and with `mmp` provenance. Under the `mmp` provenance, the outcome with two enemies has the highest probability, which aligns with our intuition.

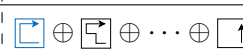
the probabilistic domain, discrete symbols do not suffice. Given n tagged tuples to aggregate over, each tagged tuple can be turned on or off, resulting in 2^n distinct *worlds*. Each world is a partition of the input set U_T ($|U_T| = n$). Denoting the positive part as X_T and the negative part as $\overline{X}_T = U_T - X_T$, 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 3.7 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.

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

(a) The Scallop program computing whether there is a path from (x, y) to (u, v) without an enemy within the path, using transitive closure.

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

(b) The two Scallop rules in (a) are compiled to 3 SclRam rules where the first rule computes an auxiliary relation `temp` and the last two rules correspond to the rules in the Scallop program. Note that the anti-join ($\triangleright$) operator is used to compute the path with no enemy within it.

Iteration i	1	2	3	4	5	6	7
$t^{(i)}_{(1,C)\text{-}(3,C)}$ in $F^{(i)}_T$	–		=		=		=
$t^{(i)}_{(1,C)\text{-}(3,C)}$ in $F^{(i)}_{\texttt{mmp}}$	–	0.1	0.1	0.2	0.2	0.9	0.9
$t^{(i)}_{(1,C)\text{-}(3,C)}$ saturated?	–	F	T	F	T	F	T
$F^{(i)}_{\texttt{mmp}}$ saturated?	F	F	F	F	F	F	T

(c) An illustration of the tags that are evolving over iterations. In the figure, = means unchanged tag, while T and F represent true and false, respectively. We use a symbol like  to represent a conjunction of negated tags of `enemy` along the illustrated path, e.g. $= (\ominus t_{(2,C)}) \otimes (\ominus t_{(3,C)})$.

Figure 3.8: A demonstration of the fixed-point iteration to check whether actor at $(1, C)$ can reach $(3, C)$ 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 $\mathbf{1}$. Let $t_{(1,C)\text{-}(3,C)}$ be the tag associated with `path(1,C,3,C)`. 2nd iter is the first time $t_{(1,C)\text{-}(3,C)}$ is derived, but the path is blocked by an enemy. On 6th iter, the best path is derived in the tag. After that, under the `mmp` provenance, both the tag $t_{(1,C)\text{-}(3,C)}$ and the database $F_{\texttt{mmp}}$ are saturated, causing the iteration to stop. Compared to untagged semantics in Datalog which will stop after 4 iterations, SclRam with `mmp` saturates slower but allowing to explore better reasoning chains.

Figure 3.9: Execution pipeline with external interface.

Rules and fixed-point iteration. Evaluating a rule $p \leftarrow e$ on database F_T concerns evaluating the expression e and merging the result with the existing facts under predicate p in F_T. The result of evaluating e may contain duplicate tuples tagged by distinct tags, owing to expressions such as union, projection, or aggregation. Thus, we perform *normalization* by joining ($\oplus$) the distinct tags corresponding to the same tuple. From here, there are three cases to merge the newly derived tuples ($\langle [\![e]\!](F_T) \rangle$) with the previously derived tuples ($[\![p]\!](F_T)$). If a fact is present only in the old or the new, we simply propagate the fact to the output. When a tuple u appears in both the old and the new, we propagate the disjunction of the old and new tags ($t^{\texttt{old}} \oplus t^{\texttt{new}}$). Combining all cases, we obtain a set of newly tagged facts under predicate p.

Recursion in SclRam is performed similarly to least fixed point iteration in Datalog Abiteboul et al. (1995). The iteration happens on a per-stratum basis to enforce stratified negation and aggregation. Evaluating a single step of stratum s means evaluating all the rules in s and returning the updated database. Note that we define a specialized least fixed point operator $\mathbf{lfp}^{\circ}$, which stops the iteration once the whole database is *saturated*. Figure 3.8 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*

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

Figure 3.10: Definitions related to boolean formulas in disjunctive normal form.

tag space O, a *tagging function* $\tau : I \to T$, and a *recover function* $\rho : T \to O$. For instance, all probabilistic provenances share the same input and output tag spaces $I = O = [0, 1]$ for a unified interface, while the internal tag spaces T could be different. We call the 4-tuple (I, O, τ, ρ) the *external interface* for a provenance T. The whole execution pipeline is then illustrated in Figure 3.9.

In the context of a Scallop application, an EDB is provided in the form $F_{\texttt{option<}I\texttt{>}}$. During the *tagging phase*, τ is applied to each input tag to obtain F_T, following which the SclRam program operates on F_T. For convenience, not all input facts need to be tagged—untagged input facts are assigned the tag $\mathbf{1}$ in F_T. In the *recovery phase*, ρ is applied to obtain F_O, the IDB that the whole pipeline returns. Scallop allows the user to specify a set of *output relations*, and ρ is only applied to tags under such relations to avoid redundant computations.

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

3.5 Exact Probabilistic Reasoning with Provenance

We say that a provenance T is *probabilistic* if its input space I and output space O are real values in the range $[0, 1]$. As such, the `max-min-prob` provenance shown in Example. 1 is a probabilistic provenance. However, while useful in practice, `max-min-prob` only computes an approximation of the real probabilities. In this section, we start by introducing a more robust provenance that derives exact probabilities. This is also the probabilistic reasoning method used in the ProbLog2 (Dries et al., 2015) system.

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 3.10. Suppose there are n facts in the EDB with independent and identically distributed (i.i.d.) probabilities; we create n boolean variables each labeled $v_1, \ldots, v_n$. Then, a literal in the boolean formula is either a positive or a negated ($\neg$) boolean variable. A set of distinct literals connected by *and* ($\wedge$) form a conjunctive clause, while a set of clauses connected by *or* ($\vee$) form a disjunctive normal form formula ϕ. We note that there are two special DNF formulas, namely *true* ($\top$) and *false* ($\bot$). $\top$ is a singleton DNF formula with one empty conjunctive clause, where $\bot$ is an empty DNF formula. As such, the formal definition of `proofs-prob` is defined as follows:

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

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

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

where the tagging process allocates a new boolean variable v_i *for each input probability* p_i*, WMC is the function for Weighted Model Counting, and* $\Gamma(v_i) = p_i$ *is the mapping from allocated boolean variables (*v_i*) to their corresponding input probabilities (*p_i*).*

Following Figure 3.6, we show one concrete example of `proofs-prob`'s derivation process in Figure 3.11. 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 $\tau_{\mathtt{pp}}$ transforms the input probabilities into internal tags. After the derivation of boolean formulas, we apply recovery function $\rho_{\mathtt{pp}}$ to compute the probabilities of the resulting facts. At the end, we

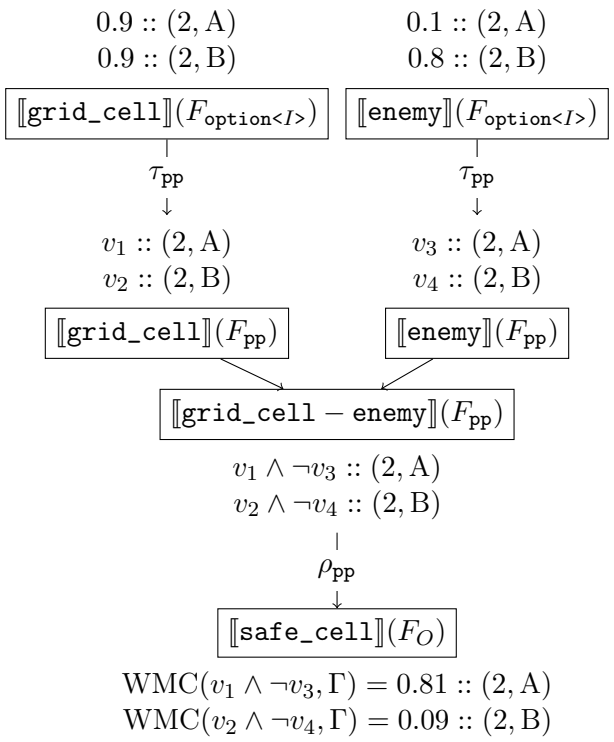


Figure 3.11: The SclRam evaluation result with `proofs-prob` on the rule shown in Figure 3.6. As shown in the first row, we assume that facts in `grid_cell` and `enemy` are provided as base facts with given probabilities. Therefore, each of the 4 shown facts on the second row is assigned a unique boolean variable $v_1, \ldots, v_4$. The third row has the two IDB facts tagged by boolean formulas such as $v_1 \wedge \neg v_3$. In the end, the output facts are tagged by probabilities derived by WMC-based recovery procedure.

note that the result computed from the weighted model counting (WMC) process is the probability of the corresponding tagged fact (Chavira and Darwiche, 2003; Van den Broeck et al., 2013).

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 Approximated Provenance for Scalable Reasoning

3.6.1 Top-k Proofs

The probabilistic nature of our problem setting opens up ample room for approximation, which may drastically improve scalability. A key observation is that, when the inference system is used in a machine 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 (Huang et al., 2021; Gutmann et al., 2008).

First, we introduce a different way of formalizing the proofs and top-k proofs. We treat each DNF boolean formula ϕ as a set of proofs, where each proof is a set of literals $\mathbf{N}$. 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 3.12. 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 $\vee^{(k)}$ and $\wedge^{(k)}$, where k is a tunable parameter controlling the level of approximation.

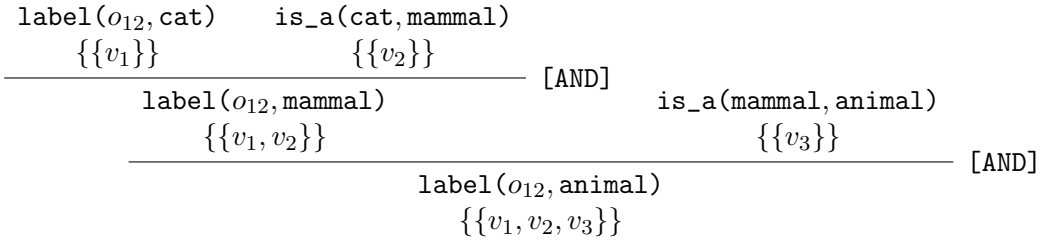
$$\phi_1 \vee^{(k)}_{\texttt{tkp}} \phi_2 = \text{top}_k(\phi_1 \cup \phi_2) \tag{3.5}$$

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

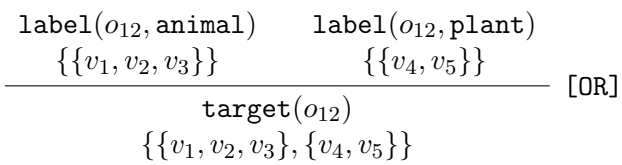
The goal is to pick out the "top-k" proofs within the result, where proofs are ranked by their respective

```
// The object o12 is identified as a cat with the highest likelihood
rel label = {0.9::(o12, "cat"); 0.01::(o12, "flower"); ...}

// Common sense ontology graph which indicates
// - a cat is a mammal
// - a mammal is an animal
rel is_a = {("cat", "mammal"), ("mammal", "animal")}

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

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

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



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



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

Figure 3.12: Derivation of set-of-proofs under different operations.

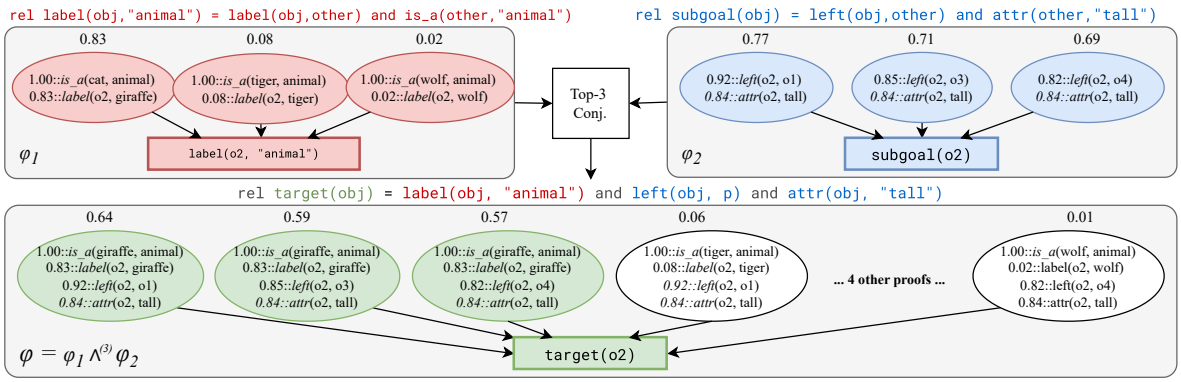


Figure 3.13: Illustration of top-k conjunction using $k = 3$. Each ellipse represents a proof of the fact shown in the box. Given the top 3 proofs for each of "`label`(o_2, `"animal"`)" and "subgoal(o_2)", we wish to derive the top 3 proofs for their conjunction, "target(o_2)". The join yields 9 possible proofs. After computing the likelihood for each of the 9 proofs, we keep the top 3 most likely ones (green ellipses) and discard the rest (white ellipses).

probability. Specifically, the probability of each proof, $\Pr(\eta)$ is computed the following:

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

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

Intuitively, whenever $\vee$ or $\wedge$ is performed, we rank proofs by their likelihood and preserve only the top-k proofs. This allows us to discard the vast majority of proofs and thus make inference tractable. When merging two proofs during $\wedge^{(k)}$, a single proof might contain the conjunction of conflicting literals, e.g. v_i and $\neg v_i$, in which case we remove the whole proof. An example run-through of *top-3 conjunction* ($\wedge^{(3)}$) is depicted in Figure 3.13, where we perform a normal $\wedge$ operation followed by a top-3 filtering.

To take negation $\neg^{(k)}$ on DNF φ, we first negate all the literals to obtain a *conjunctive normal form* (CNF) equivalent to $\neg\varphi$. Then we perform cnf2dnf operation (conflict check included) to convert it

back to a DNF. The top-k operation is performed at the end, as the following:

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

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

Combining everything above, we now give the formal definition to the top-k-proofs provenance:

Definition 2. *The `top-k-proofs` (`tkp`) provenance is the 7-tuple*

$$(\Phi_{\texttt{tkp}}, \{\}, \{\emptyset\}, \vee^{(k)}_{\texttt{tkp}}, \wedge^{(k)}_{\texttt{tkp}}, \neg^{(k)}_{\texttt{tkp}}, =), \tag{3.10}$$

where $\Phi_{\texttt{tkp}} = \mathcal{P}(\mathcal{P}(\mathbf{N}))$ represents the space of disjunctive normal form (DNF) formulas encoded as a two-level nested set structure. Here, $\mathcal{P}$ denotes power-set operator, and $\mathbf{N}$ denotes the set of literals. The inner set $\mathcal{P}(\mathbf{N})$ captures all possible conjunctive proofs (i.e., conjunctions of literals), and the outer power set $\mathcal{P}(\cdot)$ wraps these to form sets of DNF formulas, each comprising multiple conjunctions. The operations $\vee^{(k)}_{\texttt{tkp}}$, $\wedge^{(k)}_{\texttt{tkp}}$, and $\neg^{(k)}_{\texttt{tkp}}$ are lifted versions of logical disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$), respectively, augmented with a top_k selection operator. Each operation ensures that only the k most-likely conjunctive proofs are retained. The external interface for `tkp` is the quadruple

$$([0,1], [0,1], \tau_{\texttt{tkp}}, \rho_{\texttt{tkp}}), \tag{3.11}$$

where

- *$\tau_{\texttt{tkp}}(p_i) = v_i$ is the tagging function that introduces a fresh Boolean variable v_i for each input probabilistic fact with weight $p_i \in [0,1]$, and*

- $\rho_{tkp}(\phi) = WMC(\phi, \Gamma)$ *is the evaluation function that performs Weighted Model Counting on the DNF formula* ϕ *under the weight mapping* Γ*, where* $\Gamma(v_i) = p_i$.

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

We now present some desirable properties of our top-k inference algorithm. Suppose we have obtained a DNF $\phi_{\texttt{pp}}$ with the exact probabilistic reasoning module, as well as a DNF $\phi_{\texttt{tkp}}$ with the top-k proofs reasoning module. And that their success probabilities are $\Pr(\phi_{\texttt{pp}})$ and $\Pr(\phi_{\texttt{tkp}})$. The approximation error bound is given by $|\Pr(\phi_{\texttt{pp}}) - \Pr(\phi_{\texttt{tkp}})| \leq \sum_{\eta \in \phi_{\texttt{pp}} \setminus \phi_{\texttt{tkp}}} \Pr(\eta)$, and we can tune k to control the trade-off between scalability and approximation precision. Note that the size of $\phi_{\texttt{tkp}}$ is bounded by a controllable constant k ($|\phi_{\texttt{tkp}}| = \mathcal{O}(k)$). Therefore, the complexity of disjunction and conjunction operations are reduced from exponential to linear with respect to the number of boolean variables. It also drastically improves the scalability of weighted model counting during the recovery phase.

3.6.2 Top-Bottom-k Clauses

One scalability limitation of the Top-k Proofs provenance arises in the negation operator $\neg^{(k)}$, which requires a conversion from conjunctive normal form (CNF) to disjunctive normal form (DNF), also known as cnf2dnf. This operation is known to have worst-case exponential complexity, due to the potential combinatorial blowup in clause enumeration.

A practical optimization stems from observing that, by De Morgan's laws, the negation of a DNF formula naturally has the structure of an equivalent CNF, where each conjunctive clause becomes a

disjunctive clause of negated literals. For example:

$$\neg(v_1 \wedge v_2) \vee (v_2 \wedge v_3) = (\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3),$$

where the outer disjunction ($\vee$) and inner conjunctions ($\wedge$) are swapped due to De Morgan's laws, and negations are pushed down to the literals. This transformation enables us to avoid full cnf2dnf expansion by treating the negated result as a structured CNF.

Based on this observation, we design a provenance named *Top-Bottom-k Clauses*, which allows dual CNF and DNF representation of a set of clauses. In this design, clauses under a CNF are interpreted as disjunctive clauses, while clauses under a DNF are interpreted as conjunctive clauses. Similar to the `Top-`k`-Proofs` provenance, the DNF form retains the top-k conjunctive clauses (i.e., the most likely proofs). In contrast, the CNF form retains the bottom-k disjunctive clauses, representing those with the lowest associated probabilities.

The motivation behind the bottom-k heuristic is rooted in probabilistic reasoning: the probability of a conjunctive formula being true is upper-bounded by the probability of its least likely literal. Analogously, in a CNF, the falsifiability (or impact on negation) is determined by the weakest disjunctive clause, i.e., the one that is easiest to satisfy. Thus, selecting the bottom-k disjunctive clauses captures the weakest links in the negation space, which are most critical in influencing the resulting probabilistic bound.

This dual representation enables efficient approximate reasoning under negation by preserving only the most informative clauses in each polarity—top-k for affirmations, and bottom-k for denials—while avoiding exponential blowup from full CNF-to-DNF expansion during negation. Formally, we have the following definition for Top-Bottom-k Clauses provenance:

Definition 3. *The `top-bottom-`k`-clauses (tbkc)` provenance is the 7-tuple*

$$(\Phi_{\texttt{tbkc}}, \mathit{DNF}(\{\}), \mathit{DNF}(\{\emptyset\}), \vee^{(k)}_{\texttt{tbkc}}, \wedge^{(k)}_{\texttt{tbkc}}, \neg^{(k)}_{\texttt{tbkc}}, =), \tag{3.12}$$

where

$$\Phi_{\texttt{tbkc}} = \texttt{DNF}(\mathcal{P}(\mathcal{P}(\mathbf{N}))) \mid \texttt{CNF}(\mathcal{P}(\mathcal{P}(\mathbf{N}))) \tag{3.13}$$

represents the dual space of DNF and CNF formulas encoded as a two-level nested set structure. Here, $\mathcal{P}$ denotes power-set operator, and $\mathbf{N}$ denotes the set of literals. The inner set $\mathcal{P}(\mathbf{N})$ captures all possible clauses, and the outer power set $\mathcal{P}(\cdot)$ wraps these to form sets of DNF or CNF formulas, each comprising multiple clauses. The operations $\vee^{(k)}_{\texttt{tbkc}}$, $\wedge^{(k)}_{\texttt{tbkc}}$, and $\neg^{(k)}_{\texttt{tbkc}}$ are lifted versions of logical disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$), respectively, for the provenance. Each operation ensures that only the k most-likely conjunctive proofs are retained. Specifically, the $\vee^{(k)}_{\texttt{tbkc}}$ operation is defined as follows:

$$\texttt{CNF}(\phi_1) \vee^{(k)}_{\texttt{tbkc}} \texttt{CNF}(\phi_2) = \texttt{CNF}(bottom_k(\{\eta_1 \cup \eta_2 \mid \eta_1 \in \phi_1, \eta_2 \in \phi_2\})), \tag{3.14}$$

$$\texttt{DNF}(\phi_1) \vee^{(k)}_{\texttt{tbkc}} \texttt{DNF}(\phi_2) = \texttt{DNF}(top_k(\phi_1 \cup \phi_2)), \tag{3.15}$$

$$\texttt{CNF}(\phi_1) \vee^{(k)}_{\texttt{tbkc}} \texttt{DNF}(\phi_2) = \texttt{DNF}(top_k(cnf2dnf(\phi_1) \cup \phi_2)); \tag{3.16}$$

the $\wedge^{(k)}_{\texttt{tbkc}}$ operation is defined as follows:

$$\texttt{CNF}(\phi_1) \wedge^{(k)}_{\texttt{tbkc}} \texttt{CNF}(\phi_2) = \texttt{CNF}(bottom_k(\phi_1 \cup \phi_2)), \tag{3.17}$$

$$\texttt{DNF}(\phi_1) \wedge^{(k)}_{\texttt{tbkc}} \texttt{DNF}(\phi_2) = \texttt{DNF}(top_k(\{\eta_1 \cup \eta_2 \mid \eta_1 \in \phi_1, \eta_2 \in \phi_2\})), \tag{3.18}$$

$$\texttt{CNF}(\phi_1) \wedge^{(k)}_{\texttt{tbkc}} \texttt{DNF}(\phi_2) = \texttt{DNF}(top_k(\{\eta_1 \cup \eta_2 \mid \eta_1 \in cnf2dnf(\phi_1), \eta_2 \in \phi_2\})); \tag{3.19}$$

and the $\neg^{(k)}_{\texttt{tbkc}}$ operation is defined as:

$$\neg^{(k)}_{\texttt{tbkc}}\texttt{CNF}(\phi) = \texttt{DNF}(\{\{\neg\nu \mid \nu \in \eta\} \mid \eta \in \phi\}), \tag{3.20}$$

$$\neg^{(k)}_{\texttt{tbkc}}\texttt{DNF}(\phi) = \texttt{CNF}(\{\{\neg\nu \mid \nu \in \eta\} \mid \eta \in \phi\}). \tag{3.21}$$

The external interface for **`tkp`** *is the quadruple*

$$([0,1],[0,1],\tau_{\texttt{tbkc}},\rho_{\texttt{tbkc}}), \tag{3.22}$$

where

- $\tau_{\texttt{tbkc}}(p_i) = v_i$ *is the tagging function that introduces a fresh Boolean variable* v_i *for each input probabilistic fact with weight* $p_i \in [0,1]$*, and*
- $\rho_{\texttt{tbkc}}(\phi) = WMC(\phi,\Gamma)$ *is the evaluation function that performs Weighted Model Counting on the DNF or CNF formula* ϕ *under the weight mapping* Γ*, where* $\Gamma(v_i) = p_i$*.*

Here, the top-k operation is identical with the one in top-k proofs, while the bottom-k operation needs to estimate the probability of a disjunctive clause $\eta_{\texttt{disj}}$. It is done using the following algorithm:

$$\Pr(\eta_{\texttt{disj}}) = 1 - \prod_{\nu_i \in \eta_{\texttt{disj}}} (1 - \nu_i) \tag{3.23}$$

While the top-bottom-k clauses simplifies the negation operation, it does not eliminate the fundamental exponential complexity inherent in the problem. Specifically, the complexity originally incurred by $\neg^{(k)}_{\texttt{tkp}}$ is not avoided but rather shifted to the disjunction and conjunction operators ($\vee^{(k)}_{\texttt{tbkc}}$ and $\wedge^{(k)}_{\texttt{tbkc}}$) particularly when these operators must merge clauses where one operand is in CNF and the other in DNF. As a result, the computational advantages of the top-bottom-k clauses provenance depend heavily on the structure of the Scallop program. It offers meaningful scalability benefits in scenarios where a negation is applied to a fcat derived from complex logic, but is not immediately followed by further conjunctions, which would otherwise trigger expensive mixed-form merges.

3.6.3 Optimal-k Proofs

While the `top-`k`-proofs` (`tkp`) provenance focuses on retaining the k most likely conjunctive proofs for a given query, this approach may lead to over-representation of a narrow region of the probability distribution. In particular, top-k operation may select highly similar proofs that differ only slightly

Algorithm 1: Brute-force Algorithm for Optimal-k Proofs.

Data: A set of n conjunctive proofs $\{\eta_1, \ldots, \eta_n\}$, each with associated probability $\Pr(\eta_i)$, and integer k

Result: A subset $S \subseteq \{\eta_1, \ldots, \eta_n\}$ with $|S| = \min(k, n)$ maximizing $\Pr\left(\bigvee_{\eta \in S} \eta\right)$

$S^* \leftarrow \emptyset$; $p^* \leftarrow 0$; // Best subset and its disjunction probability

foreach *subset* $S \subseteq \{\eta_1, \ldots, \eta_n\}$ *with* $|S| = \min(k, n)$ **do**

  $p \leftarrow \Pr\left(\bigvee_{\eta \in S} \eta\right)$; // Evaluate disjunction probability

  if $p > p^*$ **then**

    $S^* \leftarrow S$; $p^* \leftarrow p$

return S^*

in structure or semantics, resulting in reduced diversity among the retained explanations. To address this limitation, we introduce the `optimal-k-proofs` (`okp`) provenance, adapting algorithms first presented in Renkens et al. (2012).

Formally speaking, assume that $\phi_{\texttt{pp}}$ is the DNF derived by `prob-proofs` provenance, and having n conjunctive clauses $\eta_1, \ldots, \eta_n$ within it. The optimal-k proofs $\phi_{\texttt{okp}}$ is the DNF which contains at most k conjunctive clauses $\eta_{\texttt{okp}_1}, \ldots, \eta_{\texttt{okp}_{\min(k,n)}}$ where $\texttt{okp}_i \in \{1 \ldots n\}$ are distinct and

$$\phi_{\texttt{okp}} = \text{argmax}_{\texttt{okp}_1, \ldots, \texttt{okp}_{\min(k,n)}} \Pr\left(\bigvee_{i \in 1 \ldots \min(k,n)} \eta_{\texttt{okp}_i}\right). \tag{3.24}$$

Intuitively speaking, $\phi_{\texttt{okp}}$ contains the k conjunctive clauses from $\phi_{\texttt{pp}}$ which maximizes the overall disjunction probability, therefore being optimal.

The brute-force algorithm for optimal-k proofs (Algorithm 1) exhaustively searches all subsets of up to k conjunctive clauses to find the one that maximizes disjunction probability, guaranteeing optimality but at exponential computational cost. In contrast, the greedy algorithm (Algorithm 2) builds the subset incrementally by selecting, at each step, the clause that yields the highest marginal gain in disjunction probability. While greedy is far more efficient and often performs well in practice, it does not guarantee a globally optimal solution. The key trade-off is between completeness and scalability: brute-force ensures accuracy but is infeasible for large inputs, whereas greedy offers a practical approximation with significantly lower computational overhead.

Algorithm 2: Greedy Algorithm for Optimal-k Proofs.

Data: A set of conjunctive clauses $\{\eta_1, \ldots, \eta_n\}$ and integer k

Result: A subset $S \subseteq \{\eta_1, \ldots, \eta_n\}$ with $|S| = k$ approximating maximal $\Pr\left(\bigvee_{\eta \in S} \eta\right)$

$S \leftarrow \emptyset$; // Initialize selected clause set

for $i \leftarrow 1$ **to** k **do**

  $\eta^* \leftarrow \text{argmax}_{\eta \in \{\eta_1, \ldots, \eta_n\} \setminus S} \Pr\left(\bigvee_{\eta' \in S \cup \{\eta\}} \eta'\right)$;

  $S \leftarrow S \cup \{\eta^*\}$

return S

The simpliest implementations of optimal-k proofs introduces an additional optimization layer that operates over the set of proofs generated during inference. Specifically, the system first computes the full disjunctive normal form (DNF) of a query using standard provenance techniques such as `prob-proofs`, and then applies a selection algorithm—either brute-force (Algorithm 1) or greedy (Algorithm 2)—to extract the subset of k proofs that maximizes the overall disjunction probability. These two strategies yield two provenance variants: `optimal-`k`-proofs-bf`, which guarantees optimality by exhaustively enumerating all k-sized subsets, and `optimal-`k`-proofs-greedy`, which trades optimality for efficiency by incrementally selecting clauses with maximal marginal gain. This separation of inference and recovery makes the optimization modular and interpretable.

However, computing all possible proofs before selecting the optimal k is often impractical, especially in complex queries where the number of conjunctive proofs grows exponentially. Materializing the entire DNF before applying greedy selection can itself become intractable. To address this, Scallop also supports a third, more scalable strategy that integrates selection during inference. Instead of waiting until the end, it maintains a running set of the optimal k proofs throughout evaluation. As logical operations such as conjunction, disjunction, and negation are applied, the system approximates new clauses while pruning to retain only the optimal k candidates at each step. Although this compromises the optimality of the final result, it significantly improves efficiency and often yields broader coverage than naively applying top-k or beam search on local substructures.

Definition 4. *The* `optimal-k-proofs (okp)` *provenance is the 7-tuple*

$$(\Phi_{okp}, \{\}, \{\emptyset\}, \vee^{(k)}_{okp}, \wedge^{(k)}_{okp}, \neg^{(k)}_{okp}, =), \tag{3.25}$$

where $\Phi_{okp} = \mathcal{P}(\mathcal{P}(\mathbf{N}))$ *represents the space of disjunctive normal form (DNF) formulas encoded as a two-level nested set structure. The operations* $\vee^{(k)}_{okp}$, $\wedge^{(k)}_{okp}$, *and* $\neg^{(k)}_{okp}$ *are lifted versions of logical disjunction (*$\vee$*), conjunction (*$\wedge$*), and negation (*$\neg$*), respectively, augmented with a GreedyOptimal*$_k$ *selection operator. Each operation ensures that only the k conjunctive proofs which can approximately cover the most probability are retained. The external interface for* `okp` *is the quadruple*

$$([0,1], [0,1], \tau_{okp}, \rho_{okp}), \tag{3.26}$$

where

- $\tau_{okp}(p_i) = v_i$ *is the tagging function that introduces a fresh Boolean variable* v_i *for each input probabilistic fact with weight* $p_i \in [0,1]$*, and*
- $\rho_{okp}(\phi) = WMC(\phi, \Gamma)$ *is the evaluation function that performs Weighted Model Counting on the DNF formula* ϕ *under the weight mapping* Γ*, where* $\Gamma(v_i) = p_i$*.*

In summary, there is a fundamental trade-off between using `optimal-`k`-proofs` and `top-`k`-proofs`. The `top-`k strategy selects the k most probable conjunctive clauses independently and is computationally efficient, but may result in redundant or overlapping proofs. In contrast, `optimal-`k`-proofs` explicitly chooses the subset of k clauses that maximizes the overall disjunction probability, offering better coverage at the cost of increased computational overhead due to evaluating joint probability interactions among clauses. This improved coverage is especially beneficial in learning scenarios, where probabilities are produced by neural components and gradients flow back through the selected proofs. A broader disjunction allows more proof paths to contribute, enabling more non-zero gradients and better credit assignment across the model. Notably, when $k = 1$, both methods are equivalent, since the single most probable clause trivially maximizes the disjunction.

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

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

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

Figure 3.14: Operations on dual-number $\mathbb{D} \triangleq [0, 1] \times \mathbb{R}^n$, where n is the number of input probabilities.

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 3.14). Each input tag $r_i \in [0, 1]$ is a probability, and each output tag $\hat{y}_j = (y_j, \nabla y_j)$ encapsulates the output probability y_j and its derivative w.r.t. inputs, ∇y_j. From here, we can obtain our expected output $\vec{y}$ and $\nabla \vec{y}$ by stacking together y_j-s and ∇y_j-s respectively.

Scallop provides 8 configurable built-in 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 3.15. We use r_i to denote the i-th element of $\vec{r}$, where i is called a *variable* (ID). Vector

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

Figure 3.15: Definitions of three differentiable provenances.

$\vec{\mathbf{e}}_i \in \mathbb{R}^n$ is the standard basis vector where all entries are 0 except the i-th entry.

3.7.1 `diff-max-min-prob` (`dmmp`)

This provenance is the differentiable version of `max-min-prob` (Example 1). 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 `diff-max-min-prob`. 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 `diff-add-mult-prob` 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 `diff-add-mult-prob` is $\mathcal{O}(n)$, which is slower than `diff-max-min-prob` but is still very efficient in practice.

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

This provenance extends the *top-k proofs* introduced semiring proposed in Huang et al. (2021); Gutmann et al. (2008) to additionally support negation and aggregation. As introduced in Section 3.6

and also shown in Figure 3.15, the tags of `diff-top-k-proofs` are boolean formulas $\varphi \in \Phi$ in *disjunctive normal form* (DNF). The difference between `diff-top-k-proofs` 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 `diff-top-k-proofs` are defined as:

$$\tau_{\texttt{dtkp}}(\Pr(v_i)) = v_i \tag{3.27}$$

$$\rho_{\texttt{dtkp}}(\varphi) = \text{WMC}(\varphi, \Gamma) \tag{3.28}$$

$$\Gamma(i) = (\Pr(v_i), \vec{\mathbf{e}}_i) \tag{3.29}$$

where WMC is now a differentiable weighted-model counting procedure adopted from Manhaeve et al. (2021). Other than the boolean formula φ, WMC also takes in the weights of each probabilistic variable i. Instead of simple probabilities, the weights are now dual numbers like $(\Pr(v_i), \vec{\mathbf{e}}_i)$. During differentiable WMC, the dual-number addition and multiply rules (Figure 3.14) 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 earlier remove such facts. We introduce a new function to the provenance interface, discard : $T \to$ Bool. If discard returns true ($\top$) when called on the tag of a fact, then the fact will be removed from subsequent computation. The default implementation of this function is $\text{discard}(t) = t \ominus \mathbf{0}$.

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

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

3.8.2 Mutual Exclusivity of Facts

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

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

Consider the example shown in Listing 3.1. The two sets of mutually exclusive facts are transformed into two distinct mutual exclusion IDs, which we label 0 and 1. The fact `color(OBJ_A, "red")` is technically tagged by $(0.9, 0)$. The first element 0.9 is treated as a normal probability, while the mapping from this fact ID to the mutual exclusion ID is stored for future reference. When executing the rule (line 4), we derive a temporary proof containing facts `color(OBJ_A, "red")` and

Algorithm 3: Counting over max-min-prob tagged tuples.

Data: $U_{\texttt{mmp}} = \{t_1 :: u_1, t_2 :: u_2, \ldots, t_n :: u_n\}$: $\mathcal{U}_{\texttt{mmp}}$, set of tagged-tuples to count
Result: $U'_{\texttt{mmp}}$: $\mathcal{U}_{\texttt{mmp}}$

```
/* sort all positive tuples according to their tags from small to large.
   O(nlog(n))                                                          */
```

1 $\mathbf{t}^{\text{pos}} = sorted([t_i \mid i = 1 \ldots n])$;
2 $\mathbf{t}^{\text{neg}} = [1 - t^{\text{pos}}_{n-i+1} \mid i = 1 \ldots n]$;

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

3 $U'_{\texttt{mmp}} = \{t^{\text{neg}}_n :: 0, t^{\text{pos}}_1 :: n\}$;
4 **for** $i = 1 \ldots (n-1)$ **do**
5 $\quad$ Add $\min(t^{\text{pos}}_{i+1}, t^{\text{neg}}_i) :: (n-i)$ to $U'_{\texttt{mmp}}$;
6 **return** $U'_{\texttt{mmp}}$

`color(OBJ_A, "green")`. However, when looking up the mutual exclusion information, we find that the two facts cannot co-exist in the same proof. `prob-proofs` provenance will reject such a proof, rendering the result tag to be **0**. Therefore, combined with the early removal feature, the `should_not_exist` relation is computed to be empty, as desired.

3.8.3 Specializing for Provenances

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 3. Note that we only showed the algorithm for `mmp` for simplicity, but it easily extends to `dmmp`. Scallop implements many other optimizations with varying degrees of approximations so that operations that are in principle expensive become tractable when applied to real-life scenarios.

3.8.4 Sampling Operations

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

3.8.5 Provenance Selection

Given the rich library of Scallop provenances and operations, a natural question that arises is how to select a differentiable provenance for a given Scallop application. Based on our empirical evaluation, `dtkp` is often the best performing one, and setting $k = 3$ is usually a good choice for both runtime efficiency and learning performance. This suggests that a user should start with `dtkp` before searching other provenances. In general, provenance selection in Scallop is analogous to hyperparameter tuning in machine learning.

CHAPTER 4

PROGRAMMING WITH FOUNDATION MODELS

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 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 4.1a shows a Scallop program which invokes GPT to extract the height of mountains whose names are specified in a structured table. Likewise, the program in Figure 4.1b uses the image-text alignment model CLIP to classify images into discrete labels such as `cat` and `dog`. Figure 4.1c shows relational input-output

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

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

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

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

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

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

examples for the two programs. Notice that the CLIP model also outputs probabilities that allow for probabilistic reasoning.

In this chapter, we first introduce the extensible plugin library (Section 4.1). We then dive into the different relational constructs that we have designed for large language models (Section 4.2), embedding models (Section 4.3), and vision language models (Section 4.4). We conclude the chapter by two case studies, one on face tagging (Section 4.5) and one on visual question answering (Section 4.6)

4.1 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

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

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

  return run_clip
```

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

program.

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

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

4.2 Large Language Models

Text completion In Scallop, large language models (LLMs) like GPT (OpenAI, 2023b) and LLaMA (Touvron et al., 2023) can be used as basic foreign predicates for text completion (Listing 4.2). In this case, `gpt` is an arity-2 FP that takes in `request`, a `String` as the prompt, and produces `response`, a `String` as the response. As a result, we would obtain the fact `ans("8468000")`. We note that the foreign predicate `gpt` uses the model `gpt-3.5-turbo` by default.

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

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

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

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

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

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

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

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

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

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

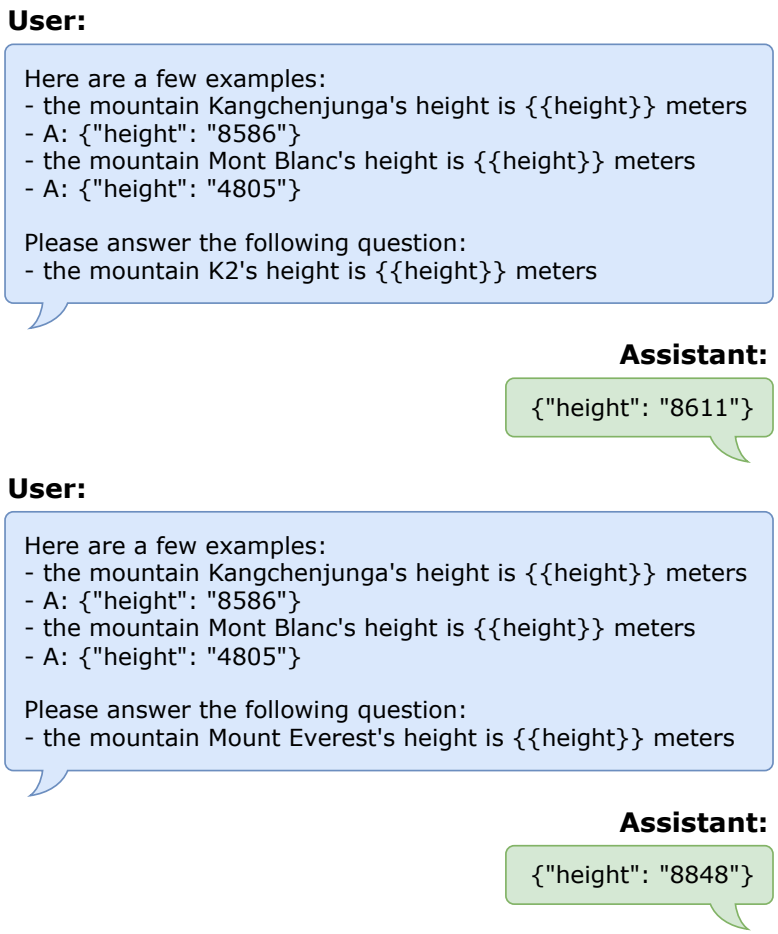


Figure 4.2: Conversation history between User (messages generated by `gpt` FA) and GPT-4 (`gpt-4-0613`) Assistant via OpenAI API after executing the program in Listing 4.4.

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

This attribute differs from the text completion attribute `gpt` in that it can extract an arbitrary number of facts for multiple relations. To motivate the need for such an attribute, we consider the *date understanding* task from the BIG-bench suite (Srivastava et al., 2023). In this task, the model

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

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

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

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

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

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

Listing 4.7: FA-annotated rules for date understanding.

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

```
1 rel question = { "[Context] What is the date...?" }
2
3 rel mentioned_date(label, date) =
4   question(q) and extract_mentioned_date(q, label, date)
5 rel goal(label) =
6   question(q) and extract_goal(q, label)
7 rel relationship(l1, l2, diff) =
8   question(q) and extract_relationship(q, l1, l2, diff)
```

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

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

Referring to Listing 4.7, each question provided in `prompts` (lines 2–6) corresponds to a relation of a given type signature that GPT-4 should extract from the bound argument `question` in Listing 4.8. The shots provided in `examples` (lines 7–17) are formatted as messages that are prepended to the conversation history given to GPT-4, as seen in Figure 4.3. Finally, the parameter `cot` (line 18) is a Boolean array where `cot[i]` toggles whether the ith relation should be extracted using zero-shot chain-of-thought (CoT) prompting 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 4.7 and Listing 4.8 is given by Figure 4.3. With a little thought, the reader will find that applying the rules in Listing 4.6 on the relations generated by GPT-4 in Figure 4.3 will yield the correct answer: 12/26/2017.

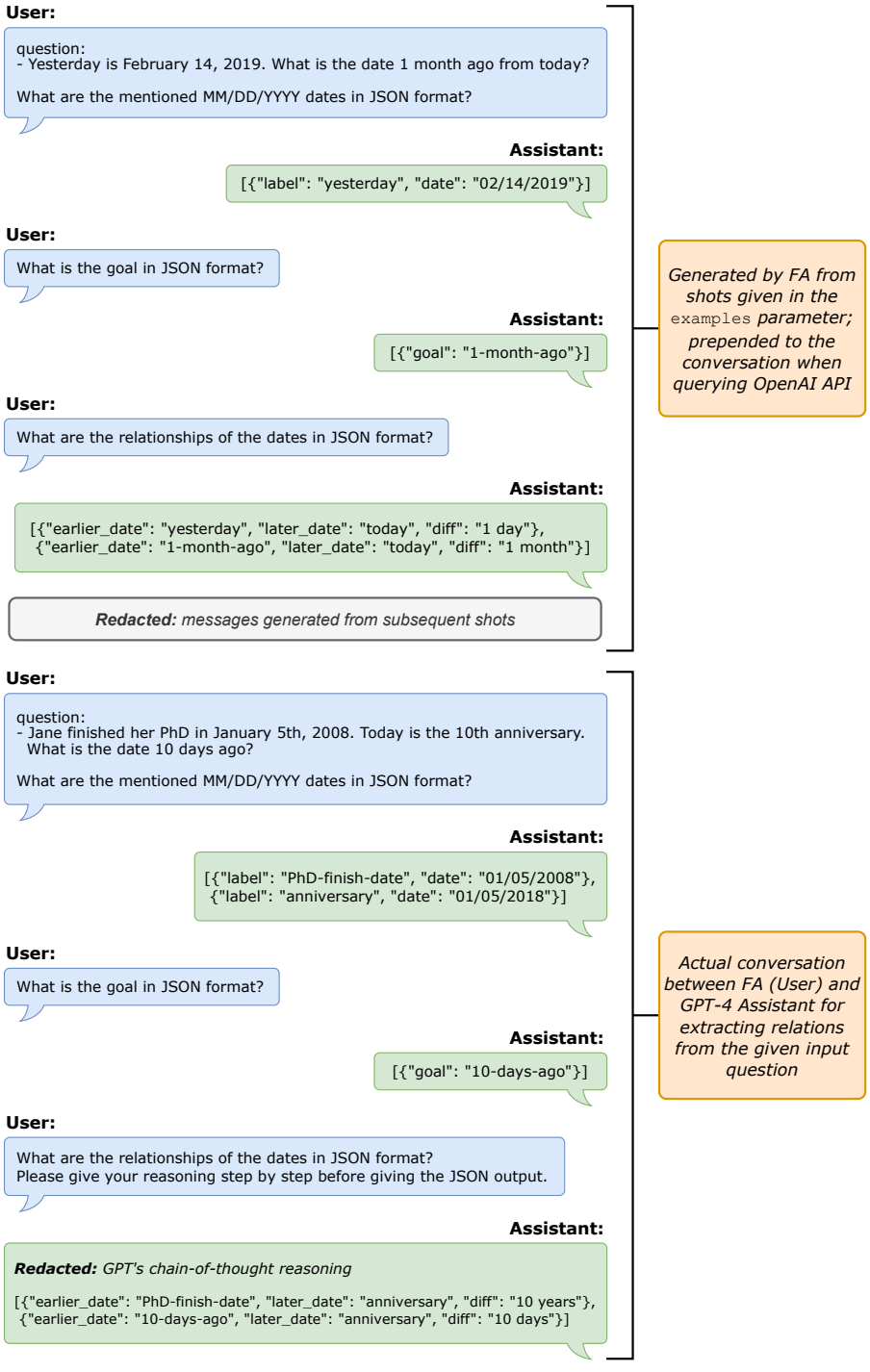


Figure 4.3: The GPT-4 conversation history after executing the program in Listing 4.7, with annotations and redactions in italics.

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

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.

4.3 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 4.9 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"`.

One application of these techniques is in information retrieval. For example, consider the task from HotpotQA Yang et al. (2018b). 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 4.10, we implement an adaptation of FE2H Li et al. (2022a). 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

```
1  @gpt_encoder
2  type $embed_text(String) -> Tensor
3
4  type question(q: String)
5
6  type context(id: i32, c: String)
7
8  rel relevant(id) = id := top<2>(
9    id: question(q) and
10   context(id, c) and
11   soft_eq<Tensor>($embed_text(q), $embed_text(c))
12 )
13 rel relevant_context($string_concat(c1, "\n", c2)) =
14   relevant(id1) and relevant(id2) and id1 < id2 and
15   context(id1, c1) and context(id2, c2)
16
17 @gpt(
18   prompt="Given {{ctxt}}\n{{q}}\n
19     Please think step-by-step {{ans}}"
20 )
21 type qa(bound q: String, bound ctxt: String, ans: String)
22
23 rel answer(a) =
24   question(q) and relevant_context(c) and qa(q, c, a)
```

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

`soft_eq`) to select the 2 documents most relevant to the question (lines 8–15), which are provided as context to GPT-4 to do QA (lines 17–24). By retrieving only 2 documents, the context we generate is inherently less distracting than the naive context of all 10 documents.

4.4 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 4.1b shows an example usage of the `@clip` attribute. We also note that dynamically-generated classification labels can be provided to CLIP via a bounded argument in the predicate.

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

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

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

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

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.

For instance, the OWL-ViT model can be used and configured as shown in Listing 4.11. Here, the `find_obj` predicate takes in an image, and finds image segments containing "human face" or "rocket". According to the names of the arguments, the model extracts 3 values per segment: an unique identifier (ID), the string label which could be either "human face" or "rocket", and the cropped image represented as a `Tensor`. If more information is desired, programmers can additionally specify other output arguments such as `bbox_x` to obtain information about the bounding-boxes. Note that each produced fact is tagged with a probability, representing the model's confidence on the segment.

Image generation models Image generation models such as Stable Diffusion (Rombach et al., 2022) and DALL-E (Ramesh et al., 2021) can also be viewed through a relational lens, where each model defines a relation between natural language inputs and generated image outputs.

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

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

One particularly compelling programmability trait of generative models is their ability to be invoked multiple times to produce distinct yet plausible images for a given textual description. In traditional imperative or empirical programming paradigms, such functionality typically requires explicitly defining the output as a list or collection of images. In contrast, Scallop-style relational programming abstracts this complexity: it defines relations, not data structures. As a result, there is no need to explicitly declare the output as a list—each plausible image generated corresponds to a tuple in the same relation. This significantly simplifies the program structure.

4.5 Case Study: Face Tagging by Foundation Model Relations

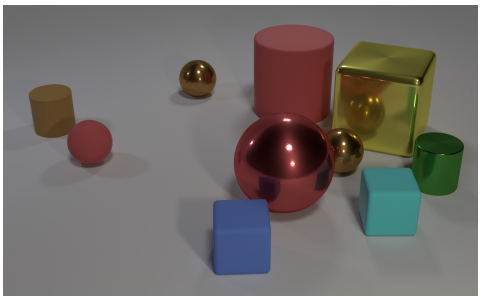
To demonstrate Scallop's usefulness in composing foundation models of various modalities, we introduce our face-tagging task based on that of VisProg (Gupta and Kembhavi, 2022). In our task, the model is given an image with a descriptive natural-language filename, and needs to output an edited image where all faces relevant to the description are boxed with their names. An example input-output pair is shown in Figure 4.4.

The code for face-tagging is provided in Listing 4.13. Our solution obtains a set of possible names from GPT-4 (lines 5–12) and candidate faces from the DSFD face detection model (lines 14–26). These are provided to CLIP for object classification (lines 28–31), after which probabilistic reasoning

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

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

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

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

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

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

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

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

Listing 4.13: Scallop program for face-tagging.

filters the most relevant face-name pairs (lines 33–37). Finally, the program calls image-editing foreign functions from the plugin library that use the face-name pairs to draw the captioned face



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

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

```
Answer: true
```

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

boxes (code omitted).

4.6 Case Study: Visual Question Answering on Scene Images

In this section, we describe the Scallop program for one of our applications, CLEVR (Johnson et al., 2016). In Figure 4.5, we illustrate a concrete example of the program we described executing on an image from the CLEVR dataset. We are given two inputs, namely the image (left) and the question (top-right), and we are supposed to produce an answer (bottom-right). In general, the images in CLEVR dataset may contain up to 10 primitive objects, each with a pre-defined set of shapes, colors, materials, and sizes. There is a range of questions whose answer maybe numbers (counting questions), true/false (existence or comparing or assertive questions), or properties (querying color).

We decompose our solution to this application into three sub-tasks: a) extracting a structured scene graph from the input image, b) extracting an executable query program from the input natural language (NL) question, and c) combining both to answer the question based on the scene graph. Here, a) and b) require the processing of unstructured data such as image and natural language question, and therefore may be *neural*. On the other hand, c) can be programed and fully symbolic. We may choose to have both neural networks for a) and b) to be trained by our end-to-end pipeline. But in light of the advancements of foundation models such as GPT-4 (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.

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

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

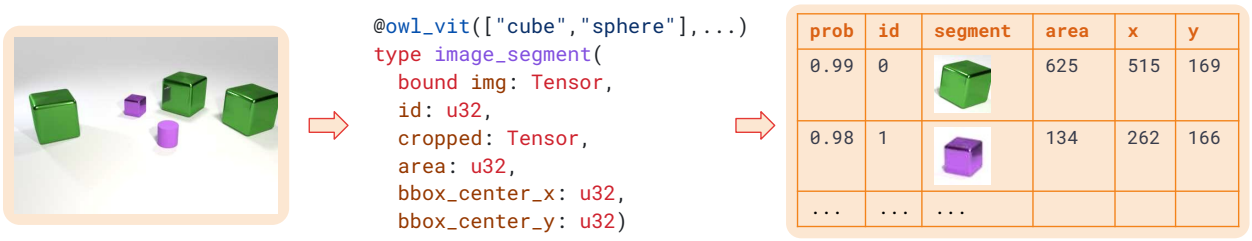


Figure 4.6: An illustration of the `segment_image` relation.

4.6.1 Image to structured scene graph

To convert image to structured scene graph, we use two off-the-shelf vision models, namely OWL-ViT (Minderer et al., 2022) and CLIP (Radford et al., 2021). We use OWL-ViT for obtaining object segments and CLIP models for classifying object properties. The goal is to construct scene graph which contains the following information: the shape, color, material, and size for each object, and the spatial relationships between each pair of objects.

Our object detection predicate is defined in Listing 4.14. We use the `@owl_vit` foreign attribute to decorate a predicate `vit_segment_image`. Here, the image has one bounded argument which is the input image, and it produces image segments represented by 5 tuples, containing segment id (`id`), segmented image (`cropped_image`), the area of segment (`area`), the center x coordinate (`bbox_center_x`), and the bottom y coordinate (`bbox_bottom_y`). Specifically, segmented images can be passed to downstream image classifiers, the area is used to classify whether the object is big or small, and the coordinates are used to determine spatial relationships between objects. We illustrate the produced table in Figure 4.6.

Note that the arguments we pass to `@owl_vit` contain expected labels of `cube`, `sphere`, and `cylinder`.

```
// the user provide the image with its directory
type img_dir(directory: String)

// load the image as a tensor
type image(img: Tensor)
rel image($load_image(d)) = img_dir(d)

// segment the image using our segment_image relation
rel obj_seg(id, seg, area, x, y) =
  image(img) and segment_image(img, id, seg, area, x, y)
rel obj(id) = obj_seg(id, _, _, _, _)
```

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

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 4.15.

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

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

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

Listing 4.16: Classifier relations using CLIP.

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

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

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

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

similarly, shown also in Listing 4.16.

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

4.6.2 Natural language question to programmatic query

We use the GPT-4 model (OpenAI, 2023b) for converting a natural language question into a

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

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

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

We then create the semantic parser for the DSL by configuring a relation to parse natural language question into a programmatic `Query`, shown in Listing 4.19. For this, we utilize the `@gpt_semantic_parse` foreign attribute provided in Scallop. Other than the `model` argument which is used to specify the OpenAI model to call, we pass the 3 main arguments to `gpt_semantic_parse`, namely `header`, `prompt`, and `examples`. `prompt` constitutes the system prompt, while the structures `examples` are expanded with the `prompt` into the few-shot examples. In Figure 4.7 we show one specific example of "conversation" with LLM to precisely parse the NL question into `Query`.

4.6.3 Putting it all together

The last part which brings everything together is the semantics of our `Query` DSL, shown in Listing 4.18. We can start by treating each variant of our DSL as a *function*. Assuming we have $O_{\text{all}} = \{o_1, o_2, \ldots, o_n\}$ representing the set of all objects in the scene. Then we have an arbitrary set of objects represented as $O \in \mathcal{P}(O_{\text{all}})$ where $\mathcal{P}$ is the powerset operation. Suppose $\mathcal{O} = \mathcal{P}(O_{\text{all}})$, we may give the following function types and functional semantics to a few set of variant in our DSL (Figure 4.8). For instance, `Scene` is a function that returns all the objects in the scene. `Count` takes in a set of objects and returns the cardinality of that set. Assuming we have relational predicates

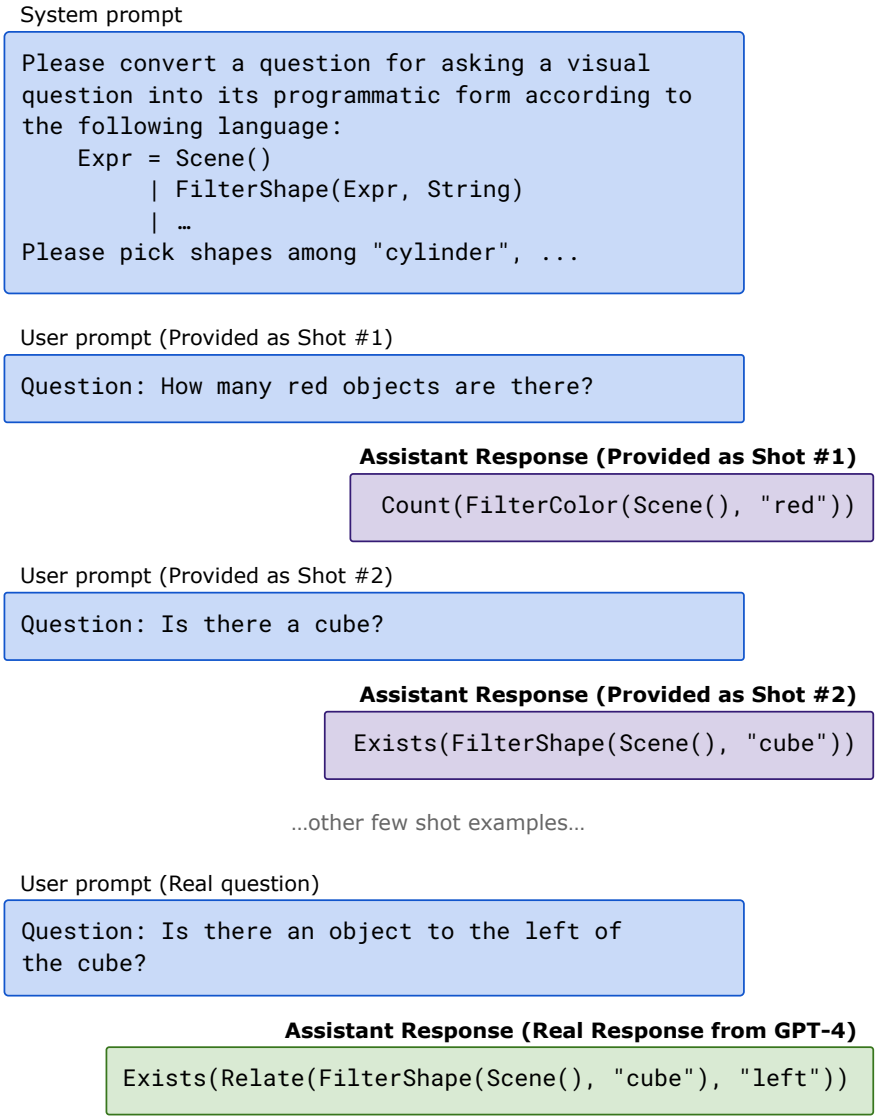


Figure 4.7: A "conversation" between Scallop and the LLM for semantically parsing the NL question into programmatic query in our domain specific language (Listing 4.18). We use few-shot prompting in order to generate accurate programmatic query. Everything except the last bubble (green) is generated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

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

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

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

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

Listing 4.19: A semantic parser relation `parse_query`.

such as `shape` and `color` pre-populated with facts in our scene graph, we may use them to define the functions such as `FilterShape` and `QueryColor`. Definitions of other predicates are omitted since they look similar to what we show.

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

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

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

```
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 (set of) objects
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 attributes
12 // the attributes are stringified for the result, such as
13 // "red", "cube", "left", "large", and etc.
14 type eval_str(q: Query, attr: String)
```

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

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

As shown by the rules, they are defined relatively concisely. Be reminded that while the rules look like their functional counterparts, they actually have their underlying probabilistic and differentiable

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

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

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

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

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

semantics. As such, the probabilities produced by image segmentation models and classifiers can propagate to produce a probabilistic distribution of answers.

CHAPTER 5

BENCHMARKS AND EVALUATIONS

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

5.1 Basic Scallop Benchmarks

First, we evaluate the Scallop language and framework on a basic benchmark suite comprising eight neurosymbolic applications. Here, our evaluation aims to answer the following research questions:

RQ1: How expressive is Scallop for solving diverse neurosymbolic tasks?

RQ2: How do Scallop's solutions compare to state-of-the-art baselines in terms of accuracy?

RQ3: Is the differentiable reasoning module of Scallop runtime-efficient?

RQ4: Is Scallop effective at improving generalizability, interpretability, and data-efficiency?

RQ5: What are the failure modes of Scallop solutions and how can we mitigate them?

In the following sections, we first introduce the benchmark tasks and the chosen baselines for each task (Section 5.1.1). Then, we answer **RQ1** to **RQ5** in Section 5.1.2 to Section 5.1.6 respectively. All Scallop related and runtime related experiments were conducted on a machine with two 20-core Intel Xeon CPUs, four GeForce RTX 2080 Ti GPUs, and 768 GB RAM.

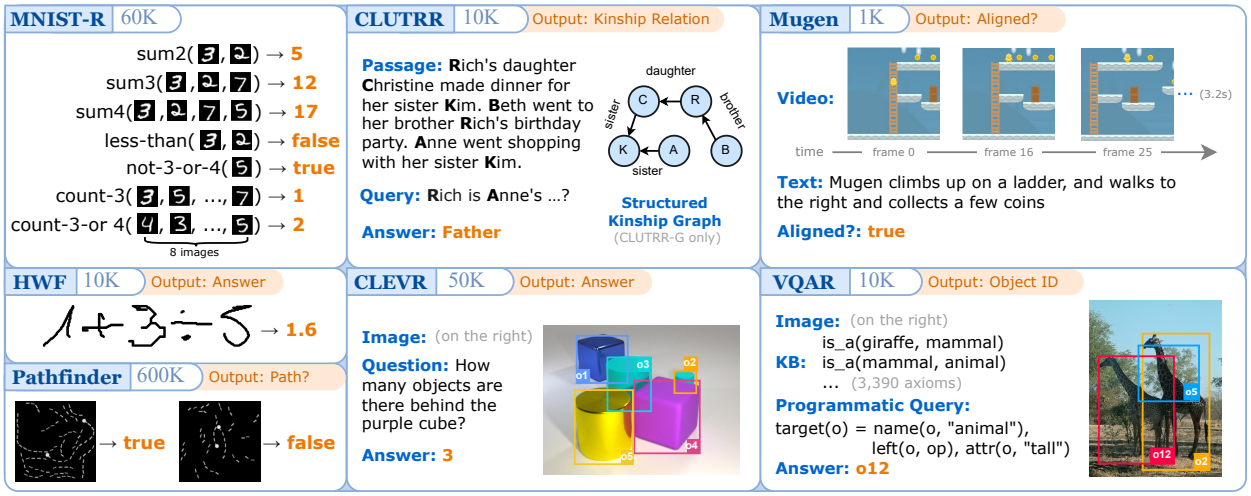


Figure 5.1: Visualization of benchmark tasks. Beside the name of each task we specify the size of the training dataset and the output domain. PacMan-Maze is omitted since it will be presented in detail in Section 5.5.

5.1.1 Benchmarks and Baselines

We present an overview of our benchmarks in Figure 5.1. They cover a wide spectrum of tasks involving perception and reasoning. The input data modality ranges from images and videos to natural language texts and knowledge bases (KB). The size of the training dataset is also presented in the figure. We next elaborate on the benchmark tasks and their corresponding baselines.

MNIST-R *A Synthetic MNIST Test Suite.* This benchmark is designed to test various features of Scallop such as negation and aggregation. Each task takes as input one or more images of hand-written digits from the MNIST dataset (Lecun et al., 1998) and performs simple arithmetic (sum2, sum3, sum4), comparison (less-than), negation (not-3-or-4), or counting (count-3, count-3-or-4) over the depicted digits. For count-3 and count-3-or-4, we count digits from a set of 8 images. For this test suite, we use a CNN-based model, DeepProbLog (DPL) (Manhaeve et al., 2021), and our prior work (Huang et al., 2021) (Prior) as the baselines.

HWF *Hand-Written Formula Parsing and Evaluation.* HWF, proposed in Li et al. (2020a), concerns parsing and evaluating hand-written formulas. The formula is provided in the form of a sequence of images, where each image represents either a digit (0-9) or an arithmetic symbol

$(+, -, \times, \div)$. Formulas are well-formed according to a grammar and do not divide by zero. The size of the formulas ranges from 1 to 7 and is indicated as part of the input. The goal is to evaluate the formula to obtain a rational number as the result. We choose from Li et al. (2020a) the baselines NGS-m-BS, NGS-RL, and NGS-MAPO, which are *neurosymbolic methods* designed specifically for this task.

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

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

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

CLUTRR-G, we pick GAT (Veličković et al., 2017) and CTP (Minervini et al., 2020) as baselines.

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

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

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

5.1.2 RQ1: Our Solutions and Expressivity

To answer **RQ1**, we demonstrate our Scallop solutions to the benchmark tasks (Table 5.1). For each task, we specify the interface relations which serve as the bridge between the neural and symbolic components. The neural modules process the perceptual input and their outputs are mapped to (probabilistic) facts in the interface relations. Our Scallop programs subsequently take these facts as

Task	Input	Neural Net	Interface Relation(s)	Scallop Program	Features			LoC
					R	N	A	
MNIST-R	Images	CNN	`digit(`*id*, *digit*`)`	Arithmetic, comparison, negation, and counting.		✔	✔	2†
HWF	Images	CNN	`symbol(`*id*, *symbol*`)` `length(`*len*`)`	Parses and evaluates formula over symbols.	✔			39
Pathfinder	Image	CNN	`dot(`*id*`)` `dash(`*from_id*, *to_id*`)`	Checks if the dots are connected by dashes.	✔			4

PacMan-Maze	Image	CNN	`actor(`x, y`)`
			`enemy(`x, y`)`
			`goal(`x, y`)`
				Plans the optimal action by finding an enemy-free path from actor to goal.	✔	✔	✔	31

CLUTRR (-G)	NL	RoBERTa	`kinship(`*rela*, *sub*, *obj*`)`	Deduces queried relationship by composing kinship rules.	✔	✔	✔	8
	Query*	–	`question(`*sub*, *obj*`)`					
	Rule	–	`composition(`r_1, r_2, r_3`)`					
Mugen	Video	S3D	`action(`*frame*, *action*, *mod*`)`	Checks if events specified in text match actions in the video.	✔	✔	✔	46
	NL	DistilBERT	`expr(`*expr_id*, *action*`)` `mod(`*expr_id*, *mod*`)`					
CLEVR	Image	FastRCNN	`obj_attr(`*obj_id*, *attr*, *val*`)` `obj_rela(`*rela*, o_1, o_2`)`	Interprets CLEVR-DSL program (extracted from question) on scene graph (extracted from image).	✔	✔	✔	51
	NL	BiLSTM	`filter_expr(`*e*, *ce*, *attr*, *val*`)` `count_expr(`*e*, *ce*`)`, ...					

VQAR	Image	FastRCNN	`obj_name(`*obj_id*, *name*`)`
			`obj_attr(`*obj_id*, *val*`)`
			`obj_rela(`*rela*, o_1, o_2`)`
				Evaluates query over scene graphs with the aid of common-sense knowledge base (KB).	✔			42

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

Table 5.1: Characteristics of Scallop solutions for each task. Structured input which is not learnt is denoted by *. Neural models used are RoBERTa (Liu et al., 2019), DistilBERT (Sanh et al., 2019), and BiLSTM (Graves et al., 2013) for natural language (NL), CNN and FastRCNN (Girshick, 2015) for images, and S3D (Xie et al., 2018) for video. We show the three key features of Scallop used by each solution: (R)ecursion, (N)egation, and (A)ggregation. †: For MNIST-R, the LoC is 2 for every subtask.

input and perform the described reasoning to produce the final output. As shown by the *features* column, our solutions use all of the core features provided by Scallop.

The Scallop program for benchmark tasks are succinct, as indicated by the LoCs in the last column of Table 5.1. We highlight three tasks, HWF, Mugen, and CLEVR, to demonstrate Scallop's expressivity. For HWF, the Scallop program consists of a formula parser. It is capable of parsing probabilistic input symbols according to a context free grammar for simple arithmetic expressions. For Mugen, the Scallop program is a *temporal specification checker*, where the specification is extracted from NL text to match the sequential events excerpted from the video. For CLEVR, the Scallop program is an interpreter for CLEVR-DSL, a domain-specific functional language introduced in the CLEVR dataset (Johnson et al., 2017).

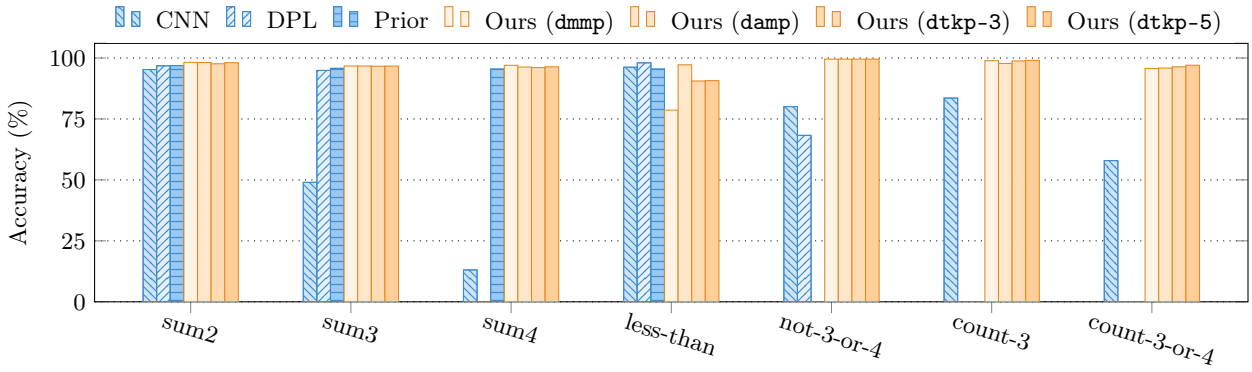


Figure 5.2: MNIST-R suite accuracy comparison.

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

Table 5.2: PacMan-Maze performance comparison to DQN.

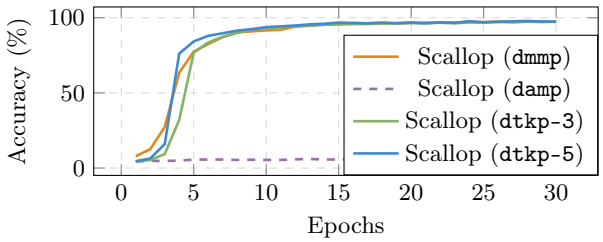


Figure 5.3: HWF learning curve.

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

Besides diverse kinds of perceptual data and reasoning patterns, the Scallop programs are applied in a variety of learning settings. As shown in Section 5.5, the program for PacMan-Maze is used in a *online representation learning* setting. For CLUTRR, we write integrity constraints (similar to the one shown in Section 2.2) to derive *semantic loss* Xu et al. (2017, 2018) used for constraining the language model outputs. For CLUTRR-G, learnable weights are attached to `composite` facts such as `composite(FATHER, MOTHER, GRANDMOTHER)`, which enables to learn such facts from data akin to *rule learning* in ILP. For Mugen, our program is trained in a *contrastive learning* setup, since it requires to maximize similarity scores between aligned video-text pairs but minimize that for un-aligned ones.

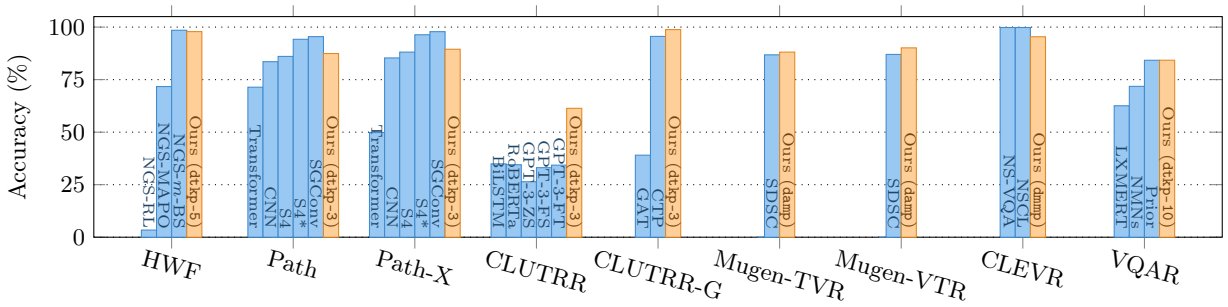


Figure 5.4: Overall benchmark accuracy comparison. The best-performing provenance structure for our solution is indicated for each task. Among the shown tasks, `dtkp` performs the best on 6 tasks, `damp` on 2, and `dmmp` on 1.

5.1.3 RQ2: Performance and Accuracy

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

We start with comparing our solutions against selected baselines on all the benchmark tasks, as shown in Figure 5.2, Table 5.2, and Figure 5.4. First, we highlight two applications, PacMan-Maze and CLUTRR, which benefit the most from our solution. For PacMan-Maze, compared to DQN, we obtain a 1,000× speed-up in terms of training episodes, and a near perfect success rate of 99.4%. Note that our solution encodes environment dynamics (i.e. game rules) which are unavailable and hard to incorporate in the DQN model. For CLUTRR, we obtain a 25% improvement over baselines, which includes GPT-3-FT, the state-of-the-art large language model fine-tuned on the CLUTRR dataset. Next, for tasks such as HWF and CLEVR, our solutions attain comparable performance, even compared to neurosymbolic baselines NGS-m-BS, NSCL, and NS-VQA specifically designed for each task. On Path and Path-X, our solution obtains a 4% accuracy gain over our underlying model CNN and even outperforms a carefully designed transformer based model S4.

The performance of the Scallop solution for each task depends on the chosen provenance structure. As can be seen from Table 5.2 and Figs. 5.2–5.4, although `dtkp` is generally the best-performing one, each presented provenance is useful, e.g., `dtkp` for PacMan-Maze and VQAR, `damp` for less-than

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

Table 5.3: Runtime efficiency comparison on selected benchmark tasks. Numbers shown are average training time (sec.) per epoch. Our variants attaining the best accuracy are indicated in bold.

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

5.1.4 RQ3: Runtime Efficiency

We evaluate the runtime efficiency of Scallop solutions with different provenance structures and compare it against baseline neurosymbolic approaches. As shown in Table 5.3, Scallop achieves substantial speed-up over DeepProbLog (DPL) on MNIST-R tasks. DPL is a probabilistic programming system based on Prolog using exact probabilistic reasoning. As an example, on sum4, DPL takes 40 days to finish only 4K training samples, showing that it is prohibitively slow to use in practice. On the contrary, Scallop solutions can finish a training epoch (15K samples) in minutes without sacrificing testing accuracy (according to Figure 5.2). For HWF, Scallop achieves comparable runtime efficiency, even when compared against the hand-crafted and specialized NGS-m-BS method.

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

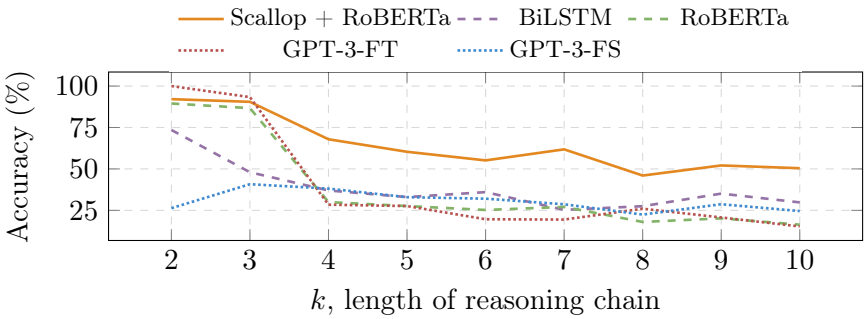


Figure 5.6: Systematic generalizability on CLUTRR dataset.

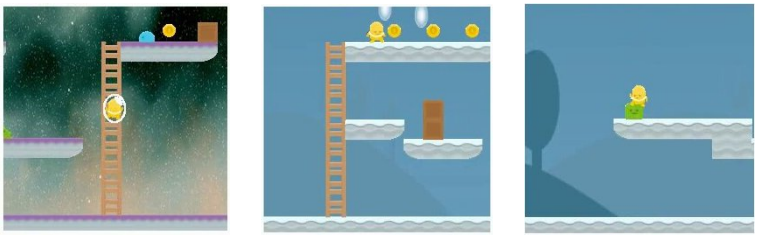


Figure 5.7: The predicted most likely (action, mod) pair for example video segments from Mugen dataset.

5.1.5 RQ4: Generalizability, Interpretability, and Data-Efficiency

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

We evaluate Scallop's generalization ability for the CLUTRR task. Each data-point in CLUTRR is annotated with a parameter k denoting the length of the reasoning chain to infer the target kinship relation. To test different solutions' *systematic generalizability*, we train them on data-points with $k \in \{2, 3\}$ and test on data-points with $k \in \{2, \ldots, 10\}$. As shown in Figure 5.6, the neural baselines suffer a steep drop in accuracy on the more complex unseen instances, whereas the accuracy of Scallop's solution degrades more slowly, indicating that it is able to generalize better.

Next, we demonstrate Scallop's interpretability on the Mugen task. Although the goal of the task is to see whether a video-text pair is aligned, the perceptual model in our method extracts interpretable

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

Table 5.4: Testing accuracy of various methods on HWF when trained with only a portion of the data. Numbers are in percentage (%).

symbols, i.e., the action of the controlled character at a certain frame. Figure 5.7 shows that the predicted (action, mod) pairs perfectly match the events in the video. Thus, our solution not only tells whether a video-text pair is aligned, but also *why* it is aligned.

Lastly, we evaluate Scallop's data-efficiency on the HWF task, using lesser training data (Table 5.4). While methods such as NGS-MAPO suffer significantly when trained on less data, Scallop's testing accuracy decreases slowly, and is comparable to the data-efficiency of the state-of-the-art neurosymbolic NGS-m-BS method. PacMan-Maze task also demonstrates Scallop's data-efficiency, as it takes much less training episodes than DQN does, while achieving much higher success rate.

5.1.6 RQ5: Analysis of Failure Modes

Compared to purely neural models, Scallop solutions provide more transparency, allowing programmers to debug effectively. By manually checking the interface relations, we observed that the main source of error lies in inaccurate predictions from the neural components. For example, the RoBERTa model for CLUTRR correctly extracts only 84.69% of kinship relations. There are two potential causes—either the neural component is not powerful enough, or our solution is not providing adequate supervision to train it. The former can be addressed by employing better neural architectures or more data. The latter can be mitigated in different ways, such as tuning the selected provenance or incorporating integrity constraints (discussed in Section 2.3) into training and/or inference. For instance, in PacMan-Maze, the constraint that "there should be no more than one *goal* in the arena" helps to improve robustness.

5.2 Scallop Benchmarks with Foundation Models

We apply Scallop to solve 9 benchmark tasks depicted in Figure 5.8 with foundation models. The tasks span the domains of natural language and vision, and require capabilities such as reasoning, information retrieval, and multi-modal compositionality. Table 5.5 summarizes the datasets, evaluation metrics, and the foundation models used in our solutions. We elaborate upon the evaluation settings and our solutions in Section 5.2.1. We aim to answer the following research questions:

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

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

5.2.1 Benchmarks

Date reasoning (DR) In this task adapted from BIG-bench (Srivastava et al., 2023), the model is given a context and asked to compute a date. The questions test the model's temporal and numerical reasoning skills, as well as its grasp of common knowledge. Unlike BIG-bench where multiple-choice answers are given, we require the model to directly produce its answer in MM/DD/YYYY form.

Our solution leverages GPT-4 (5-shot) for extracting 3 relations: mentioned dates, duration between date labels, and the target date label. Then, the program's relational rules yield the symbolic dates for all date labels. The rest of the program iteratively traverses through the date differences to compute dates for all occurring date labels. Lastly, the date of the target label is returned as the output.

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

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

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

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

Kinship reasoning (KR) CLUTRR (Sinha et al., 2019) is a kinship reasoning dataset of stories which indicate the kinship between characters, and requires the model to infer the relationship between two specified characters. The questions have different difficulty levels based on the length of the reasoning chain, denoted by $k \in \{2 \dots 10\}$.

Our solution for kinship reasoning invokes GPT-4 (2-shot) to extract the kinship graph from the context. We also provide an external common-sense knowledge base for rules like "mother's mother is

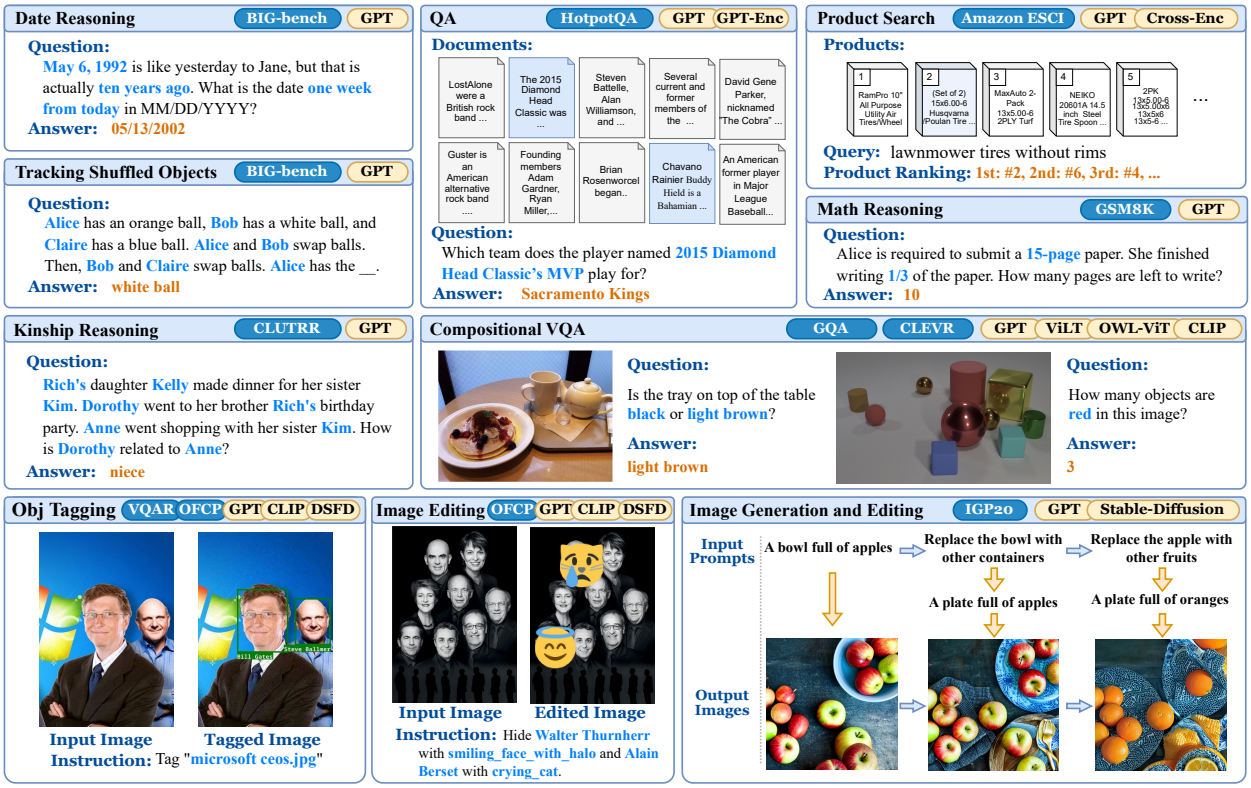


Figure 5.8: Benchmark tasks. The top of each box lists the dataset(s) and the foundation models used in our solutions.

grandmother". Our program then uses the rules to derive other kinship relations. Lastly, we retrieve the kinship between the specified pair of people.

Math reasoning (MR) This task is drawn from the GSM8K dataset of arithmetic word problems (Cobbe et al., 2021). The questions involve grade school math word problems created by human problem writers, and the model is asked to produce a number as the result. Since the output can be fractional, we allow a small delta when comparing the derived result with the ground truth.

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

Question answering with information retrieval (QA) We choose HotpotQA (Yang et al., 2018b), a Wikipedia-based question answering (QA) dataset under the "distractor" setting. Here, the model takes in 2 parts of inputs: 1) a question, and 2) 10 Wikipedia paragraphs as the context for answering the question. Among the 10 Wikipedia pages, at most 2 are relevant to the answer, while the others are distractors.

Our solution is an adaptation of FE2H (Li et al., 2022a), which is a 2-stage procedure. First, we turn the 10 documents into a vector database by embedding each document. We then use the embedding of the question to retrieve the 2 most related documents, which are then fed to a language model to do QA. In this case, the QA model does not have to process all 10 documents, leading to less distraction.

Product search (PS) We use Amazon's ESCI Product Search dataset (Reddy et al., 2022). The model is provided with a natural language (NL) query and a list of products (23 products on average). The goal is to rank the products that best match the query. In the dataset, for each pair of query and product, a label among E (exact match), S (substitute), C (complementary), and I (irrelevant) is provided. The metric we use to evaluate the performance is nDCG. The gains are set to be 1.0 for E, 0.1 for S, 0.01 for C, and 0.0 for I.

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

Compositional visual question answering (VQA) We choose two compositional VQA datasets, GQA (Hudson and Manning, 2019a) and CLEVR (Johnson et al., 2016). In this task, the model is given an image and a question, and needs to answer the question. For GQA, the majority of questions expect yes/no answers, while CLEVR's questions demand features like counting and spatial reasoning. We uniformly sample 500 and 480 examples from GQA and CLEVR datasets respectively.

Following VQA conventions (Kim et al., 2021), we use Recall@k where $k \in \{1, 3\}$ as the evaluation metrics.

Our solution for GQA is an adaptation of VisProg (Gupta and Kembhavi, 2022). We create a DSL for invoking vision modules such as ViLT and OWL-ViT, and use GPT-4 for converting questions into programs in this DSL. Our solution for CLEVR is similar, directly replicating the DSL provided by the original work. OWL-ViT and CLIP are used to detect objects and infer attributes, while the spatial relations are directly computed using the bounding box data.

Visual object tagging (VOT) We evaluate on two datasets, VQAR (Huang et al., 2021) and OFCP. For VQAR, the model is given an image and a programmatic query, and is asked to produce bounding boxes of the queried objects in the image. Our solution composes a relational knowledge base, defining entity names and relationships, with object retrieval (OWL-ViT) and visual QA (ViLT) models.

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

Language-guided image generation and editing (IGE) We adopt the task of image editing from Gupta and Kembhavi (2022). In this task, the instruction for image editing is provided through NL, and can invoke operations such as blurring background, popping color, and overlaying emojis. Due to the absence of an existing dataset, we repurpose the OFCP dataset by introducing 50 NL image editing prompts. Our solution for this task is centered around a DSL for image editing. We incorporate GPT-4 for semantic parsing, DSFD for face detection, and CLIP for entity classification. Modules for image editing operations are implemented as individual foreign functions.

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

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

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

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

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

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

For free-form generation and editing of images, we curate IGP20, a set of 20 prompts for image generation and editing. Instead of using the full prompt, we employ an LM to decompose complex NL instructions into simpler steps. We define a DSL with high-level operators such as generate, reweight, refine, replace, and negate. We use a combination of GPT-4, Prompt-to-Prompt (Hertz et al., 2022), and diffusion model (Rombach et al., 2022) to implement the semantics of our DSL. We highlight our capability of grounding positive terms from negative phrases, which enables handling prompts like "replace apple with other fruits" (Figure 5.8).

5.2.2 RQ1: Programmability

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

5.2.3 RQ2: Baselines and Comparisons

We compare the performance of our solutions to existing baselines under the no-training setting. In particular, our solutions achieve better performance than comparable baselines on 6 out of 8 studied datasets with baselines. Below, we classify the tasks into 4 categories and discuss the respective performance and comparisons.

Natural language reasoning For the tasks of DR, TSO, CLUTRR, and GSM8K, we pick a generic baseline of GPT-4 under zero-shot, few-shot, and chain-of-thought (CoT) settings. All our solutions also rely on GPT-4 (few-shot), but we note that our shots only include extracted facts, and not the final answer or any reasoning chains. The data in Table 5.7 indicates that our method can significantly enhance reasoning performance and reduce hallucination, exemplified by achieving a flawless 100% accuracy on the TSO dataset. Note that on GSM8K, our method scores slightly lower than the baseline; we conjecture that our solution demands more from GPT-4 itself to extract structured computation steps. On CLUTRR, our solution even outperforms Faithful Chain-of-thought (fCoT) (Lyu et al., 2023), a special prompting technique with external tool use, by 0.6%. In Figure 5.10 we illustrate the systematic generalizability of our methods. The performance of our solutions remains relatively consistent even when the problems become harder. We provide illustrative examples in Figure 5.9 showing comparisons between our method and GPT-4 (zero-shot

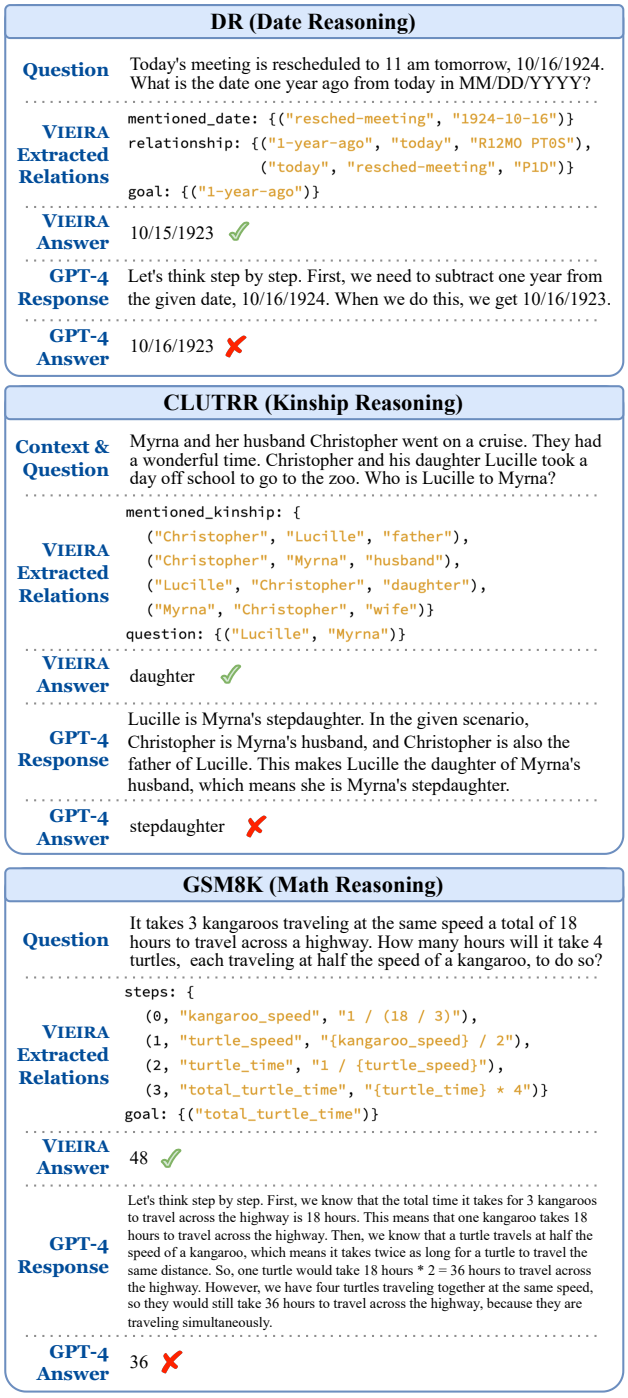
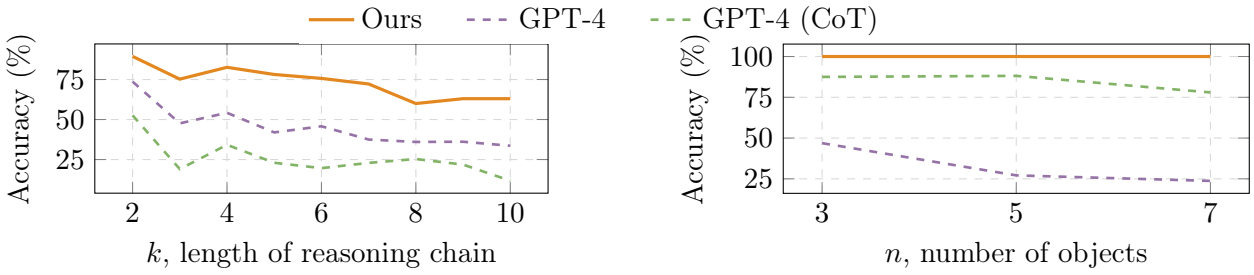


Figure 5.9: Illustrative comparisons between our solution and GPT-4 (zero-shot CoT) on selected questions from DR, CLUTRR, and GSM8K datasets. We also include the extracted relations used for subsequent reasoning.

CoT).



(a) CLUTRR (b) TSO

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

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

Table 5.9: Quantitative results on the VQA datasets.

Retrieval augmentation and semantic search For the HotpotQA dataset, our solution is an adaptation of FE2H (Li et al., 2022a), a retrieval-augmented question answering approach. As seen in Table 5.8, with no fine-tuning, our method scores only a few percentages lower than fine-tuned methods C2FM Yin et al. (2022) and FE2H. For the Amazon ESCI dataset, our solution performs semantic search for product ranking. While performing slightly lower than the fine-tuned methods (Reddy et al., 2022; Song et al., 2020), our solution outperforms maximum inner product search (MIPS) based on GPT text encoder (`text-embedding-ada-002`).

Compositional multi-modal reasoning For VQA, we pick ViLT-VQA (Kim et al., 2021) (a pre-trained foundation model) and PNP-VQA (Tiong et al., 2022) (a zero-shot VQA method) as baselines. As shown in Table 5.9, our method significantly outperforms the baseline model on both datasets. Compared to the neural-only baseline, our approach that combines DSL and logical reasoning more effectively handles intricate logical operations such as counting and numerical comparisons. On GQA, out method outperforms previous zero-shot state-of-the-art, PNP-VQA, by 0.16 (0.42 to 0.58). For object and face tagging, without training or fine-tuning, our method achieves 67.61% and 60.82% semantic correctness rates (Table 5.10).

Original Ours InstructPix2Pix

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

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

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

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

Image generation and editing For image generation and editing, we apply our technique to the OFCP and IGP20 datasets. We rely on manual inspection for evaluating our performance on the OFCP dataset, and we observe 37 correctly edited images out of the 50 evaluated ones, resulting in a 74% semantic correctness rate (Table 5.10). For IGP20, we choose as the baseline a diffusion model, InstructPix2Pix Brooks et al. (2023), which also combines GPT-3 with image editing. We show one example baseline comparison illustrated in Figure 5.11.

5.3 Case Study: Summing Two MNIST Digits

The `MNIST-Sum2` task from Manhaeve et al. (2021) concerns classifying sums from pairs of handwritten digits, e.g., 3 + 7 = 10. A model receives only the two MNIST digits as the input, and need to learn to recognize the two digits with only the supervision of the sum.

As depicted in Figure 5.12, we specify this task using a neural and a symbolic component, following the style of DeepProbLog (Manhaeve et al., 2021). The neural component is a perception model

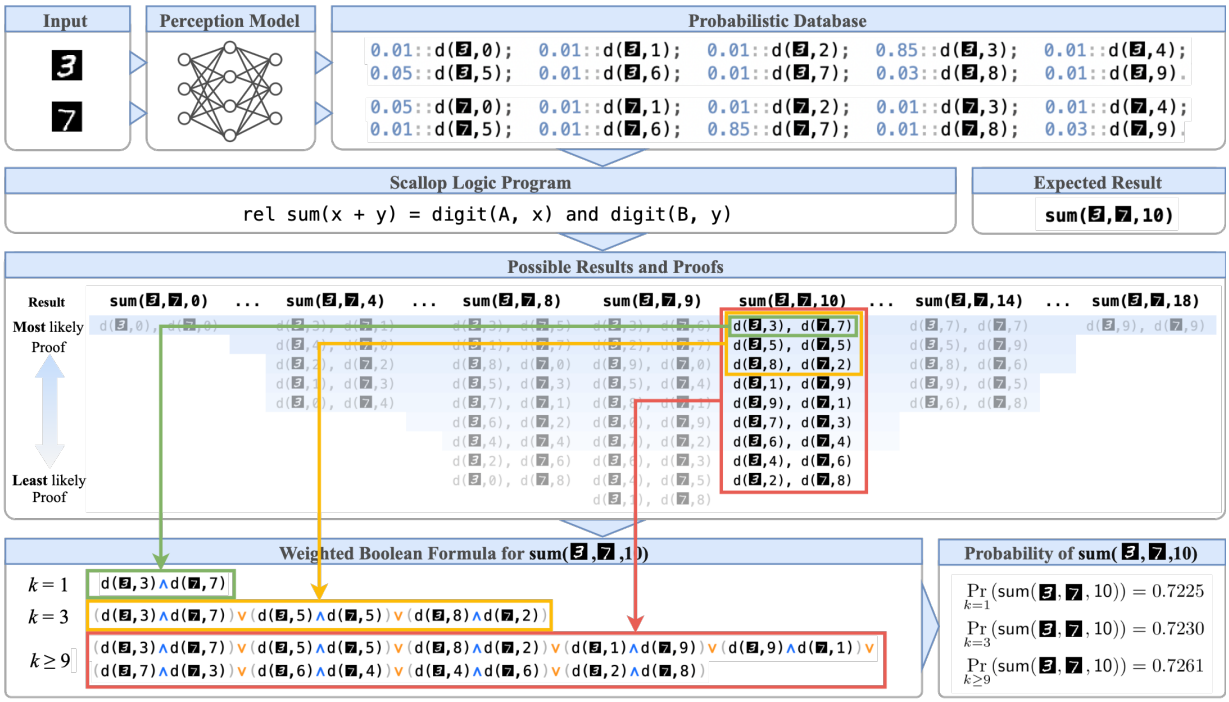


Figure 5.12: Illustration of applying Scallop's top-k-proofs provenance on the task 3 + 7 = 10 using different values of parameter k.

that takes in an image of hand-written 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 :: d(3, 3) denotes that image 3 is recognized to be the digit 3 with probability 0.85. The database thus consists of 20 facts—one for each of the 10 possible digits corresponding to each of the two images.

Evaluating the logic program on the probabilistic database yields a weighted boolean formula for each possible result of the sum of two digits, i.e., values in the range $\{0, \ldots, 18\}$. Each *clause* of such a formula represents a different *proof* of the corresponding result. For instance, the bottom left of Figure 5.12 shows the formula representing all 9 proofs of the ground truth result 10. Each such formula is input to an off-the-shelf weighted model counting (WMC) solver to yield the probability of the corresponding result, e.g., 0.7261 :: sum(10).

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

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

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

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

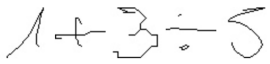


Figure 5.13: One hand-written formula $1 + 3 \div 5$ which should evaluate to 1.6.

relational symbols (and vice versa for the output). Line 13 configures the provenance to use for the reasoning, which is the `dtkp` provenance with $k = 1$.

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

5.4 Case Study: Evaluating Handwritten Formulas

In this case study we take the `MNIST-Sum2` one step further by allowing multiple symbols including hand-written digits and also simple arithmetic operators like $+$, $-$, $\times$, and $\div$. This is the task of hand-written formula evaluation (HWF) (Li et al., 2020a). The input to the task is a sequence of images of hand-written symbols, forming a hand-written formula. The output is the rational number result of evaluating the formula. The dataset provided for this task contains variable-sized formulas with 1 to 7 symbols, where the operands are all single-digit numbers. For brevity of exposition, we presume that the input formulas always parse and are free of divide-by-zero errors.

A natural solution to this task is to decompose the problem into separate perception and reasoning components. The perception component is a standard convolutional neural network (CNN) that classifies each symbol into discrete classes (digits 0-9 and $+$, $-$, $\times$, $\div$). The reasoning component then takes in the classified probabilistic symbols, parses and evaluates the formula, and returns a probability distribution of the result. Notably, the neural model does not receive supervision on the label of each individual symbol in the formula. Instead, we only have supervision on the final

```
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) = symbol(b, x) and digit(x)
14
15 // A mult/div expression
16 type term(value: f32, begin: usize, end: usize)
17 rel term(x, b, r) = factor(x, b, r)
18 rel term(x * y, b, e) = term(x, b, m) and symbol(m, "*") and
19                         factor(y, m + 1, e)
20 rel term(x / y, b, e) = term(x, b, m) and symbol(m, "/") and
21                         factor(y, m + 1, e)
22
23 // An add/minus expression which has higher precedence
24 type expr(value: f32, begin: usize, end: usize)
25 rel expr(x, b, r) = term(x, b, r)
26 rel expr(x + y, b, e) = expr(x, b, m) and symbol(m, "+") and
27                         term(y, m + 1, e)
28 rel expr(x - y, b, e) = expr(x, b, m) and symbol(m, "-") and
29                         term(y, m + 1, e)
30
31 // Obtain the result
32 rel result(y) = expr(y, 0, l) and length(l)
```

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

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.

The reasoning component is written in Scallop as shown in Listing 5.2. The program uses Datalog-like syntax. It specifies two input relations, `symbol` and `length` (line 2-3). The former relates each symbol image's index with its recognized symbol (digits and operators represented as strings), and

```
1  class HWFNet(nn.Module):
2    def __init__(self):
3      MAX_LEN = 7
4      ALPHABET = ["0", ..., "9", "+", "-", "*", "/"]
5      # other setup code...
6      self.symbol_cnn = SymbolNet() # Symbol recognition
7      # Scallop module for formula evaluation
8      self.eval_formula = scallopy.Module(
9        file="hwf.scl", provenance="diff-top-k-proofs", k=3,
10       input_mappings={"symbol": scallopy.InputMapping(
11         {0: range(MAX_LEN), 1: ALPHABET},
12         retain_k=3, sample_dim=1)},
13       output="result", non_probabilistic=["length"])
14
15   def forward(self, img_seq, img_seq_len):
16     length = [[(l.item(),)] for l in img_seq_len]
17     # First recognize the symbols
18     symbol = self.symbol_cnn(img_seq.flatten(0, 1)).view(len(length),
           -1)
19     # Then evaluate the formula with Scallop
20     (out_symbols, out_distr) = self.eval_formula(symbol=symbol,
         length=length)
21     return (out_symbols, out_distr)
```

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

the latter encodes the length of the formula. The rest of the program defines relations `factor` (lines 9-10), `term` (lines 12-15), and `expr` (lines 17-20), going up the standard context-free grammar of simple arithmetic expressions. The first argument of each of these relations is a floating point number, with type `f32`, denoting the evaluated results of the corresponding expressions. Lastly, we fetch the expression which covers the whole formula (line 23), and store the evaluated result in the `result` unary relation.

Next, we may integrate this program into an end-to-end learning pipeline. Listing 5.3 shows the PyTorch module for the HWF task. During initialization, we setup the CNN to process each symbol image (line 6). Then we create a Scallop module to load the program from file `hwf.scl` (line 8-13). We also configure the provenance semiring to be used as `diff-top-k-proofs` with `k` set to 3. During the training or inference phase, we simply pass the symbol images to the CNN (line 17) and the result distributions to our Scallop module (line 19). Since both the CNN and the Scallop module

are differentiable, we obtain an end-to-end learning pipeline. While being conceptually similar, there are a few core complexities of HWF when compared to `MNIST-Sum2`. We now explain each of them and how the Python interface helps to ease the handling of such complexities.

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

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

Unbounded set of outputs Instead of a fixed set of possible outputs ($\{0, \ldots, 18\}$) in `MNIST-Sum2`, HWF has a much larger set of potential outputs, due to the fact that formulas contain division operator ($\div$). It is not realistic to enumerate all of them, which is why we do not explicitly specify an output mapping. The outcome of this is that the `eval_formula` cannot return a straightforward

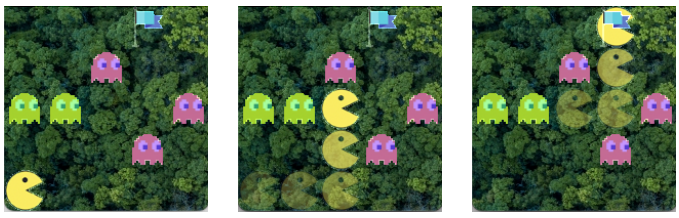
```
# calling hwf_net yields the set of outputs as well as the
# predicted distributions over the set of outputs
(outputs, y_pred) = hwf_net(formula_imgs)

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

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

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

vectorized tensor as the output. As shown on line 19, we obtain two results, `out_symbols` and `out_distr`. Specifically, `out_symbols` will be a list of fraction numbers that are actually derived with the sampled inputs. Meanwhile, `out_distr` will contain computed distribution over the results in `out_symbols`. We present in Listing 5.4 the loss function that is used to process such output.

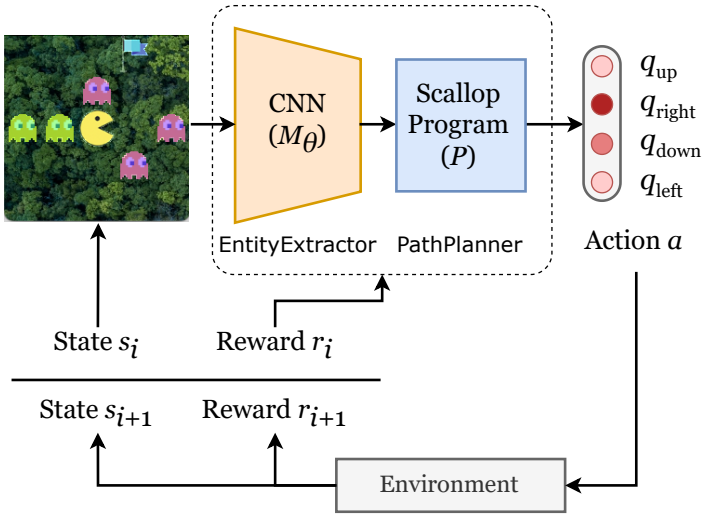
5.5 Case Study: Playing the PacMan-Maze Game

We further illustrate Scallop using an reinforcement learning (RL) based planning application which we call PacMan-Maze. The application, depicted in Figure 5.14a, concerns an intelligent agent realizing a sequence of actions in a simplified version of the PacMan maze game. The maze is an implicit 5×5 grid of cells. Each cell is either empty or has an entity, which can be either the *actor* (PacMan), the *goal* (flag), or an *enemy* (ghost). At each step, the agent moves the actor in one of four directions: up, down, right, or left. The game ends when the actor reaches the goal or hits an enemy. The maze is provided to the agent as a raw image that is updated at each step, requiring the agent to process sensory inputs, extract relevant features, and logically plan the path to take. Additionally, each session of the game has randomized initial positions of the actor, the goal, and the enemies.



Step 0 Step 4 Step 7

(a) Three states of one gameplay session.



(b) Architecture of application with Scallop.

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

Concretely, the game is modeled as a sequence of interactions between the agent and an environment, as depicted in Figure 5.14b. The game state $s_i \in S$ at step i is a 200×200 colored image ($S = \mathbb{R}^{200\times200\times3}$). The agent proposes an action $a_i \in A = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$ to the environment, which generates a new image s_{i+1} as the next state. The environment also returns a reward r_i to the agent: 1 upon reaching the goal, and 0 otherwise. This procedure repeats until the game ends or reaches a predefined limit on the number of steps.

A popular RL method to realize our application is Q-Learning (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

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

```
class PacManAgent(torch.nn.Module):
  def __init__(self, grid_dim, cell_size):
    # initializations...
    cells = [(i,j) for i in range(grid_dim) for j in range(grid_dim)]
    actions = [UP, RIGHT, DOWN, LEFT]
    self.extract_entities = EntityExtractor(grid_dim, cell_size)
    self.path_planner = ScallopModule(
      file="path_planner.scl",
      provenance="diff-top-k-proofs", k=1,
      input_mappings={"actor":cells, "goal":cells, "enemy":cells},
      output_mappings={"next_action": actions})

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

Listing 5.5: Snippet of implementation in Python.

the Q function using a convolutional neural network (CNN) model with learned parameter θ. An end-to-end deep learning based approach for our application involves training the model to predict the Q-value of each action for a given game state. This approach takes 50K training episodes to achieve a 84.9% test success rate, where a single episode is one gameplay session from start to end.

In contrast, a neurosymbolic solution using Scallop only needs 50 training episodes to attain a 99.4% test success rate. Scallop enables to realize these benefits of the neurosymbolic paradigm by decomposing the agent's task into separate neural and symbolic components, as shown in Figure 5.14b. These components perform sub-tasks that are ideally suited for their respective paradigms: the neural component perceives pixels of individual cells of the image at each step to identify the entities in them, while the symbolic component reasons about enemy-free paths from the actor to the goal to determine the optimal next action. Figure 5.5 shows an outline of this architecture's implementation using the popular PyTorch framework.

Concretely, the neural component is still a CNN, but it now takes the pixels of a single cell in the input image at a time, and classifies the entity in it. A snippet of the overall Scallop application in

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

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

Python is shown in Figure 5.5. The implementation of the neural component (`EntityExtractor`) is standard and elided for brevity. It is invoked on lines 14-15 with input `game_state_image`, a tensor in $\mathbb{R}^{200\times200\times3}$, and returns three $\mathbb{R}^{5\times5}$ tensors of entities. For example, `actor` is an $\mathbb{R}^{5\times5}$ tensor and $\texttt{actor}_{ij}$ is the probability of the actor being in cell (i, j). A representation of the entities is then passed to the symbolic component on lines 16-17, which derives the Q-value of each action. The symbolic component, which is configured on lines 6-11, comprises the Scallop program shown in Figure 5.6. We next illustrate three key design decisions of Scallop with respect to this program.

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

Symbols extracted from unstructured inputs by neural networks have associated probabilities, such as the $\mathbb{R}^{5\times5}$ tensor `actor` produced by the neural component in our example (line 14 of Listing 5.5). Scallop therefore allows to associate tuples with probabilities, e.g. 0.96 :: `actor(2,3)`, to indicate that the actor is in cell (2,3) with probability 0.96. More generally, Scallop enables the conversion of tensors in the neural component to and from relations in the symbolic component via input-output mappings (lines 9-11 in Listing 5.5), allowing the two components to exchange information seamlessly.

Declarative language Another key consideration in a neurosymbolic language concerns what constructs to provide for symbolic reasoning. Scallop uses a declarative language based on Datalog, which we illustrate here using the program in Listing 5.6. The program realizes the symbolic component of our example using a set of logic rules. Instead of having to explicitly encode a searching algorithm for path-finding, the logic can be declaratively specified in Scallop, simplifying the programming experience from an end-user point-of-view.

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

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

Differentiable reasoning With the neural and symbolic components defined, the last major consideration concerns how to train the neural component using only end-to-end supervision. In our

[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.

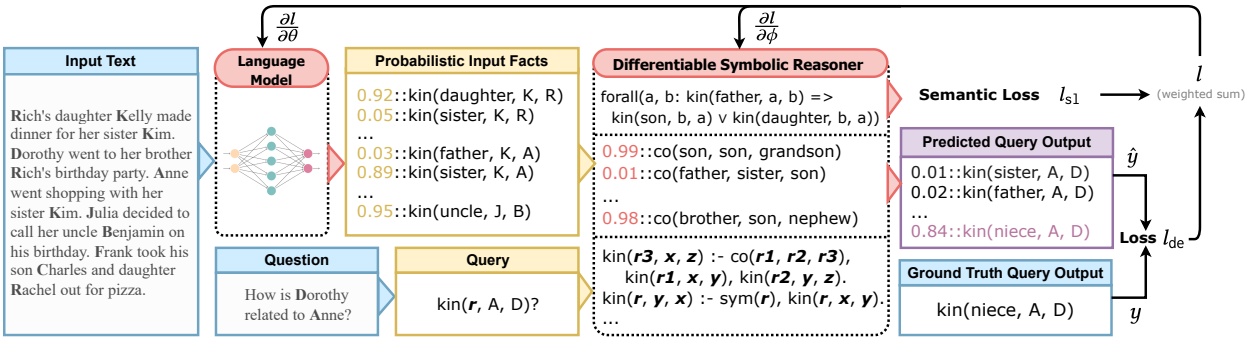


Figure 5.15: Overview of kinship reasoning with an example where "Anne is the niece of Dorothy" can be inferred from the context. We abbreviate the names with their first initials in the relational symbols, and the composite relationship with "co".

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

The key insight in Scallop is to exploit the structure of the logic program to guide the gradient calculations, as achieved by the differentiable provenances implemented within our provenance framework. In our example, line 8 in Figure 5.5 specifies `diff-top-k-proofs` with `k=1` as the heuristic to use, which is the default in Scallop that works best for many applications.

5.6 Case Study: Learning Composition Rules for Kinship Reasoning

CLUTRR (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 5.15. The input text is *"Rich's daughter Kelly made dinner for her sister Kim. Dorothy went to her brother Rich's birthday party. Anne went shopping*

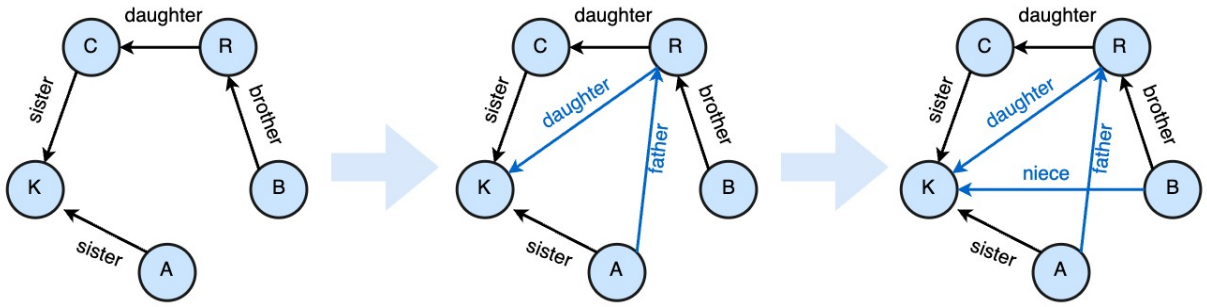


Figure 5.16: The family graph corresponding to the story shown in Figure 5.15. Edges representing family relations directly extracted from the story are colored in black, while those requiring derivation using common sense knowledge are colored in blue. Additionally, names are abbreviated using their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

with her sister Kim." From this narrative, we infer several relationships: Rich is Dorothy's brother, Kelly is Rich's daughter, Kim is Kelly's sister, and Anne is Kim's sister. Leveraging our common sense knowledge, we understand that one's sister's sister is also her sister, a sister's father is her father, and a brother's daughter is his niece. Consequently, we deduce that Anne is Kelly's sister, making Rich Anne's father, and Dorothy, Anne's aunt.

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

5.6.1 Structured Representation: Family Graph

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

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

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

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

Table 5.11: Higher-order predicate examples.

The logic for reasoning over kinship relations is realized in Scallop in Listing 5.7. 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 have line 7 declaring the rule that composites two existing kinship facts to derive a new kinship fact.

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

5.6.2 Learning Pipeline

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

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

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

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

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

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

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

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

5.6.3 Relation Extraction

Since pre-trained LMs have strong pattern recognition capabilities for tasks like Named-Entity-Recognition (NER) and Relation Extraction (RE) 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 pre-trained language model, to be fine-tuned, that converts windowed text into embeddings, and 3) a classifier that takes in the embedding of entities and predicts the relationship between them. The set of parameters θ contains the parameters of both the LM and the classifier.

We assume the relations to be classified come from a finite set of relations $\mathbf{R}$. For example in CLUTRR Sinha et al. (2019), we have 20 kinship relations including mother, son, uncle, father-in-law, etc. In practice, we perform $(|\mathbf{R}| + 1)$-way classification over each pair of entities, where the extra class stands for "n/a". The windowed contexts are split based on simple heuristics of

"contiguous one to three sentences that contain at least two entities", to account for coreference resolution. The windowed contexts can be overlapping and we allow the reasoning module to deal with noisy and redundant data. Overall, assuming that there are m windows in the context x, we extract $mn(n-1)(|\mathbf{R}|+1)$ probabilistic relational symbols. Each symbol is denoted as an atom of the form $p(s, o)$, where $p \in \mathbf{R} \cup \{\text{n/a}\}$ is the relational predicate, and s, o are the two entities connected by the predicate. We denote the probability of such symbol extracted by the LM and relational classifier as $\Pr(p(s, o) \mid \theta)$. All these probabilities combined form the output vector $\mathbf{r} = \mathcal{M}_\theta(x) \in \mathbb{R}^{mn(n-1)(|\mathbf{R}|+1)}$.

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

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

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

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

Note that LMs often make simple mistakes answering such prompt. In fact, with the above prompt, even GPT-3 can only produce 62% of composition rules correctly. While we can edit prompt to

include few-shot examples, in this work we consider fine-tuning such rule weights ϕ within our differentiable reasoning pipeline. Note that there are exponentially many rule weights to be fine-tuned. For example, the composition rule used for kinship reasoning has 3 arguments, resulting in $|R|^3 = 20^3$ candidate rules.

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

Semantic loss and integrity constraints In general, learning with weak supervision label is hard, not to mention that the deductive rules are learnt as well. We thereby introduce an additional semantic loss during training (Xu et al., 2017, 2018). Here, semantic loss is derived by a set of integrity constraints used to regularize the predicted entity-relation graph as well as the learnt logic rules. In particular, we consider rules that detect *violations* of integrity constraints. For example, "if A is B's father, then B should be A's son or daughter" is an integrity constraint for relation extractor—if the model predicts a father relationship between A and B, then it should also predict a son or daughter relationship between B and A. Encoded in first order logic, it is

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

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

5.6.4 Experimental Results

The baseline and Scallop performance results are shown in in Figure 5.4, which shows that our neurosymbolic solution outperforms all compared neural baselines by a large margin. With a more advanced language model (GPT-4), Scallop is capable of reaching an even higher accuracy (Table 5.7) on CLUTRR. We also show the top 20 learnt composition rules from the CLUTRR and CLUTRR-G

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)

Table 5.12: Showcase of the learnt logic rules (expressed as first order Horn rules) with top@20 confidence of CLUTRR rule learning.

experiments in Table 5.12. All top-20 learned rules match our expected real-life kinship relations.

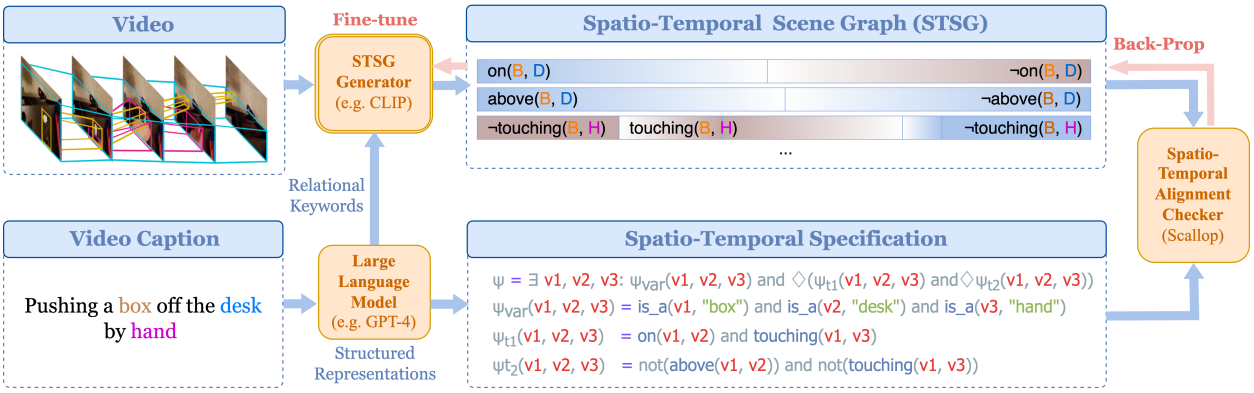


Figure 6.1: An example from 20BN demonstrating the end-to-end learning pipeline. The model M_θ processes a video to generate a probabilistic STSG. With 3-shot GPT-4, an STSL specification is derived from the video caption, which describes a temporal sequence of two events: "the box is on the desk touched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

CHAPTER 6

APPLICATION: VIDEO SCENE GRAPH GENERATION

Understanding video semantics has become increasingly important due to its wide range of applications, including video search, text-video retrieval, question answering, segmentation, and captioning. Video semantics can be decomposed into two essential dimensions: spatial semantics, which concern the entities present in a video, their attributes, and their spatial relationships; and temporal semantics, which capture how these elements and their interactions evolve over time. For example, the phrase "pushing a box off the desk by hand" (Figure 6.1) involves spatial relationships such as "touching" between "box" and "hand", and a temporal progression where the "box" transitions from being "on" the "desk" to being "not above" it.

To model such rich spatial and temporal semantics, prior work has proposed Spatio-Temporal Scene Graphs (STSGs) (Shang et al., 2017; Zhu et al., 2022) as structured representations that track entity relations over time. However, existing approaches for learning STSGs typically rely on fully supervised training (Nag et al., 2023; Cong et al., 2021), which requires costly low-level annotations that are impractical at scale (Yang et al., 2023b). In this work, we explore weakly supervised

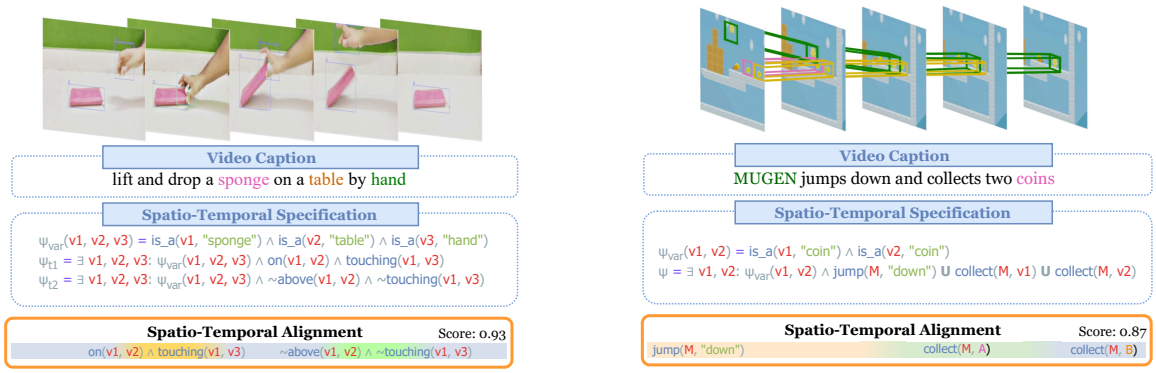
alternatives by combining STSGs with formal logic specifications, forming a novel framework that enables fine-grained video understanding without the need for expensive annotation.

Central to our approach is the Spatio-Temporal Specification Language (STSL), a domain-specific language grounded in finite linear temporal logic (LTL_f) (De Giacomo and Vardi, 2013), which allows users to write expressive logical descriptions of video content. These specifications can capture temporal operators such as "until" (**U**) and "finally" ($\Diamond$), as well as spatial predicates like "is pushing off" or "lies above". We implement a differentiable specification checker in the Scallop neurosymbolic framework that computes an alignment score between predicted STSGs and STSL programs, enabling end-to-end learning of the STSG model from weak supervision.

To address the scarcity of logical specifications in existing datasets, we design a prompting strategy that uses large language models (e.g., GPT-4 (OpenAI et al., 2024)) to convert natural-language captions into STSL programs. This allows us to use widely available video-caption pairs as a source of supervision. We further incorporate multiple learning signals—including contrastive alignment, time-span supervision, and semantic loss (Xu et al., 2017, 2018)—to enhance training effectiveness.

Building on these ideas, we introduce **LASER**, a neurosymbolic learning pipeline, and **SGClip**, a CLIP-based STSG generator trained using 87,000 video-specification pairs curated in the ESCA-Video-87K dataset. LASER is effective in fine-tuning a range of vision-language models, while SGClip demonstrates both strong zero-shot generalization and fine-tuning performance on downstream tasks such as action recognition. Together, LASER and SGClip show that large-scale STSG learning is feasible using only high-level video captions and logical reasoning as supervision.

This chapter introduces LASER, a neurosymbolic learning pipeline for spatio-temporal scene graph (STSG) generation. We begin with an illustrative overview (Section 6.1) of the STSG task and its challenges, followed by a formal problem definition (Section 6.2). We then present our neurosymbolic solution using Scallop (Section 6.3), including the LASER learning pipeline (Section 6.3.1) and the training of SGClip, a large-scale video scene graph model (Section 6.3.2). We conclude with an empirical evaluation (Section 6.4) and a discussion of related work (Section 6.5).



(a) 20BN (b) MUGEN

Figure 6.2: Two illustrative examples of videos and captions and their spatio-temporal alignment, from the 20BN and the MUGEN dataset.

6.1 Illustrative Overview

We illustrate our approach using two examples adapted from the 20BN dataset (Goyal et al., 2017) and the MUGEN dataset (Hayes et al., 2022) (Figure 6.2). The 20BN dataset contains real-life short videos involving a human performing simple actions, such as "push", "move-towards", and "lift", on other objects. The MUGEN dataset contains short gameplay footage of a 2D platform game *CoinRun*. In the video game, the player character MUGEN can walk, jump, collect coins, and kill enemies.

Figure 6.2a shows one spatio-temporal specification describing the pre- and post-condition of a single high-level action "push". Before a box is pushed off the desk, it must be on the desk (`on(v1, v2)`), and a hand is touching it (`touching(v1, v3)`). After it is pushed off the desk, it must be under the desk (`¬above(v1, v2)`). These predicates that encoding static spacial relation between entities are connected by temporal logic modality (e.g., $\Diamond$ (finally)) to express the dynamic change of spacial relation, that is, a temporal ordering over the moments before (ϕ_{pre}) and after (ϕ_{post}) the action "push" happens.

Figure 6.2b shows another specification connecting three events "jump", "collect", and "collect" together using the **U** (until) operator. This suggests that the three events are likely consecutive.

Different from the traditional sequence of actions (Chang et al., 2019; Huang et al., 2016), the events here are relational predicates connecting entities and possibly properties. For example, `jump(M, ''down'')` specifies that the actor of jump is (M)UGEN, and that the direction of jump is "down".

Despite expressing diverse semantics, both specifications can be uniformly represented under our LTL_f based formalism. The next question is how to learn the underlying STSG from the video and specifications. In this work, we frame it as a weakly-supervised specification alignment problem, illustrated in Figure 6.1. That is, the STSG generated from video should align with (i.e., satisfy) the corresponding logic specification. We design a neurosymbolic alignment checker to compute an alignment score, i.e., the probability of alignment. During training, our approach learns to extract an STSG such that it maximizes the alignment score between a corresponding pair of video and specification. With additional semantic and contrastive loss, we show that our method can effectively generate an STSG that conforms to the specification. In the upcoming section, we elaborate on the neurosymbolic learning framework as well as a foundation model, SGClip, that we have trained

6.2 Problem Definition

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

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

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

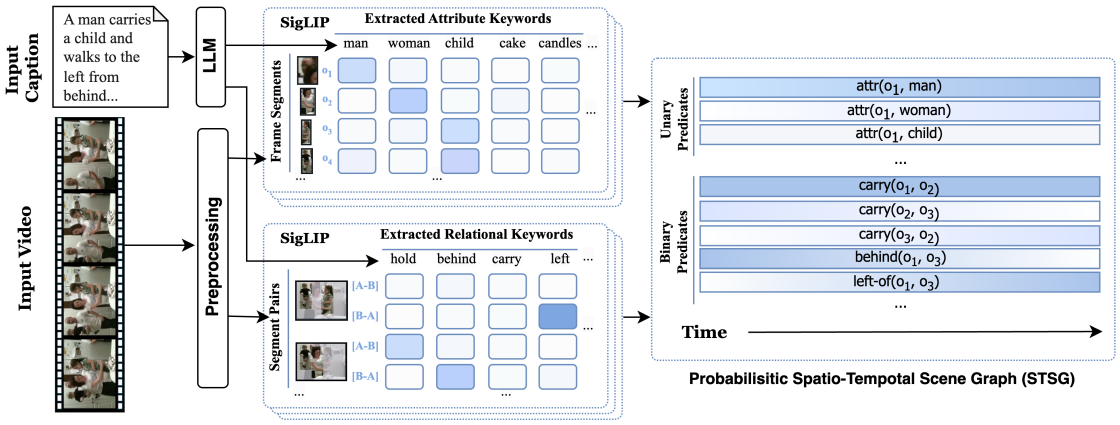


Figure 6.3: Pipeline illustration with SigLIP (Zhai et al., 2023) as the backbone model for probabilistic STSG generation.

6.3 Neurosymbolic Solution with Scallop

6.3.1 LASER: Neurosymbolic Learning for Scene Graph

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 LASER to be model-agnostic, supporting both closed-domain STSG classification models and open-world vision-language models for converting input video data into relational database representations. With a unified formalization, an STSG generator, M_θ, parameterized by θ, takes in pixel-based raw video data X, and generate a distribution of STSGs. This distribution is then encoded as a predicted probabilistic relational database, $\hat{\mathbf{r}} = M_\theta(X)$.

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

Figure 6.4: The formal syntax of STSL, where a represents relational predicates, c represents constants, and v represents variables. Here, $\wedge$, and $\neg$ represents logical "and", "or", and "not". Formula may also contain temporal operators $\bigcirc$ (next), $\mathbf{U}$ (until), $\Box$ (global), and $\Diamond$ (finally).

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 LTL_f (De Giacomo and Vardi, 2013), which supports LTL reasoning over finite traces. Thus, we use LTL_f as a framework for specifying events and their temporal relationships.

Our STSL (Figure 6.4) further extends LTL_f by introducing relational predicates and variables. It starts from the specification ψ which existentially quantifies variables in an STSL formula. The formula φ is inductively defined, with basic elements as relational atoms α of the form $a(t_1, \ldots, t_n)$. Note that the terms $\bar{t} = \{t_1, \ldots, t_n\}$ can contain quantified variables to be later grounded into concrete entities based on context Γ, noted by $\text{subst}_\Gamma(\bar{t})$. From here, φ can be constructed using basic propositional logic components $\wedge$ (and), $\vee$ (or), and $\neg$ (not). The system additionally includes temporal unary operators $\Box$ (always), $\Diamond$ (finally), $\bigcirc$ (next), and a binary operator $\mathbf{U}$ (until) (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

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

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

```
1  // Term: either constant string or variable or wildcard
2  type Term = Const(String) | Var(Var) | Wildcard()
3
4  // TForm: temporal formula, containing `Global`, `Finally`,
5  // `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 conjunction
13 // and atomic queries
14 type LForm = And(TForm, TForm)
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 6.1: STSL defined in Scallop.

to "$\Diamond$(`touch(h, b)` $\wedge$ $\Diamond$`drop(b, _)`)". Here, the two events, `touch` and `drop`, need to happen in chronological order but are not required to be consecutive.

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

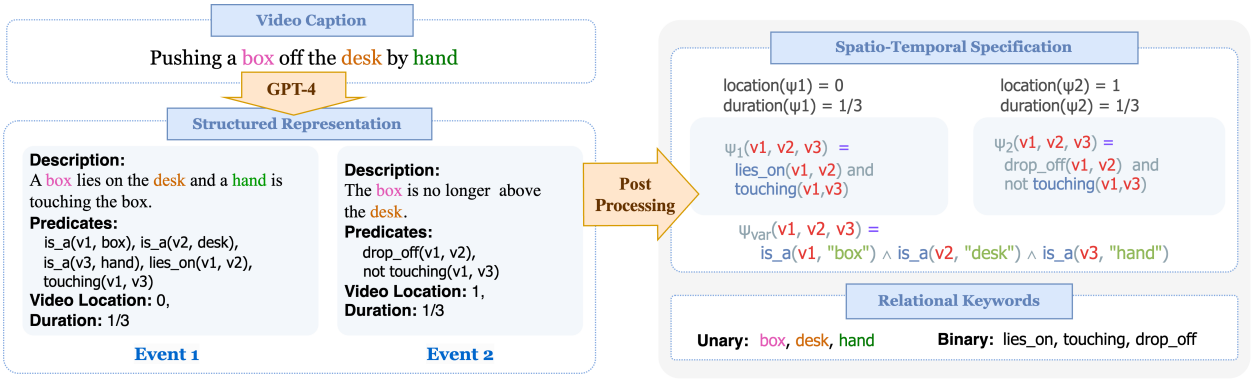


Figure 6.5: An illustration of our pipeline for natural language caption to programmatic spatio-temporal specification.

Natural Language to Spatio-Temporal Specification

To leverage the abundance of video captions as weak supervision signals, we employ a large language model (LLM) to automatically extract a programmatic specification ψ from each video caption c. Directly converting captions into a formal program is particularly challenging for an LLM, especially in a low-data language like STSL. We hence use a few-shot learning approach with an LLM to generate an intermediate structured representation of the caption in JSON format. For each caption c, our goal is to convert it into a series of events $\bar{e} = \{e_1, e_2, \ldots, e_n\}$. Each event includes (a) a detailed natural language description of the event, which guides the generation of subsequent details, (b) a series of unary, binary, positive, and negative predicates describing the semantics of the scenario, (c) the location of the event, $\text{loc}(e_i)$, in the video where the event occurs, represented as a fraction of the video length, and (d) the duration of the event, $\text{dur}(e_i)$, also expressed as a fraction.

To extract such structured representations from the caption, we designed a generic prompt template, which consists of the following components: (a) examples for temporal specification in fraction numbers: "0", "1/2", "2/3", "1". (b) scene graph keywords, such as object names and relations. (c) few-shot examples of caption and JSON structured representations pairs. We illustrate a caption and its structured representation in Figure 6.5.

The programmatic spatio-temporal specification is then generated by postprocessing the events in sequential order. Consequently, we can generate the programmatic spatio-temporal specification ψ

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

Figure 6.6: Formal semantics of STSL. $\langle w, s\rangle \models \psi$ means the STSL specification ψ is *aligned* with the ST-SG w starting from time s. We use $w \models \psi$ as an abbreviation for $\langle w, 1\rangle \models \psi$.

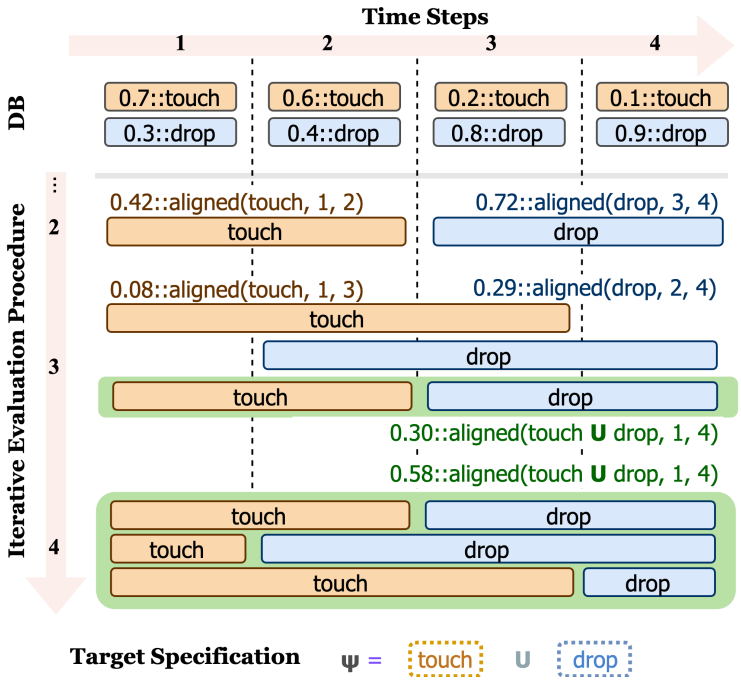


Figure 6.7: The evaluation process aligning a spatio-temporal scene graph (DB) with a specification `touch` **U** `drop`, that is, the drop event happens immediately after touch. This figure elides showing the arguments of the relational predicates and focuses only on matching sequential events.

for the caption as a sequence of events in chronological order:

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

Spatio-Temporal Alignment Checking

Given a probabilistic database $\mathbf{r}$ that encodes a distribution of STSGs (§6.3.1), and a specification ψ in STSL, we aim to measure the alignment score $\Pr(\mathbf{r} \models \psi)$ in an end-to-end and differentiable

manner. Conceptually, each probabilistic fact f in the database can be toggled on or off, resulting in $2^{|\mathbf{r}|}$ distinct *worlds*. Denoting each world (i.e. a concrete STSG) as $w \in \mathcal{P}(\mathbf{r})$ [3], we can check whether the world w satisfies the specification ψ or not. From here, the alignment score can be computed as the sum of the probabilities of worlds satisfying ψ: $\Pr(\mathbf{r} \models \psi) = \sum_{w \in \mathcal{P}(\mathbf{r}), w \models \psi} \Pr(w)$. Enumerating all possible worlds is intractable due to its exponential complexity. Since Scallop employ scalable algorithms, we can approximate this probability and greatly reduce the probabilistic reasoning time. We also note that some of the STSG w sampled from $\mathcal{P}(\mathbf{r})$ might be infeasible due to involving conflicting facts (e.g., a box is above and below a desk at the same time). To further enhance the logic deduction efficiency, we extend Scallop's "top-k proofs" provenance to support general disjunctive constraints and early removal of infeasible STSGs that do not satisfy the specification.

We implement the alignment checker with Scallop, as partially shown in Listing 6.3. This helps us to succinctly and precisely encode the formal semantics of STSL. It inductively computes the alignment between a temporal slice of $\mathbf{r}$ and an STSL formula. The whole specification ψ is aligned if the full $\mathbf{r}$ (from 1 to m) satisfies ψ with a concrete variable grounding Γ, which maps variables to concrete entities. We illustrate one simplified evaluation process in Figure 6.7. 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 (line 7-9 of Listing 6.2). Since in STSG we only deal with unary and binary relations, we hardcode the two relations `unary_atom` and `binary_atom`. One important difference between our alignment checker and the semantics of other DSLs shown before is that, alignment checker needs the *constraint solving* capability due to specifications having variables. Specifically, we need to solve each variable v to a concrete object o in the scene. This means that each variable can only be assigned to one object, forming a *mutual exclusion*. For this, we use an aggregator in Scallop, `disjunct`, that constructs the mutual exclusion in the tag space, as shown on line 5 of Listing 6.2.

We now present the Scallop code for the alignment checker for STSL (Listing 6.3). Starting from

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

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

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

line 1-5, we define the relation used to ground each term into concrete objects. Specifically, when the term is a variable (`Var`), we use the `var_obj` relation defined in Listing 6.2 to ground it into an object `o`. Note that `var_obj` has mutual exclusion within it, meaning that if two facts where a single variable is assigned to two objects present in a single proof, then the proof will be rejected by Scallop's provenance system. We continue to define the rules for aligning logical formula, which require the grounding of all terms appearing in the atoms. As for aligning temporal formula, we deal with temporal relations. For instance, on line 21, aligning `Global` formula at time step `s` means that the subformula `p1` needs to be aligned for all time steps between `s` and `n`. This is exactly what we define in the formal semantics of STSL (Figure 6.6).

Loss Functions

We now elaborate on the three loss functions used for training under the LASER framework.

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, ψ_i) and (X_j, ψ_j) be two datapoints in a mini-batch B, where ψ_i and ψ_j are the specifications for video X_i and X_j correspondingly. If $X_i \models \psi_j$, then it is an extra positive sample to the video X_i; otherwise, it is a

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

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

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

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

negative sample to X_i. We can thus define our per-batch contrastive loss $\mathcal{L}_c(B)$:

$$\mathcal{L}_c(B) = \frac{1}{|B|^2} \sum_{(X_i,\psi_i)\in B} \sum_{(X_j,\psi_j)\in B} \mathcal{L}(Pr(M_\theta(X_i) \models \psi_j), \mathbb{1}[\psi_i = \psi_j]) \tag{6.4}$$

Time-span supervision A video caption is expanded into a sequence of events using LLM, with each event assigned a specific temporal target, detailing its location and duration within the video, as illustrated in Section 6.3.1. By aligning the spatio-temporal specification ψ 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. 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 σ into the computation of probabilistic alignment between STSG w and an atomic specification $a(\bar{t})$, where we utilize the expected relative location $\text{loc}(a) \in [0, 1]$ within the video and the duration $\text{dur}(a) \in [0, 1]$ relevant to the length of the video:

$$\sigma(s, l, d) = \max(0, 1 - \frac{2|s - l|}{d}), \tag{6.5}$$

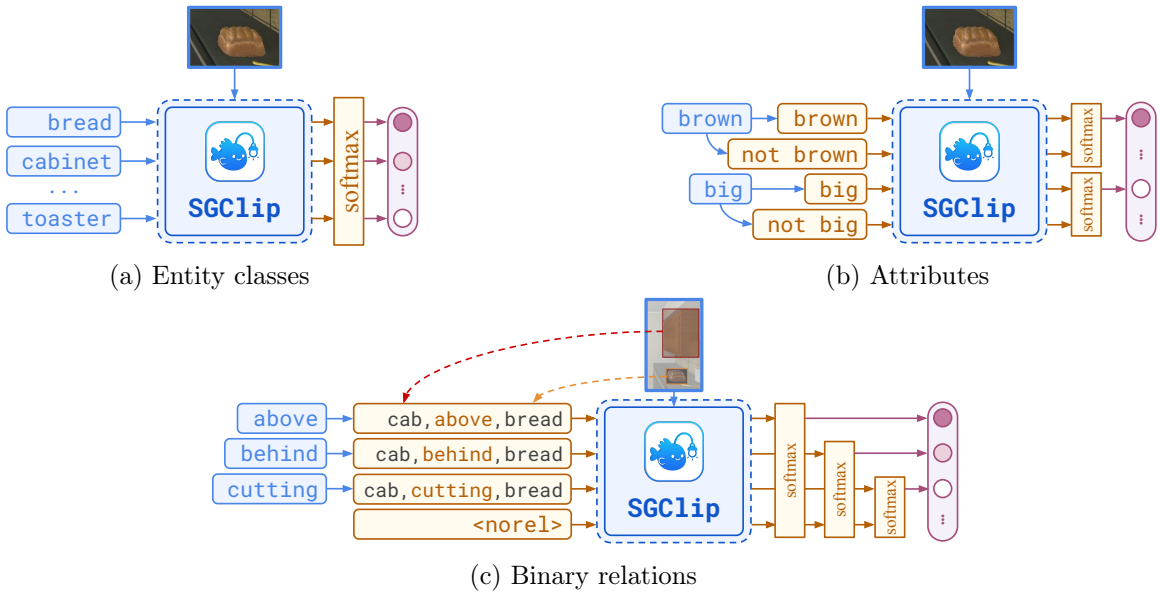
$$\Pr(\langle \Gamma, w, s \rangle \models a(\bar{t})) = \Pr(a(\bar{c}) \mid a(\bar{c}) \in w[s] \ \wedge \ \bar{c} = \text{subst}_\Gamma(\bar{t})) \cdot \sigma(s, \text{loc}(a), \text{dur}(a)). \tag{6.6}$$

Semantic loss To provide further supervision, we resort to human knowledge encoded in the form of integrity constraints. We introduce semantic loss (Xu et al., 2017, 2018) 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 per-sample semantic loss $\mathcal{L}_s(X_i)$ as an extra-weighted term after calculating the other loss components. Let n be the number of integrity constraints and let $\texttt{IC}_i$ be the i-th integrity constraint, we have

$$\mathcal{L}_s(X_i) = \sum_{i=1}^{n} \mathcal{L}(\Pr(M_\theta(X_i) \not\models \psi_{\texttt{IC}_i}), 0). \tag{6.7}$$

6.3.2 A Foundation Model for Scene Graph Generation

LASER enables the generation of spatio-temporal scene graphs in a generalized and open-domain setting. To fulfill this goal, we develop SGClip, a CLIP-based foundation model (Radford et al., 2021) for structured scene understanding. SGClip is designed to operate in an open-domain fashion, recognizing a wide and extensible set of concepts. Secondly, it must adapt to different types of concepts, while be capable of generalizing to unseen visual and textual domains. Finally, it must



(a) Entity classes

(b) Attributes

(c) Binary relations

Figure 6.8: Illustration of the inference modes of SGClip for three types of concepts: entity classes, attributes, and binary relations. While the model stays the same, the three inference modes perform different pre- and post-processing for a more accurate semantic estimation of probabilities.

produce probabilistic predictions to capture uncertainty.

To meet these goals, SGClip builds on CLIP's vision-language architecture, which naturally supports joint reasoning over images and textual phrases. However, deploying CLIP directly is insufficient, as it lacks specialization for structured scene graph prediction. To bridge this gap, we fine-tune SGClip to balance adaptability and generalizability, with the power of LASER. In this section, we describe 1) how to handle different types of concepts through inference time adaptation, 2) how to overcome the lack of data by collecting a model-driven self-supervision dataset, and 3) how to learn without relying on human annotations via a self-supervised neurosymbolic learning pipeline.

Model Architecture and Inference Time Adaptation

At its core, SGClip (Scene Graph CLIP) is a single CLIP-based model designed to score concept relevance within an image. Formally, it operates as $\text{SGClip}(\sigma, \bar{c}) \in \mathbb{R}^{|\bar{c}|}$, where σ is the input image and $\bar{c}$ is the a set of candidate concepts, producing a logit score for each concept that reflects the model's confidence in its presence. SGClip supports three distinct inference modes to handle different

types of concepts: entity classes $\bar{c}_{\text{class}}$, attributes $\bar{c}_{\text{attr}}$, and binary relations $\bar{c}_{\text{rela}}$. Illustrated in Figure 6.8, each mode requires a specialized formulation of the input concepts and scoring process, effectively allowing SGClip to operate flexibly across these concept types:

Entity classes In this setting, SGClip is used to identify the most likely entity class presented in an image segment. Let $\bar{c}_{\text{class}}$ denote the list of candidate entity classes. Since an entity is typically assumed to belong to a single class, we apply softmax normalization over the logit scores produced by SGClip for these candidates: $\text{softmax}(\text{SGClip}(\sigma, \bar{c}_{\text{class}}))$.

Attributes To estimate the likelihood that a specific segment σ possesses a particular attribute c, we construct a binary contrast between the attribute and its negation by evaluating $\text{softmax}(\text{SGClip}(\sigma, \{c, \neg c\}))$, where $\neg c$ denotes the negated textual phrase (e.g., "not red" for the attribute "red"). The first element of the resulting probability distribution corresponds to the model's estimated likelihood. To improve computational efficiency, we perform batched evaluation by merging all attribute-contradiction pairs into a single concept set $\bar{c}^*_{\text{attr}} = \bar{c}_{\text{attr}} \cup \{\neg c \mid c \in \bar{c}_{\text{attr}}\}$.

Binary relations For binary relation prediction, the goal is to determine whether a relation c holds between two segments σ_i and σ_j. To do this, we first compute a bounding region σ^*_{ij} that tightly encloses both segments. Within this region, we apply distinct color tinting to σ_i and σ_j to indicate their directional roles (subject and object). To provide additional relational context, especially for interactions like "`cutting`," we augment the relation phrase by including the predicted entity classes of the subject and object, generating "(`robot`, `cutting`, `cabbage`)". Relation predictions are thus conditioned on the classes of both the subject and the object. Specifically, for each segment σ_i, we compute its most likely class ν_i by selecting the top prediction from the entity class:

$$\nu_i = c_{\text{class}\, u}, \quad \text{where } u = \text{argmax}_{u \in 1 \ldots |\bar{c}_{\text{class}}|} \text{SGClip}(\sigma_i, \bar{c}_{\text{class}})_u.$$

We then form the augmented relation phrase as (ν_i, c, ν_j). Similar to attribute prediction, we contrast the candidate relation with a special token `<norel>` denoting "no relation," and compute $\text{softmax}(\text{SGClip}(\sigma^*_{ij}, \{(\nu_i, c, \nu_j), \texttt{<norel>}\}))$, to obtain the probability of whether the relation c holds

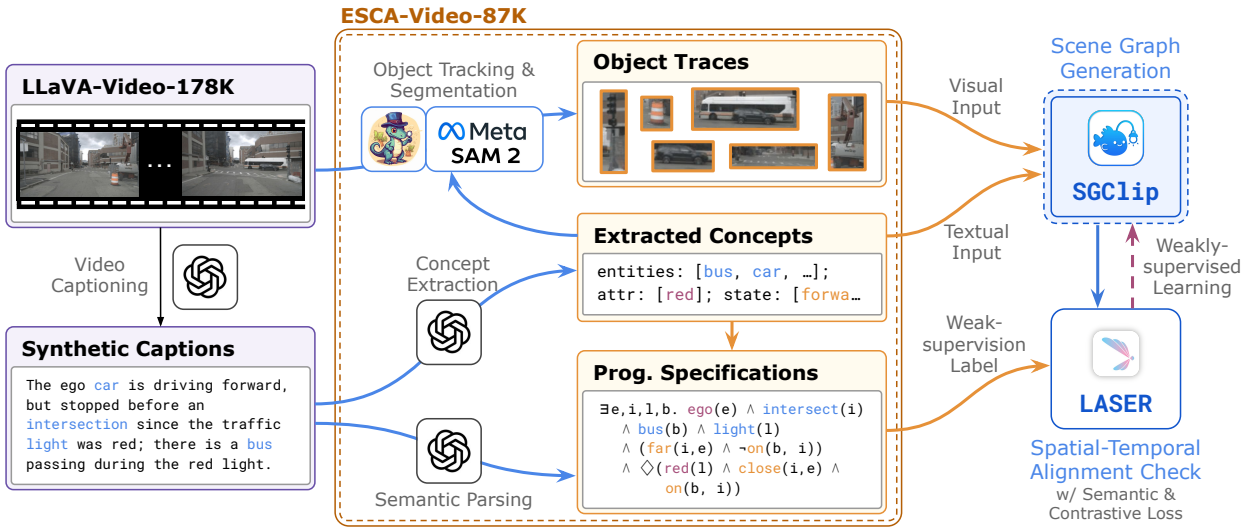


Figure 6.9: Illustration of the construction of ESCA-Video-87K dataset and the model-driven self-supervised fine-tuning pipeline of our SGClip model. In addition to videos and their natural language captions, ESCA-Video-87K includes object traces, open-domain concepts, and programmatic specifications for 87K video-caption pairs. The dataset is then used to train SGClip via LASER (Huang et al., 2025), a neurosymbolic learning procedure based on spatio-temporal alignment.

between object i and j. In practice, this process is batched over all segment pairs and relation concepts to maximize efficiency: $\bar{c}^*_{\text{rela}} = \{(\nu_i, c, \nu_j) \mid i, j \in 1 \ldots |\bar{\sigma}|, c \in \bar{c}_{\text{rela}}\} \cup \{\texttt{<norel>}\}$.

ESCA-Video-87K Dataset

We adopt the neurosymbolic weak-supervision pipeline introduced in Section 6.3.1 (Huang et al., 2025), which enables learning fine-grained STSGs from *weak supervision signals* derived from spatio-temporal programmatic specifications, eliminating the need for costly manual annotations. While the details of the adapted learning pipeline for SGClip are provided in Section 6.3.2, we begin by introducing the ESCA-Video-87K dataset, the dataset we curate and use to train SGClip.

The ESCA-Video-87K dataset is constructed on top of the publicly available LLaVA-Video-178K dataset (Zhang et al., 2024), and consists of 87K short video clips, each paired with natural language captions generated by GPT-4 (Hurst et al., 2024). As illustrated in Figure 6.9, these captions are first processed to extract relevant concepts which are then fed into GD (Liu et al., 2024) and SAM2 (Ravi et al., 2024) to obtain object traces, which are sequences of object segmentations that

evolve across multiple video frames. In addition, these concepts are also used to assist in generating STSL specifications, which are constructed via the semantic parsing pipeline detailed in Section 6.3.1. These specifications formally describe how the semantics of object traces evolve, capturing temporal relations using operators such as "until", "finally", or "always".

In summary, each data point in ESCA-Video-87K is represented as a 5-tuple $(\bar{I}, L_{\text{cap}}, \Sigma, \bar{c}, \phi)$, where $\bar{I} = \{I_1, I_2, \dots\}$ is the video with I_i as the image of the i-th frame, L_{cap} is the associated natural language caption, $\Sigma = \{\bar{\sigma}_1, \bar{\sigma}_2, \dots\}$ is the set of object traces, $\bar{c} = \{c_1, c_2, \dots\}$ is the set of extracted concepts, and ϕ is the spatio-temporal programmatic specification. This rich, multi-level annotation enables training models like SGClip without requiring manual scene graph labeling.

Neurosymbolic Learning Pipeline

Given the ESCA-Video-87K dataset, our goal is to fine-tune the SGClip model using the provided object traces, concepts, and spatio-temporal programmatic specifications. This is achieved by aligning the scene graphs generated by SGClip with the expected specifications (Huang et al., 2025), where the degree of alignment serves as the learning signal (Figure 6.9). At a high-level, we use the Scallop-based neurosymbolic learning pipeline, LASER, described in Section 6.3.1 to fine-tune the model. Specifically, the alignment loss computation mirrors the inference-time adaptation procedure described in Section 6.3.2, where different types of concepts are processed differently but unified under the same model. The pipeline is further enhanced with *semantic losses*, derived from evaluating common-sense and temporal constraint satisfaction, as well as a *contrastive loss* that encourages the model to distinguish between matched and unmatched scene graph-specification pairs.

6.4 Empirical Evaluation

Our evaluation attempts to answer several key questions about our method.

RQ1: How effective is LASER in learning STSG generators?

RQ2: How does its performance compare to fully supervised baselines and existing methods?

RQ3: Is STSL versatile and expressive enough when applied to diverse specification patterns?

RQ4: How generalizable and adaptable is SGClip when evaluated independently on open-domain, zero-shot, and downstream transfer tasks?

To address these issues, we evaluate LASER on three datasets: OpenPVSG (Yang et al., 2023b), a realistic dataset with diverse and fine-grained STSG annotations, and 20BN (Goyal et al., 2017), a video dataset focusing on daily actions. These datasets vary significantly in their temporal patterns, showcasing the versatility of STSL. Specifically, OpenPVSG captions focus on natural and complex events, while 20BN captions are in the form of action pre-conditions and post-conditions. Further, we evaluate our trained SGClip model on its zero-shot generalizability on open-domain video scene graph datasets (OpenPVSG (Yang et al., 2023b), Action Genome (Ji et al., 2020), and VidVRD (Shang et al., 2017)), as well as its down-stream fine-tunability on ActivityNet (Yu et al., 2019).

Our main result shows significant improvements of LASER over fully supervised baselines. On OpenPVSG, LASER achieves a unary predicate prediction accuracy of 27.78% and a binary recall@5 of 0.42, surpassing the best fully supervised baseline by 12.65% and 0.22, respectively. Furthermore, LASER outperforms baselines by 7% on 20BN. We now delve into the experiments conducted on each of these datasets.

6.4.1 OpenPVSG Dataset

The OpenPVSG (Yang et al., 2023b) 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, 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.

As illustrated in Figure 6.3, 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

Method (with LASER)		**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
	Fine-tuned	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
	Fine-tuned	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
	Fine-tuned	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

Table 6.1: 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.

between objects. We leverage Scallop 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.

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 probabilistic STSG. In particular, unary R@k assesses object category prediction capability, while binary R@k evaluates pair-wise prediction of binary relations.

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 6.1, Scallop significantly enhances backbone performance using only captions for weak supervision on the OpenPVSG dataset.

To further assess the data efficiency of LASER, we train the model on 10% and 50% of the training dataset. As illustrated in Figure 6.10, even with just 10% of the training data (183 video-caption

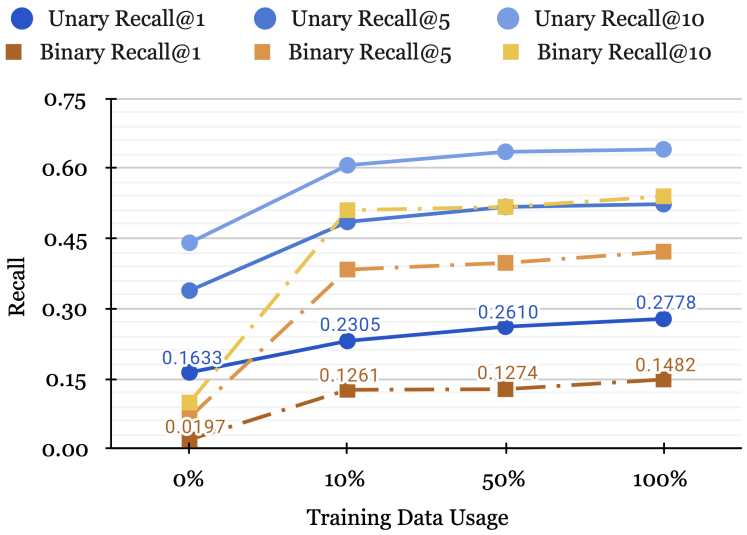


Figure 6.10: Data-efficient fine-tuning on OpenPVSG dataset with Scallop: Providing only 10%, 50%, and 100% of the training dataset significantly enhances the performance of CLIP model.

pairs), Scallop 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 LASER's data efficiency.

6.4.2 20BN Dataset

The 20BN dataset consists of (a) video and action pairs of humans performing everyday actions with ordinary objects (Goyal et al., 2017), (b) expert designed pre-conditions and post-conditions for the actions in the PDDL language (Migimatsu and Bohg, 2022), and (c) frame-based object bounding boxes (Materzynska et al., 2020). There are 172 actions with 37 underlying predicates in this dataset, capturing object attributes, object states, and relationships between two objects. Specifically, there are 6 static predicates, 21 unary predicates, and 10 binary predicates. We train on $10,000$ training datapoints and test on $14,816$ data points.

In the 20BN dataset, each video is assigned an action label from a set of 172 possible actions. Each action is annotated with a natural language description and a logical specification, represented as a pair of pre-condition and post-condition in PDDL format. These PDDL specifications naturally map to the $\Diamond(\psi_{\text{pre}} \wedge \Diamond\psi_{\text{post}})$ structure in STSL. This setup enables us to assess the performance

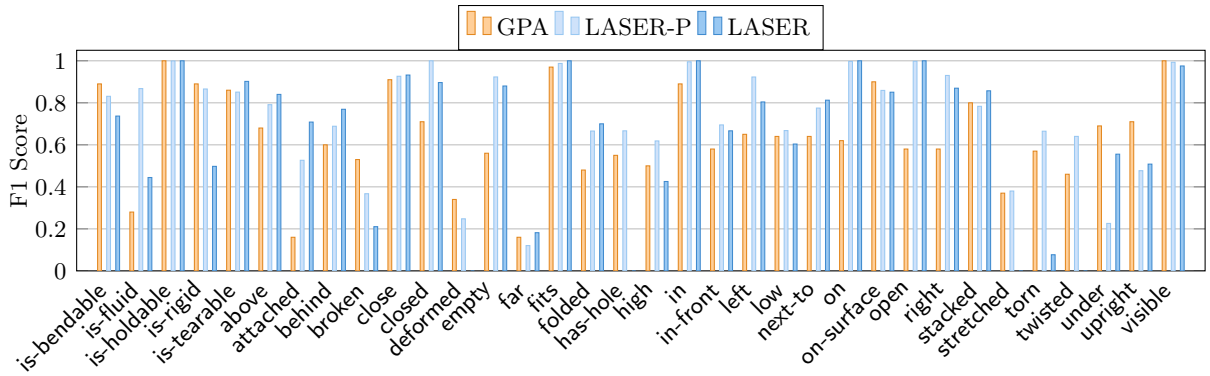


Figure 6.11: Per-predicate F1 score performance comparison of LASER, LASER-P, and GPA, all trained on the full 20BN dataset. LASER-P outperforms GPA on 71% of predicates, and LASER outperforms GPA on 59% of the predicates.

difference between caption-generated specifications and ground truth programmatic specifications. Scallop received the generated specifications from natural language captions using GPT-4 as weak supervision label. For ablation study, we refer to a variant of LASER that uses ground truth specifications as weak supervision labels as LASER-P.

We consider exact accuracy and F1-score as the metrics to evaluate STSG generators for 20BN dataset. Specifically, we compute F1-score and accuracy for each predicate in the vocabulary, and evaluate the weighted average of F1-score and accuracy for overall evaluation. Our backbone STSG generator model comprises an S3D (Xie et al., 2018) video encoding model pre-trained on Kinetics 400 (Kay et al., 2017), followed by ROIpooler for extract object embeddings, then passed to MLP layers for relation classification.

We compare our approach against GPA (Migimatsu and Bohg, 2022), a weakly supervised baseline which uses the ground truth specification as learning signals. Our evaluation shows that LASER-P, with the same programmatic supervision, achieves a higher average F1-score of 0.77 and accuracy of 91%, compared to the baseline's 0.74 F1-score and 76% accuracy. Scallop, using only natural language descriptions, achieves a comparable F1-score of 0.73 and better accuracy of 83%. Furthermore, LASER-P outperforms the baseline in 71% of the fine-grained predicate recognition tasks, and LASER outperforms 59%, as shown in Figure 6.11. For the qualitative study, we present the STSG generator's frame-wise predictions on several test data points in Figure 6.12.

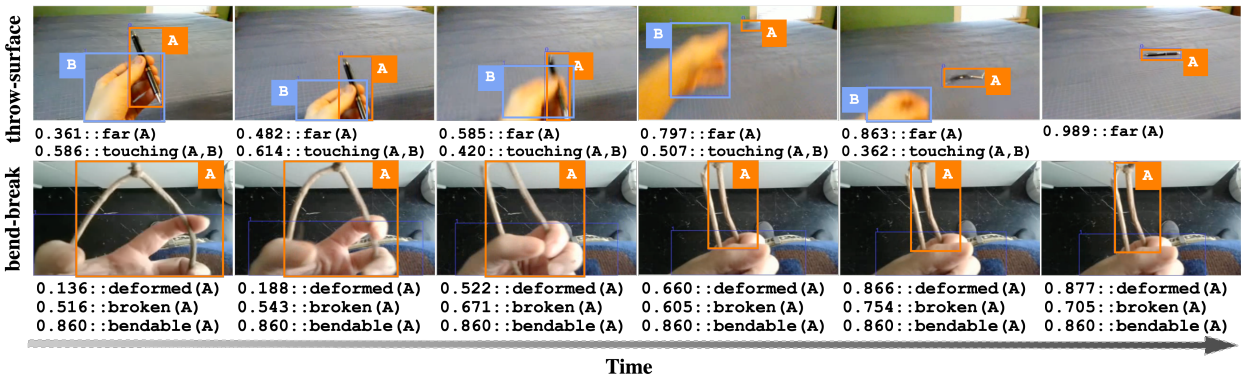
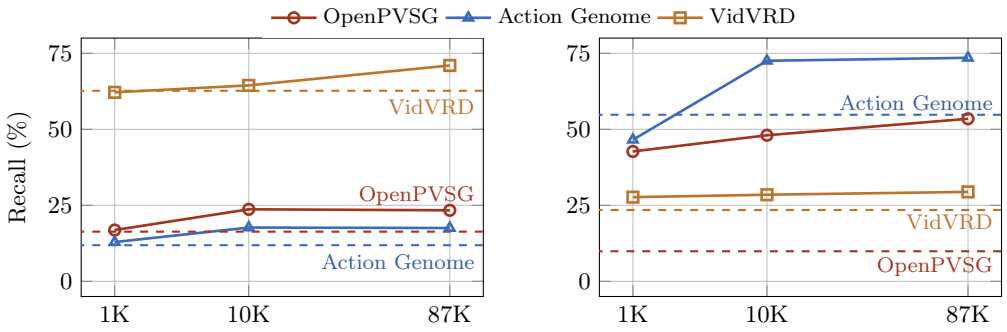


Figure 6.12: Qualitative study of the model trained with Scallop on the full 20BN dataset. Each row displays a sequence of frames from a video, with bounding boxes labeled by object IDs. The left side of each row shows the action label, while the bottom of each row lists the attributes and relationships associated with the objects, along with the corresponding likelihoods of these facts holding true.

6.4.3 SGClip Evaluation

Evaluation setup For evaluating SGClip independently, we consider out-of-domain scene graph benchmarks, including OpenPVSG (Yang et al., 2023b), Action Genome (Ji et al., 2020), and VidVRD (Shang et al., 2017), comparing SGClip against strong baselines such as CLIP (Radford et al., 2021), InternVL-6B (Chen et al., 2024), BIKE (Wu et al., 2023b), and Text4Vis (Wu et al., 2023a). To assess SGClip's downstream adaptability beyond structured scene graph prediction, we further test the fine-tunability on the ActivityNet action recognition dataset (Yu et al., 2019) by applying a transfer protocol.

Results Evaluating SGClip's zero-shot generalization, Figure 6.13 shows that SGClip trained on ESCA-Video-87K consistently outperforms CLIP on OpenPVSG, Action Genome, and VidVRD, demonstrating strong out-of-domain robustness. Further, SGClip shows strong adaptability, achieving notable improvements when fine-tuned on VidVRD. Beyond scene graph tasks, Figure 6.14 highlights SGClip's downstream transferability to action recognition on ActivityNet. Fine-tuned with only 1% of the training data, SGClip outperforms state-of-the-art zero-shot video recognition baselines. With 5% of the data (approximately 800 videos), SGClip achieves 92.10% accuracy, approaching the performance of InternVideo2-6B with end-to-end finetuning on the ActivityNet dataset.



(a) Entity class prediction R@1 (b) Binary relation prediction R@10

Figure 6.13: The zero-shot performance of SGClip compared to CLIP (shown in dashed lines) on OpenPVSG, Action Genome, and VidVRD datasets. We showcase the Recall@1 metrics on entity class prediction, as well as the Recall@10 metrics on binary relation prediction. To illustrate data-efficiency, we include the performance of checkpoints of SGClip when trained on 1K, 10K, or 87K (full) portion of ESCA-Video-87K.

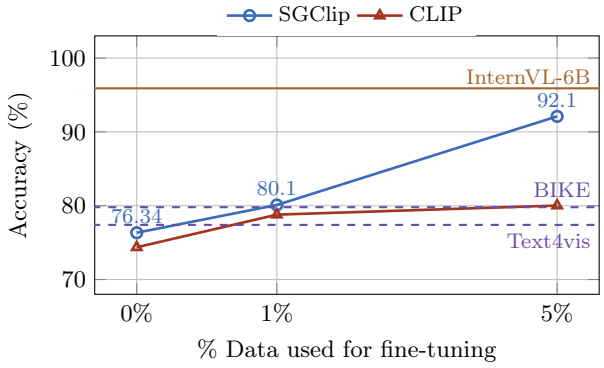


Figure 6.14: Down-stream fine-tunability on action recognition, evaluated on ActivityNet dataset. We also illustrate zero-shot baselines (BIKE and Text4vis) as well as a fully-supervised baseline (InternVL-6B).

6.5 Related Work

Structured representation of image/video semantics Significant advances have been made in representing structured information within vision data. A widely adopted representation for capturing spatial semantics in images is *Scene Graph* (Kuznetsova et al., 2018; Lu et al., 2016), with various generation techniques emerging over the years (Zhu et al., 2022; Liu et al., 2021; Huang et al., 2020, 2021; Yang et al., 2018a; Li et al., 2024c). More recently, research has increasingly focused on extending these representations to integrate both spatial and temporal semantics in videos (Yang et al., 2023b; Li et al., 2022b). However, learning spatio-temporal structures remains a

challenging open problem, which LASER addresses by introducing a neurosymbolic approach.

Video scene graph learning Learning video scene graphs has attracted significant attention from the vision community. Various tasks, including entity tracking, object identification, dynamic relation analysis, and pathfinding, are being explored (Sun et al., 2023; Xu et al., 2022a; Shang et al., 2017). New techniques involving spatio-temporal aware networks have been developed (Nag et al., 2023; Cong et al., 2021; Ji et al., 2021; Sun et al., 2019). A few recent works have developed point-solutions for extracting fine-grained video semantics (Lee et al., 2023; Apriceno et al., 2022; Chang et al., 2019). Such approaches includes both dynamic time warping (Dvornik et al., 2021; Chang et al., 2019; Richard et al., 2018; Ding and Xu, 2018), soft nearest neighbor (Han et al., 2022; Dwibedi et al., 2019), and semantic loss (Xu et al., 2022b, 2017, 2018). To our knowledge, LASER *is the first framework to train STSG generation models using video captions as weak-supervisory labels.*

Vision language pre-training Vision language pre-training is crucial in video understanding and has a wide range of downstream applications. Current works have succeeded in learning visual representations using large-scale paired visual-textual data through contrastive learning in both image-text (Zhai et al., 2023; Radford et al., 2021; Jia et al., 2021) and video-text (Li et al., 2021a; Xu et al., 2021; Miech et al., 2019) representation learning. Recent works also explore the viability of utilizing pre-trained foundation models for generating image and video scene graphs (Shindo et al., 2024; Liang et al., 2024; Zhang et al., 2023; Yao et al., 2021).

CHAPTER 7

APPLICATION: SECURITY VULNERABILITY DETECTION

Security vulnerabilities pose a major threat to the safety of software applications and its users. In 2023 alone, more than 29,000 CVEs were reported—almost 4000 higher than in 2022. Detecting vulnerabilities is extremely challenging despite advances in techniques to uncover them. A promising such technique called static taint analysis is widely used in popular tools such as GitHub CodeQL (Avgustinov et al., 2016), Facebook Infer (FB Infer), Checker Framework (Checker Framework), and Snyk Code (Snyk.io). These tools, however, face several challenges that greatly limit their effectiveness and accessibility in practice.

False negatives due to missing taint specifications First, static taint analysis predominantly relies on *specifications* of third-party library APIs as sources, sinks, or sanitizers. In practice, developers and analysis engineers have to manually craft such specifications based on their domain knowledge and API documentation. This is a laborious and error-prone process that often leads to missing specifications and incomplete analysis of vulnerabilities. Further, even if such specifications may exist for many libraries, they need to be periodically updated to capture changes in newer versions of such libraries and also cover new libraries that are developed.

False positives due to lack of context-sensitive reasoning Second, it is well-known that static analysis often suffers from low precision, i.e., it may generate many false alarms (Kang et al., 2022; Johnson et al., 2013). Such imprecision stems from multiple sources. For instance, the source or sink specifications may be spurious, or the analysis may over-approximate over branches in code or possible inputs. Further, even if the specifications are correct, the context in which the detected source or sink is used may not be exploitable. Hence, a developer may need to triage through several potentially false security alerts, wasting significant time and effort.

Limitations of prior data-driven approaches Many techniques have been proposed to address the challenges of static taint analysis. For instance, Livshits et al. (2009) proposed a prob-

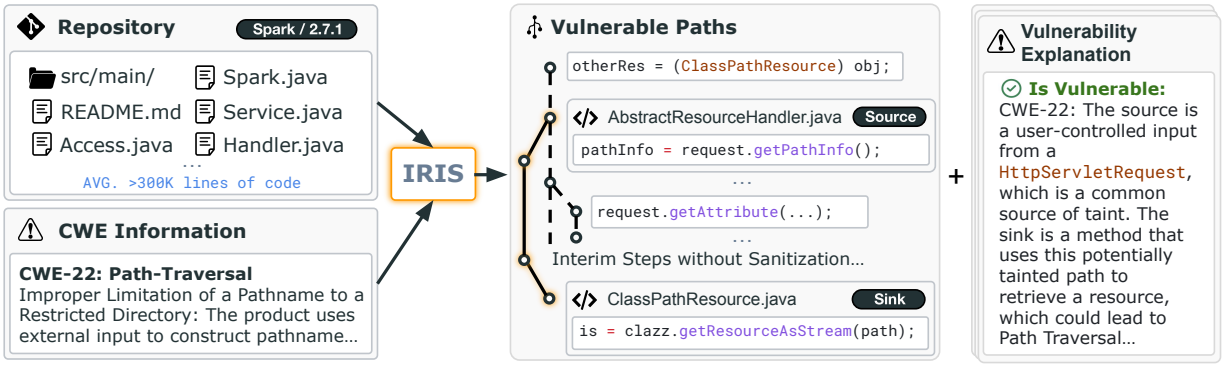


Figure 7.1: Overview of the IRIS neurosymbolic system. It checks a given whole repository for a given type of vulnerability (CWE) and outputs a set of potential vulnerable paths with explanations.

abilistic approach, MERLIN, to automatically mine taint specifications. A more recent work, Seldon (Chibotaru et al., 2019), improves the scalability of this approach by formulating the taint specification inference problem as a linear optimization task. However, such approaches rely on analyzing the code of third-party libraries to extract specifications, which is expensive and hard to scale. Researchers have also developed statistical and learning-based techniques to mitigate false positive alerts (Jung et al., 2005; Heckman and Williams, 2009; Hanam et al., 2014; Li et al., 2021d). However, such approaches still have limited effectiveness in practice (Kang et al., 2022).

Large Language Models have made impressive strides in code generation and summarization. LLMs have also been applied to code related tasks such as program repair (Xia et al., 2023), code translation (Pan et al., 2024), test generation (Lemieux et al., 2023), and static analysis (Li et al., 2024a). Recent studies (Steenhoek et al., 2024; Khare et al., 2023) evaluated LLMs' effectiveness at detecting vulnerabilities at the method level and showed that LLMs fail to do complex reasoning with code, especially because it depends on the *context* in which the method is used in the project. On the other hand, recent benchmarks like SWE-Bench (Jimenez et al., 2023) show that LLMs are also poor at doing project-level reasoning. Hence, an intriguing question is whether LLMs can be combined with static analysis to improve their reasoning capabilities.

Approach We propose **IRIS-Scallop**, a neurosymbolic approach for vulnerability detection that combines the strengths of static analysis and LLMs. Figure 7.1 presents an overview of IRIS-Scallop.

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

In this chapter, we present IRIS-Scallop, a neurosymbolic framework that combines large language models with static analysis for software vulnerability detection. At its core is a Scallop program that unifies symbolic reasoning (reachability analysis) with learned components (taint specification inference). We begin with a motivating example (Section 7.1), followed by a problem definition (Section 7.2) and a detailed description of the framework (Section 7.3). Section 7.4 presents experimental results, including CWE-Bench-Java, a new dataset for evaluating whole-repository vulnerability detection (Section 7.4.1). We conclude with related work (Section 7.5).

7.1 Illustrative Overview

We illustrate the effectiveness of IRIS-Scallop in detecting a previously known code-injection (CWE-094) vulnerability in cron-utils (ver. 9.1.5), a Java library for Cron data manipulation. Figure 7.2 shows the relevant code snippets. A user-controlled string `value` passed into `isValid` function is transferred without sanitization to the `parse` function. If an exception is thrown, the function constructs an error message with the input. However, the error message is used to invoke method `buildConstraintViolationWithTemplate` of class `ConstraintValidatorContext` in `javax.validator`, which interprets the message string as a Java Expression Language (Java EL) expression. A malicious user may exploit this vulnerability by crafting a string containing a shell

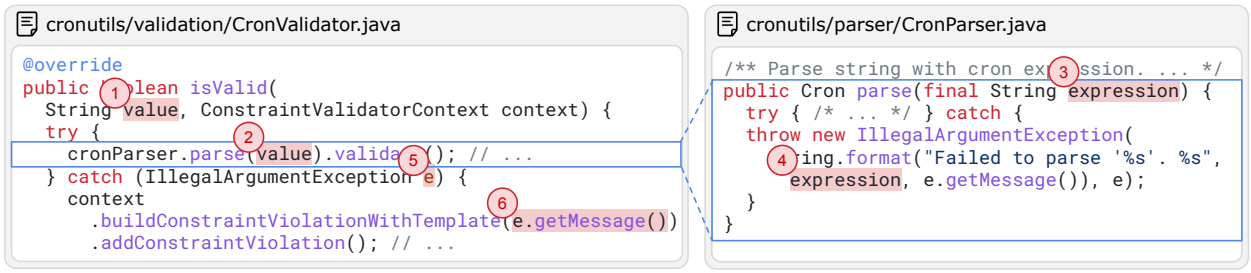


Figure 7.2: An example of Code Injection (CWE-94) vulnerability found in cron-utils (CVE-2021-41269) that CodeQL fails to detect. We number the program points of the vulnerable path.

command such as `Runtime.exec('rm -rf /')` to delete critical files on the server.

Detecting this vulnerability poses several challenges. First, the cron-utils library consists of 13K SLOC (lines of code excluding blanks and comments), which needs to be analyzed to find this vulnerability. This process requires analyzing data and control flow across several internal methods and third-party APIs. Second, the analysis needs to identify relevant *sources* and *sinks*. In this case, the `value` parameter of the public `isValid` method may contain arbitrary strings when invoked, and hence may be a source of malicious data. Additionally, external APIs like `buildConstraintViolationWithTemplate` can execute arbitrary Java EL expressions, hence they should be treated as sinks that are vulnerable to Code Injection attacks. Finally, the analysis also requires identifying any sanitizers that block the flow of untrusted data.

Modern static analysis tools, like CodeQL, are effective at tracing taint data flows across complex codebases. However, CodeQL fails to detect this vulnerability due to missing specifications. CodeQL includes many manually curated specifications for sources and sinks across more than 360 popular Java library modules. However, manually obtaining such specifications requires significant human effort to analyze, specify, and validate. Further, even with perfect specifications, CodeQL may often generate numerous false positives due to a lack of contextual reasoning, increasing the developer's burden of triaging the results.

In contrast, IRIS-Scallop takes a different approach by inferring project- and vulnerability-specific specifications *on-the-fly* by using LLMs. The LLM-based components in IRIS-Scallop correctly

identify the untrusted source and the vulnerable sink. IRIS-Scallop augments CodeQL with these specifications and successfully detects the unsanitized dataflow path between the detected source and sink in the repository.

However, augmented CodeQL produces many false positives, which are hard to eliminate using logical rules. To solve this challenge, IRIS-Scallop encodes the detected code paths and the surrounding context into a simple prompt and uses an LLM to classify it as true or false positive. Specifically, out of 8 paths reported by static analysis, 5 false positives are filtered out, leaving the path in Figure 7.2 as one of the final alarms. Overall, we observe that IRIS-Scallop can detect many such vulnerabilities that are beyond the reach of CodeQL-like static analysis tools, while keeping false alarms to a minimum.

7.2 Problem Definition

We formally define the static taint analysis problem for vulnerability detection. Given a project P, taint analysis extracts an inter-procedural data flow graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of nodes representing program expressions and statements, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the set of edges representing data or control flow edges between the nodes. A vulnerability detection task comes with two sets $\boldsymbol{V}^C_{source} \subseteq \mathbb{V}$, $\boldsymbol{V}^C_{sink} \subseteq \mathbb{V}$ that denote source nodes where tainted data may originate and sink nodes where a security vulnerability can occur if tainted data reaches it, respectively. Naturally, different classes C of vulnerabilities (or CWEs) have different source and sink specifications. Additionally, there can be sanitizer specifications, $\boldsymbol{V}^C_{sanitizer} \in \mathbb{V}$, that block the flow of tainted data (such as escaping special characters in strings).

The goal of taint analysis is to find pairs of sources and sinks, $(V_s \in \boldsymbol{V}^C_{source}, V_t \in \boldsymbol{V}^C_{sink})$, such that there is an *unsanitized* path from the source to the sink. More formally, $\mathit{Unsanitized\_Paths}(V_s, V_t) = \exists\ \mathit{Path}(V_s, V_t)$ s.t. $\forall V_n \in \mathit{Path}(V_s, V_t), V_n \notin \boldsymbol{V}^C_{sanitizer}$. Here, $\mathit{Path}(V_1, V_k)$ denotes a sequence of nodes $(V_1, V_2, \ldots, V_k)$, such that $V_i \in \mathbb{V}$ and $\forall i \in 1\ \mathit{to}\ k-1 : (v_i, v_{i+1}) \in \mathbb{E}$.

Two key challenges in taint analysis include: 1) identifying relevant taint specifications for each class

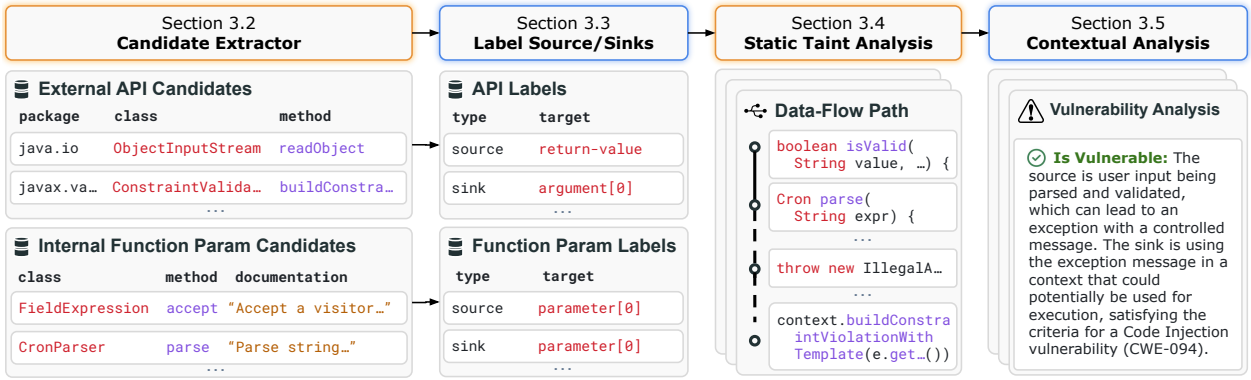


Figure 7.3: An illustration of the IRIS-Scallop pipeline.

C that can be mapped to $\boldsymbol{V}^{C}_{source}$, $\boldsymbol{V}^{C}_{sink}$ for any project P, and 2) effectively eliminating false positive paths in $Unsanitized\_Paths(V_s, V_t)$ identified by taint analysis. In the following sections, we discuss how we address each challenge by leveraging LLMs.

7.3 Neurosymbolic Solution with Scallop

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

As illustrated in Figure 7.3, IRIS-Scallop has four main stages: First, IRIS-Scallop extracts all taint specification candidates, including invoked external APIs and internal function parameters. Second, IRIS-Scallop queries an LLM to label these APIs as sources or sinks that are specific to the given vulnerability class C. Third, IRIS-Scallop transforms the labeled sources and sinks into specifications that can be fed into a static analysis engine, such as CodeQL, and runs a vulnerability class-specific taint analysis query to detect vulnerabilities of that class in the project. This step generates a set of vulnerable code paths (or alerts) in the project. Finally, IRIS-Scallop triages the generated alerts by automatically filtering false positives, and presents them to the developer.

```
import "codeql.scl"

// The declaration of relation storing function parameter candidates
type func_param_candidate(
    package: String,
    clazz: String,
    method: String,
    javadoc: String,
    param_index: i32,
    param_name: String,
    param_type: String,
)

// The query getting the candidates
rel func_param_candidate(p, c, m, doc, i, param_name, param_type) =
    callable(callable) and
    (is_public(callable) or is_override(callable)) and
    get_declaring_type(callable, decl_ty) and

    // Get basic callable information
    get_package(decl_ty, package) and
    get_name(decl_ty, clazz) and
    get_name(callable, m) and
    get_javadoc_string(callable, doc) and

    // Iterate through parameters and get parameter information
    get_number_of_parameters(callable, num_params) and
    i in 0..num_params and
    get_parameter(callable, i, param) and
    get_name(param, param_name) and
    get_type(param, ty_id) and get_name(ty_id, param_type)
```

Listing 7.1: Getting candidates in Scallop via CodeQL database foreign predicates.

7.3.1 Candidate Source and Sink Extraction

A project may use various third-party APIs whose specifications may be unknown—reducing the effectiveness of taint analysis. In addition, internal APIs might accept untrusted input from downstream libraries. Hence, our goal is to automatically infer specifications for such APIs. We define a specification S^C as a 3-tuple $\langle T, F, R \rangle$, where $T \in \{\mathit{ReturnValue}, \mathit{Argument}, \mathit{Parameter}, \dots\}$ is the type of node to match in $\mathbb{G}$, F is an N-tuple of strings describing the package, class,

method name, signature, and argument/parameter position (if applicable) of an API, and $R \in \{\textit{Source}, \textit{Sink}, \textit{Taint-Propagator}, \textit{Sanitizer}\}$ is the role of the API. For example, the specification $\langle \textit{Argument}, (\texttt{java.lang}, \texttt{Runtime}, \texttt{exec}, (\texttt{String[]}), 0), \textit{Sink} \rangle$ denotes that the first argument of `exec` method of `Runtime` class is a sink for a vulnerability class (OS command injection). A static analysis tool maps these specifications to sets of nodes $\boldsymbol{V}^C_{source}$ or $\boldsymbol{V}^C_{sink}$ in $\mathbb{G}$.

To identify taint specifications $\boldsymbol{S}^C_{source}$ and $\boldsymbol{S}^C_{sink}$, we first extract $\boldsymbol{S}^{\text{ext}}$: external library APIs that are invoked in the given Java project and are potential candidates to be taint sources or sinks. We also extract $\boldsymbol{S}^{\text{int}}$, internal library APIs that are public and may be invoked by a downstream library.

To extract candidates and their associated metadata, we use the **Scallop-CodeQL** plugin, which provides a set of foreign predicates that allow Scallop programs to directly query CodeQL databases. In Listing 7.1, we show a Scallop rule that extracts function parameters as potential taint source candidates. Each candidate is represented by its Java package, class (noted as clazz to avoid conflict with the keyword class), method name, and parameter index. We also extract auxiliary metadata such as the parameter's name, type, and JavaDoc string to support further processing and potential LLM-assisted inference. The rule (lines 15-31) utilizes several foreign predicates—such as `callable`, which retrieves all Callable entities (i.e., functions or methods) in the project, and `is_public`, which checks whether a given callable is publicly accessible. These foreign predicates internally index into the CodeQL database to retrieve the relevant information, enabling seamless integration of static code facts into the neurosymbolic Scallop reasoning pipeline.

7.3.2 Inferring Taint Specifications using LLMs

We develop an automated specification inference technique: $\textit{LabelSpecs}(\boldsymbol{S}^{\#}, \texttt{LLM}, C, R) = \boldsymbol{S}^C_R$, where $\boldsymbol{S}^{\#} = \boldsymbol{S}^{\text{ext}} \cup \boldsymbol{S}^{\text{int}}$ are candidate specifications for sources and sinks. In this work, we do not consider sanitizer specifications, because they typically do not vary for the vulnerability classes that we consider. We use LLMs to infer taint specifications. Specifically, external APIs in $\boldsymbol{S}^{\text{ext}}$ can be classified as either source or sink, while internal APIs in $\boldsymbol{S}^{\text{int}}$ can have their formal parameters identified as sources. Notably for internal APIs, we also include information from repository readme

```
@gpt(
    system="""
    You are a security expert. You are given a list of APIs
    implemented in established Java libraries, and you need to
    identify whether some of these APIs could be potentially
    invoked by downstream libraries with malicious end-user
    (not programmer) inputs. For instance, functions that
    deserialize or parse inputs might be used by downstream
    libraries and would need to add sanitization for malicious
    user inputs... (prompt trimmed for brievity)
    """,
    prompt="""
    You are analyzing the Java package {{source_project_org}}/
    {{source_project_name}}.

    Please look at the following public method in the library
    and its documentation. Does it look like can be invoked by
    a downstream Java package, and that the function can be called
    with potentially malicious end-user inputs?

    package: {{package}}, class: {{clazz}}, method: {{method}}
    parameter: {{param_ty}} {{param_name}}
    """,
    model="gpt-4o"
)
type func_param_is_source(
    bound source_project_org: String,
    bound source_project_name: String,
    bound package: String,
    bound clazz: String,
    bound method: String,
    bound param_name: i32,
    bound param_ty: i32,
    is_source: bool,
)

// Inferring function parameter's taint specifications
rel source_func_param(p, c, m, i) =
    source_project_info(proj_org, proj_name) and
    func_param_candidate(p, c, m, doc, i, p_name, p_ty) and
    func_param_is_source(proj_org, proj_name,
                         p, c, m, p_name, p_ty, true)
```

Listing 7.2: Querying large language models for function parameter taint specification.

```
import "codeql.scl"

// A CodeQL reachability query which takes in sources and sinks
// and producing a Path. The Path details can be queried later.
@codeql_path_query
type reachable(bound source: Node, bound sink: Node, path: Path)

// Getting the dataflow node associated with the source
rel source_node(node) =
    callable(callable) and get_declaring_type(callable, decl) and
    get_package(decl, p) and get_name(decl, c) and
    get_name(callable, m) and
    source_func_param(p, c, m, i) and
    get_parameter(callable, i, param) and
    get_dataflow_node(param, node)
// ...the definition of sink_node and barrier is omitted

// Performing reachability analysis for vulnerability candidate
rel vulnerability(source, sink, path) =
    source_node(source) and sink_node(sink) and
    reachable(source, sink, path) and
    forall(step: dataflow_path_step(path, from, to) implies
        not barrier(from, to))
```

Listing 7.3: Querying CodeQL for reachability analysis.

and JavaDoc documentations, if applicable. In practice, we find that this extra information helps LLM understand the high-level purpose and usage of the codebase, resulting in better labeling accuracy.

In Listing 7.2, we show an `@gpt` annotated foreign predicate `func_param_is_source`, which is designed to classify whether a given function parameter may act as a potential source of tainted data. The corresponding system and user prompts, which guide the model's reasoning, are shown (in slightly abbreviated form) in lines 3-10 and 13-22, respectively. The model produces a boolean label `is_source` (line 34) to indicate its prediction. According to the rule defined in lines 38-42, we first retrieve candidate parameters—extracted via the logic in Listing 7.1—and then apply GPT to label each of them accordingly. This integration allows Scallop to seamlessly incorporate LLM-generated inferences into its symbolic reasoning pipeline.

7.3.3 Taint Reachability Analysis

Once we obtain all the source and sink specifications from the LLM, the next step is to combine it with a static analysis engine to detect vulnerable paths, i.e., $\mathit{Unsanitized\_Paths}(V_s, V_t)$, in a given project. In this work, we use CodeQL (Avgustinov et al., 2016) for this step. Given a data flow graph $\mathbb{G}^P$ of a project P, CWE-specific source and sink specifications, and a query for a given vulnerability class C, CodeQL returns a set of unsanitized paths in the program. Formally,

$$\mathit{CodeQL}(\mathbb{G}^P, \boldsymbol{S}^C_{source}, \boldsymbol{S}^C_{sink}, \mathit{Query}^C) = \{\mathit{Path}_1, \ldots, \mathit{Path}_k\}.$$

CodeQL itself contains numerous specifications of third-party APIs for each vulnerability class. However, as we show later in our evaluation, despite having such specialized queries and extensive specifications, CodeQL fails to detect a majority of vulnerabilities in real-world projects. For our analysis, we write a specialized CodeQL query for each vulnerability that uses our mined specifications instead of those provided by CodeQL.

In Listing 7.3, we sketch out the Scallop program that performs the core reachability analysis for vulnerability detection. The program begins by loading a foreign predicate reachable using the `@codeql_path_query` attribute provided by the Scallop-CodeQL plugin (lines 5-6), which connects to a CodeQL path query to determine dataflow reachability between nodes in the program. The program then derives the labeled sources, sinks, and sanitizers—where sanitizers are encoded as barriers to taint propagation—using rules defined in lines 9-16. Finally, this reachability information is combined with LLM-inferred specifications of sources, sinks, and sanitizers to identify actual vulnerable paths in the program (lines 19-23), enabling precise vulnerability detection.

7.3.4 Triaging of Alerts Via Contextual Analysis

Inferring taint specifications only solves part of the challenge. We observe that while LLMs help uncover many new API specifications, sometimes they detect specifications that are not relevant to the vulnerability class being considered, resulting in too many predicted sources or sinks and consequently many spurious alerts as a result. For context, even a few hundred taint

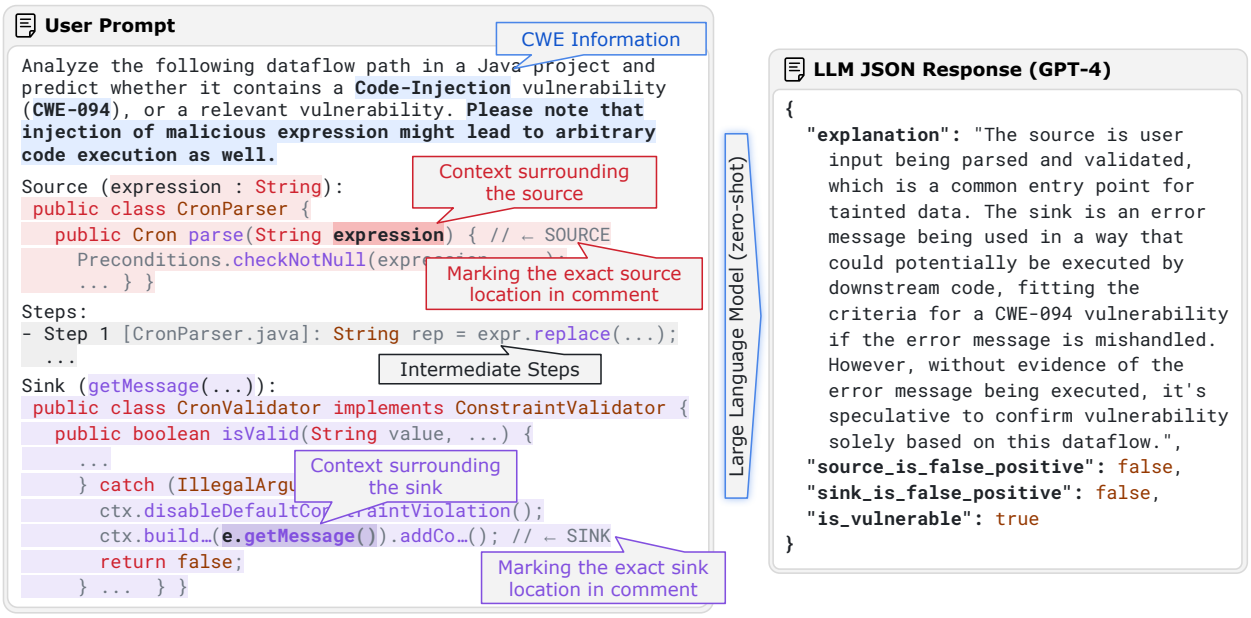


Figure 7.4: LLM user prompt and response for contextual analysis of dataflow paths. In the user prompt, we mark with color the CWE and path information that is filling the prompt template. For cleaner presentation, we modify the snippets and left out the system prompt.

specifications may sometimes produce thousands of $Unsanitized\_Paths(V_s, V_t)$ that a developer needs to triage. To reduce the developer burden, we also develop an LLM-based filtering method, $FilterPath(Path, \mathbb{G}, \texttt{LLM}, C) = \texttt{True} \mid \texttt{False}$ that classifies a detected vulnerable path ($Path$) in $\mathbb{G}$ as a true or false positive by leveraging context-based and natural language information.

Figure 7.4 presents an example prompt for contextual analysis. The prompt includes CWE information and code snippets for nodes along the path, with an emphasis on the source and sink. For the intermediate steps, we include the file names and the line of code. When the path is too long, we keep only a subset of nodes to limit the size of the prompt. In addition, if the verdict is false, we ask the LLM to indicate whether the source or sink is a false positive, which helps to prune other paths and thereby save on the number of calls to the LLM.

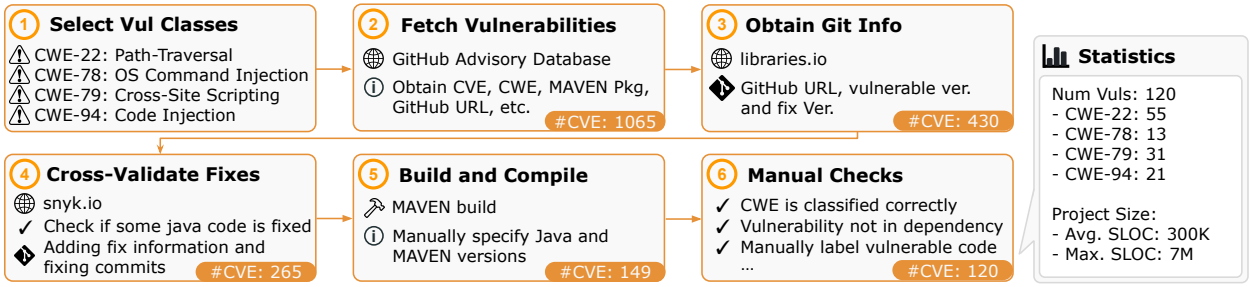


Figure 7.5: Steps for curating CWE-Bench-Java, and dataset statistics.

7.4 Empirical Evaluation

We perform extensive experimental evaluations of IRIS-Scallop and demonstrate its practical effectiveness in detecting vulnerabilities in real-world Java repositories in CWE-Bench-Java. We address the following key research questions:

- **RQ 1:** How many previously known vulnerabilities can IRIS-Scallop detect?
- **RQ 2:** Does IRIS-Scallop detect new, previously unknown vulnerabilities?
- **RQ 3:** How good are the inferred source/sink specifications by IRIS-Scallop?

7.4.1 CWE-Bench-Java Dataset

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

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

which are prone to security risks. Further, due to Java's long history, there are many existing CVEs in numerous Java projects that are available for analysis. We initially use the GitHub Advisory database GitHub (2024a,b) to obtain such vulnerabilities, and further filter it with cross-validated information from multiple sources, including manual verification. Figure 7.5 illustrates the complete set of steps for curating CWE-Bench-Java.

As shown in the statistics (Figure 7.5), the sheer size of these projects make them challenging to analyze for any static analysis tool or ML-based tool. Each project in CWE-Bench-Java comes with GitHub information, vulnerable and fix version, CVE metadata, a script that automatically fetches and builds, and the set of program locations that involve the vulnerability.

7.4.2 Experimental Setup

We select two closed-source LLMs from OpenAI: GPT-4 (`gpt-4-0125-preview`) (OpenAI et al., 2024) and GPT-3.5 (`gpt-3.5-turbo-0125`) (Brown et al., 2020) for our evaluation. We also select instruction-tuned versions of four open-source LLMs via huggingface API: Llama 3 (L3) 8B and 70B (Grattafiori et al., 2024), Qwen-2.5-Coder (Q2.5C) 32B (Hui et al., 2024), Gemma-2 (G2) 27B (Team et al., 2024), and DeepSeekCoder (DSC) 7B (Guo et al., 2024). For the CodeQL baseline, we use version 2.15.3 and its built-in `Security` queries specifically designed for each CWE. Other baselines included are Facebook Infer (FB Infer), SpotBugs (Lavazza et al., 2020), and Snyk (Snyk.io).

7.4.3 Evaluation Metrics

We evaluate the performance of IRIS-Scallop and its baselines using three key metrics: number of vulnerability detected (*#Detected*), average false discovery rate (*AvgFDR*), and average F1 (*AvgF1*). For evaluation, we assume that we have a dataset $\mathcal{D} = \{P_1, \ldots, P_n\}$ where each P_i is a Java project, and known to contain at least one vulnerability. The label for a project P is provided as a set of crucial program points $\mathbf{V}^P_{\text{vul}} = \{V_1, \ldots, V_n\}$ where the vulnerable paths should pass through, indicated by $\mathit{Path} \cap \mathbf{V}^P_{\text{vul}} \neq \emptyset$. In practice, these are typically the patched methods that can be collected from each vulnerability report. If at least one detected vulnerable path passes through a

fixed location for the given vulnerability, then we consider the vulnerability detected. Let $Paths^P$ be the set of detected paths for each project P from prior stages. The evaluation metrics are formally defined as follows:

$$\#VulPath(P) = |\{Path \in Paths^P \mid Path \cap \mathbf{V}_{\text{vul}}^P \neq \emptyset\}|, \tag{7.1}$$

$$\#Detected(\mathcal{D}) = \sum_{P \in \mathcal{D}} Rec(P), \tag{7.2}$$

$$AvgFDR(\mathcal{D}) = \text{avg}_{P \in \mathcal{D}, |Paths^P| > 0} 1 - Prec(P), \tag{7.3}$$

$$Rec(P) = \mathbb{1}_{\#VulPath(P) > 0}, \tag{7.4}$$

$$Prec(P) = \frac{\#VulPath(P)}{|Paths^P|}, \tag{7.5}$$

$$AvgF1(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{P \in \mathcal{D}} \frac{2 \cdot Prec(P) \cdot Rec(P)}{Prec(P) + Rec(P)}. \tag{7.6}$$

Specifically, a lower *AvgFDR* is preferable, as it indicates a lower ratio of false positives. We note that $Prec(P)$ might sometimes be undefined due to division-by-zero if the detection tool retrieves no path ($|Paths^P| = 0$). Therefore, for *AvgFDR* to be meaningful, we only consider the projects where at least one positive result is produced ($|Paths^P| > 0$). *AvgF1* avoids this issue since $Rec(P) = 0$ when no positive labels exist, forcing the F1 term to be zero regardless of $Prec(P)$.

7.4.4 RQ1: Effectiveness of IRIS-Scallop on Detecting Existing Vulnerabilities

The results in Table 7.1 highlight IRIS-Scallop's superior performance compared to CodeQL. Specifically, IRIS-Scallop, when paired with GPT-4, identifies 55 vulnerabilities—28 more than CodeQL. While GPT-4 shows the highest efficacy, smaller, specialized LLMs like DeepSeekCoder 7B still detect 52 vulnerabilities, suggesting that our approach can effectively leverage smaller-scale models, enhancing accessibility. Notably, this increase in detected vulnerabilities does not compromise precision, as evidenced by IRIS-Scallop's lower average false discovery rate (FDR) with GPT-4 compared to CodeQL.

Moreover, IRIS-Scallop improves average F1 by 0.1, reflecting a better balance between precision and recall. We note that the reported average FDR is a coarse measure as our metrics may consider a

	Method	#Det (/120)	Det Rate (%)	Avg FDR (%)	Avg F1
	CodeQL	27	22.50	90.03	0.076
IRIS +	GPT-4	**55** (↑ 28)	**45.83** (↑ 23.33)	**84.82** (↓ 5.21)	**0.177** (↑ 0.101)
	GPT-3.5	47 (↑ 20)	39.17 (↑ 16.67)	90.42 (↑ 0.39)	0.096 (↑ 0.020)
	L3 8B	41 (↑ 14)	34.17 (↑ 11.67)	95.55 (↑ 5.52)	0.058 (↓ 0.018)
	L3 70B	54 (↑ 27)	45.00 (↑ 22.50)	90.96 (↑ 0.93)	0.113 (↑ 0.037)
	Q2.5C 32B	47 (↑ 20)	39.17 (↑ 16.67)	92.38 (↑ 2.35)	0.097 (↑ 0.021)
	G2 27B	45 (↑ 18)	37.50 (↑ 15.00)	91.23 (↑ 1.20)	0.100 (↑ 0.024)
	DSC 7B	52 (↑ 25)	43.33 (↑ 20.83)	95.40 (↑ 5.37)	0.062 (↓ 0.014)

Table 7.1: Overall performance comparison of CodeQL vs IRIS-Scallop on Detection Rate (↑), Average FDR (↓), and Average F1 (↑). We present results of IRIS-Scallop with LLMs including GPT-4 and GPT-3.5, L3 8B and 70B, Q2.5C 32B, G2 27B, and DSC 7B.

true (but unknown) vulnerability found by IRIS-Scallop as a false positive. Hence, the reported FDR is an upper bound. To get a better sense of detection accuracy, we manually analyzed 50 random alarms reported by IRIS-Scallop (using GPT-4) and found that 27/50 alarms exhibit potential attack surfaces, *yielding a more refined estimated false discovery rate of 46%*. Hence, IRIS-Scallop will likely be more effective in practice.

Table 7.2 presents a detailed breakdown of detected vulnerabilities, comparing IRIS-Scallop against various baselines. With the exception of IRIS-Scallop using Llama-3 8B, which underperforms in detecting CWE-22 vulnerabilities, IRIS-Scallop consistently outperforms all other baselines. Notably, CWE-78 (OS Command Injection) remains particularly challenging for all LLMs. Our manual investigation revealed that the vulnerability patterns in CWE-78 are highly intricate, often involving OS command injections via gadget-chains (Cao et al., 2023) or external side effects, such as file writes, which are difficult to track using static analysis. This highlights the inherent limitations of static analysis, as opposed to dynamic approaches—an area that we leave for future work.

7.4.5 RQ2: Previously Unknown Vulnerabilities by IRIS-Scallop

We applied IRIS-Scallop with GPT-4 to the latest versions of 30 Java projects. Among the 16 inspected projects where IRIS-Scallop raised at least one alert, we identified 4 vulnerabilities, including 3 instances of path injection (CWE-22) and one case of code-injection (CWE-94). To

CWE	#Vuls	Baselines				IRIS-Scallop with				
		QL	Infer	SB	Snyk	GPT-4	GPT-3.5	L3 8B	L3 70B	DSC 7B
CWE-22	55	22	0	2	21	**31** (↑ 9)	25 (↑ 3)	19 (↓ 3)	29 (↑ 7)	25 (↑ 3)
CWE-78	13	1	0	1	1	**3** (↑ 2)	1 (= 0)	2 (↑ 1)	2 (↑ 1)	**3** (↑ 2)
CWE-79	31	4	0	1	1	13 (↑ 9)	13 (↑ 9)	9 (↑ 9)	**14** (↑ 10)	**14** (↑ 10)
CWE-94	21	0	0	0	0	8 (↑ 8)	8 (↑ 8)	**11** (↑ 11)	9 (↑ 9)	10 (↑ 10)
All	120	27	0	4	23	**55** (↑ 28)	47 (↑ 20)	41 (↑ 14)	54 (↑ 27)	52 (↑ 25)

Table 7.2: Per-CWE statistics of number of vulnerabilities detected (*#Detected*) by baselines and IRIS. The compared baselines are CodeQL (QL), Facebook Infer (Infer), Spotbugs (SB), and Snyk. The values in parentheses show the differences of detection by IRIS-Scallop against CodeQL.

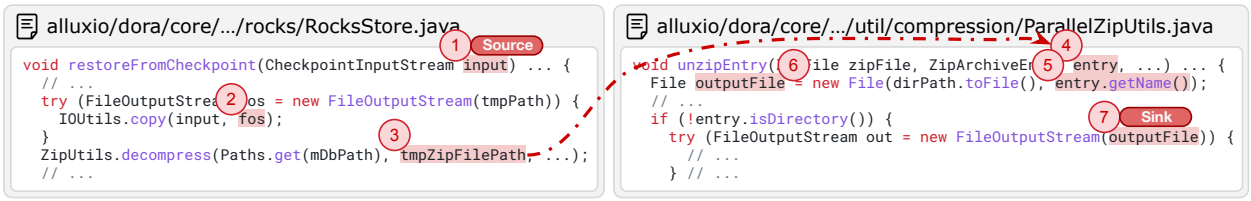


Figure 7.6: A previously unknown vulnerability found in alluxio 2.9.4. The snippets are slightly modified for presentation purpose. A user with database restoration permission may supply a database checkpoint Zip file with malicious entry name. When unzipped, the entry may be written to an arbitrary directory, causing a Zip-Slip vulnerability (CWE-022) that could corrupt the hosting server.

ensure that these vulnerabilities were indeed uncovered due to IRIS-Scallop's integration with LLMs, we verified that CodeQL alone did not detect them. We highlight one such vulnerability in Figure 7.6. CodeQL was unable to detect this issue due to a missing source specification, while GPT-4 successfully flagged the API endpoint `restoreFromCheckpoint` as a potential entry point for attack.

7.4.6 RQ3: Quality of LLM-Inferred Taint Specifications

The LLM-inferred taint specifications are fundamental to IRIS-Scallop's effectiveness. To assess the quality of these specifications, we conducted two experiments. First, we used CodeQL's taint specifications as a benchmark to estimate the recall of both source and sink specifications inferred by LLMs (Figure 7.7). However, since CodeQL offers a limited set of specifications, we also needed to assess the quality of inferred specifications outside of its known coverage. To this end, we manually

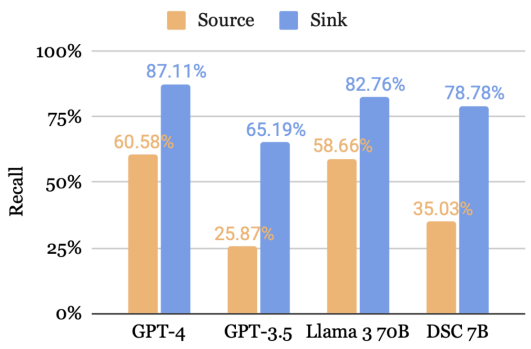


Figure 7.7: Recall of LLM-inferred taint specifications against CodeQL's taint specifications.

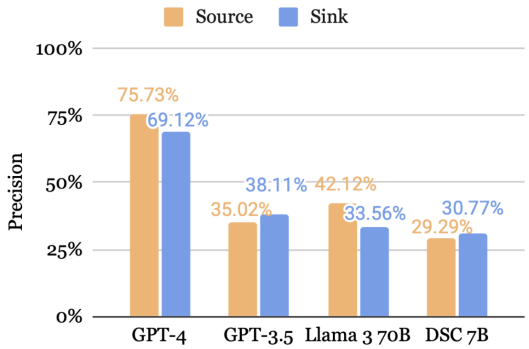


Figure 7.8: Estimated precision of LLM-inferred specifications on randomly sampled labels.

analyzed 960 randomly selected samples of LLM-inferred source and sink labels (30 per combination of CWE and LLM) and estimated the overall precision of the specifications (Figure 7.8).

LLM-inferred sinks can replace CodeQL sinks Overall, LLMs demonstrated high recall when tested against CodeQL's sink specifications (Figure 7.7), with GPT-4 scoring the highest (87.11%). While the recall for source specifications was generally lower, we found that CodeQL tends to over-approximate its source specifications to compensate for a low detection rate. On the other hand, GPT-4 achieved high precision (over 70%) in manual evaluations (Figure 7.8), aligning with the lower false discovery rate previously reported in Table 7.1. For other LLMs, the combination of high recall but lower precision suggests a tendency to over-approximate sink specifications.

Over-approximating specifications can benefit Although the precision for LLMs other than GPT-4 is lower, over-approximation can actually help address a core limitation of CodeQL—its restricted set of taint specifications. By over-approximating, LLMs expand the coverage of taint analysis, offering a partial solution to CodeQL's limited scope. The impact of this imprecision can be mitigated through contextual analysis as we show next in the ablation studies.

7.5 Related Work

Learning-based approaches for vulnerability detection Numerous prior techniques incorporate deep learning for detecting vulnerabilities. This includes Graph Neural Network based models such as Zhou et al. (2019); Chakraborty et al. (2020); Dinella et al. (2020); Hin et al. (2022); Li et al. (2021b); LSTM-based models such as Li et al. (2020b, 2021c); and fine-tuning of Transformer-based models such as Fu and Tantithamthavorn (2022); Steenhoek et al. (2023); Cheng et al. (2022). These approaches focus on method-level detection of vulnerabilities and provide only a binary label classifying a method as vulnerable or not. In contrast, IRIS-Scallop performs whole-project analysis and provides a distinct code path from a source to a sink and can be tailored for detecting different CWEs. More recently, multiple studies demonstrated that LLMs are not effective at detecting vulnerabilities in real-world code (Steenhoek et al., 2024; Ding et al., 2024; Khare et al., 2023). While these studies only focused on method-level vulnerability detection, it reinforces our motivation that detecting vulnerabilities requires whole-project reasoning, which LLMs currently cannot do alone.

Static analysis tools Apart from CodeQL (Avgustinov et al., 2016), other static analysis tools (CPPCheck; Semgrep, 2023; FlawFinder; FB Infer; Code Checker) also include analyses for vulnerability detection. More general query engines (Scholz et al., 2016; Li et al., 2023b) have also been applied to find program bugs. But these tools are not as feature-rich and effective as CodeQL (Li et al., 2023a; Lipp et al., 2022). Recently, proprietary tools such as Snyk (Snyk.io) and SonarQube (SonarQube) are also gaining in popularity, although sharing the same fundamental limitations of missing specifications and false positives, which IRIS-Scallop improves upon. We envision our technique to benefit all such tools. Works such as MERLIN (Livshits et al., 2009), Seldon (Chibotaru et al., 2019) and InspectJS (Dutta et al., 2022) tackle the problem of specification inference through probabilistic modeling. Specifically, like IRIS-Scallop, InspectJS also augments CodeQL with specifications inferred using machine learning. However, InspectJS relies on the quality of seed specifications and requires expensive analysis of each third-party library, which IRIS-Scallop does not—making it more scalable. Future work could explore incorporating probability estimates for specifications.

LLM-based approaches for software engineering Researchers are increasingly combining LLMs with program reasoning tools for challenging tasks such as fuzzing (Lemieux et al., 2023; Xia et al., 2024), program repair (Xia et al., 2023; Joshi et al., 2023; Xia and Zhang, 2022), and fault localization (Yang et al., 2023a). While we are on a similar direction as (Li et al., 2024a; Wang et al., 2024a), to our knowledge, our work is among the first to combine LLMs with static analysis to detect application-level security vulnerabilities via whole-project analysis.

CHAPTER 8

APPLICATION: RNA SECONDARY STRUCTURE PREDICTION

Ribonucleic acids (RNAs) play vital roles in biological systems, going far beyond their well-known part in the central dogma. They take on diverse functions—ribozymes act as catalysts, riboswitches sense metabolites and regulate gene expression, while microRNAs (miRNAs), long non-coding RNAs (lncRNAs), and circular RNAs contribute to regulation in development and disease. These functions are made possible by the RNA's structure, which enables it to fold into specific shapes, bind to different molecules, and carry out complex cellular tasks (Spitale and Incarnato, 2023; Assmann et al., 2023; Ganser et al., 2019; Edwards et al., 2007; Fu, 2014). Understanding RNA structure is thus not only key to basic biology but also opens up paths for medical applications, including RNA-based therapeutics and vaccines.

RNA structure is organized in a hierarchy: the sequence of nucleotides (primary structure) folds into secondary structures like stems and loops (Figure 8.1), which then assemble into complex three-dimensional forms. Among these, the secondary structure is especially important because it forms the foundation for further folding and largely determines RNA function (Mathews et al., 2010). Secondary structure arises through both standard base pairing (like `A-U` and `G-C`) and less common interactions (like `G-U` pairs), creating recognizable patterns of paired and unpaired bases. Interestingly, these secondary structures tend to be more conserved than the actual RNA sequence across species, underscoring their functional importance. However, experimentally determining RNA structure is challenging due to technical and practical limitations, making computational methods essential.

There are three main strategies for predicting RNA secondary structure: thermodynamic, comparative, and neural approaches. Thermodynamic models use physical principles to predict the most stable structure by minimizing free energy but can struggle with unusual structures like pseudoknots. Comparative methods use evolutionary information, finding conserved structures through multiple sequence alignments and probabilistic models such as stochastic context-free grammars. These

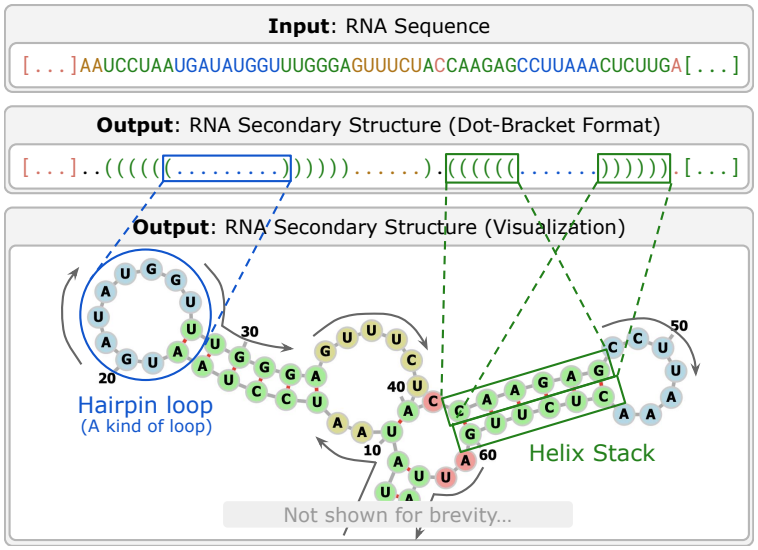


Figure 8.1: Illustration of the RNA folding problem. We visualize the output RNA secondary structure in the bottom. The grey arrow indicates the direction in which the indexes of nucleotides are increasing.

models detect base-pair covariation, choosing structures that best explain observed mutations. Neural methods take a different path, using machine learning to identify structural patterns directly from data without relying on explicit physical rules or sequence alignments.

A neurosymbolic solution is well-motivated for RNA secondary structure prediction because the task itself sits at the intersection of learned heuristics and strict structural rules. Purely neural approaches often fall short when it comes to modeling long-range dependencies and enforcing structural consistency—such as ensuring valid base-pairing and avoiding overlapping pairs—especially in the presence of noisy or ambiguous data. Conversely, purely symbolic methods lack adaptability; while they can enforce pairing rules and parse sequences using grammars, they struggle to generalize to varied datasets or capture subtle statistical signals from biological data. By integrating both paradigms, a neurosymbolic system can leverage the neural model to propose high-quality local structure predictions while relying on a symbolic parser to ensure global consistency and adherence to biochemical constraints. This combination enables more accurate, interpretable, and generalizable RNA structure prediction.

In this chapter, we introduce **ScallopFold**, a neurosymbolic system that combines a fine-tuned

RNA foundation model with a Scallop-based symbolic parser grounded in a context-free grammar (CFG). Section 8.1 defines the problem and outlines our approach. Section 8.2 details ScallopFold (Section 8.2.1) and its GPU-accelerated runtime, **Lobster**, which significantly boosts symbolic parsing performance (Section 8.2.2). In Section 8.3, we present empirical results showing ScallopFold achieves both higher accuracy and faster inference. We conclude with related work in Section 8.4.

8.1 Problem Definition

Given an RNA sequence of nucleotides $\mathbf{x} = x_1 x_2 \dots x_n$, where each $x_i \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$, the goal is to determine its *secondary structure*, a pairing map that specifies, for each position i, whether it remains unpaired or forms a base pair with some position j. A common way to represent this structure is through *dot-bracket notation*, where unpaired nucleotides are denoted by dots (`'.'`) and each base-pair (i, j) is represented by matched brackets (e.g., `'('` at position i and `')'` at position j), as shown in Figure 8.1. For the structure to be valid, all brackets must be properly matched and nested, forming a well-balanced string. Such properties may naturally be captured by a formal grammar, which we name $\mathcal{L}_{\mathsf{RSS}}$.

For data-driven learning, we are given a dataset $\mathcal{D} = \{(\mathbf{x}, \mathbf{y})\}$, where $\mathbf{x} = x_1 x_2 \dots x_n$ is an RNA sequence, and $\mathbf{y} \in \mathcal{L}_{\mathsf{RSS}}$ is the corresponding RNA secondary structure represented in dot-bracket notation. The goal is to learn a model f_θ, parameterized by neural weights θ, that maps an input sequence $\mathbf{x}$ to a predicted structure $\hat{\mathbf{y}} = f_\theta(\mathbf{x}) \in \mathcal{L}_{\mathsf{RSS}}$.

During training, we minimize a loss function $\mathcal{L}$ that compares the predicted structure $\hat{\mathbf{y}}$ with the ground truth $\mathbf{y}$. The training objective is defined as:

$$J(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y}). \tag{8.1}$$

This setup defines the RNA secondary structure prediction task as a supervised sequence-to-structure learning problem.

8.2 Neurosymbolic Solution with Scallop

8.2.1 Neurosymbolic RNA Secondary Structure Prediction

A neurosymbolic approach A neurosymbolic approach is particularly well-suited for this problem due to its hybrid nature: while the pairing constraints are symbolic and rule-based, the folding process is inherently non-deterministic, as RNA molecules may adopt multiple valid structures depending on environmental conditions. Despite this ambiguity, certain biochemical principles are consistent—such as pairing constraints that restrict allowable base pairs to $(\texttt{A}, \texttt{U})$, $(\texttt{C}, \texttt{G})$, and wobble pair $(\texttt{A}, \texttt{G})$, and structural constraints that enforce proper bracket nesting and spacing.

In our solution, the neural component predicts a latent annotation $\mathbf{t} = t_1 t_2 \dots t_n$, where each t_i indicates the likely structural role of nucleotide x_i (e.g., part of a helix, a loop, or unpaired). These predictions capture local statistical cues from data. The symbolic component then parses the sequence $\mathbf{x}$, guided by $\mathbf{t}$, into a globally consistent secondary structure $\mathbf{y} \in \mathcal{L}_{\mathsf{RSS}}$, where $\mathcal{L}_{\mathsf{RSS}}$ denotes the language of well-formed RNA secondary structures defined by a context-free grammar (CFG). Without this integration, a purely neural model may struggle to enforce global pairing constraints or model long-range dependencies, while a purely symbolic model may fail to generalize to the underlying distribution of real-world RNA folds.

Probabilistic parsing Our solution employs a CFG defined over a set of *terminal symbols*, or *structure tokens*, as illustrated in Figure 8.2a. Let the input RNA sequence be $\mathbf{x} = x_1 x_2 \dots x_n$, where each nucleotide $x_i \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$. For each position i, a neural network outputs a probability distribution $\mathbf{p}_i$ over the token vocabulary Σ (the set of terminal symbols), representing a data-driven estimate of the structural role $t_i \in \Sigma$ at that position.

The symbolic component, a probabilistic parser, then processes the sequence of distributions $\bar{\mathbf{p}} = \mathbf{p}_1, \dots, \mathbf{p}_n$ to find the most probable parse tree $\mathcal{R} = \texttt{parse}(\bar{\mathbf{p}})$. This parsing step enforces global structural constraints, such as balanced pairing and valid motif composition, while being guided by local probabilistic predictions from the neural model (Figure 8.2b). Finally, to produce the RNA dot-bracket notation, we extract the set of paired positions $\{(i, j)\}$ from the parse tree $\mathcal{R}$, which

$$\text{(Structure Token)} \quad \Sigma \ni \tau ::= H_l \mid H_r \mid L_l \mid L_r \mid L_u \mid E_u$$

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

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

(b) The complete CFG $\mathcal{L}_{\mathsf{RSS}}$ used to parse RNA secondary structures ($\mathcal{R}$).

Figure 8.2: The set of terminal symbols used to represent RNA secondary structure tokens, along with a context-free grammar (CFG) that parses a sequence of these tokens into a valid secondary structure.

correspond to matched structural tokens (e.g., opening and closing brackets) in the CFG. The result is a well-formed, interpretable secondary structure consistent with both learned statistical patterns and formal grammatical rules.

Scallop implementation We implement the symbolic component of our system as a Scallop program. A partial program is shown in Listing 8.1. At a high-level, the base input is encoded in the `rna` relation (Line 6), and the to-be-learnt probabilities of structure tokens is passed in as another input in the `token` relation (Line 9). The program then recursively builds the parse tree according to the grammar in a bottom-up fashion. Scallop offers a high-level declarative interface for specifying both data types (lines 2-3) and input relations (lines 6 and 9). Specifically, the RNA sequence is encoded as a binary relation `rna(idx, nuc)`, mapping each position index i to a nucleotide $x_i \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$.

The program leverages recursive rules to capture the hierarchical structure specified by the context-free grammar. In line 15, the predicate `paired_ss(i, j)` defines paired substructures ($\mathcal{P}$), which may correspond to either an internal loop ($\mathcal{L}$) or a helix-stack ($\mathcal{H}$). Lines 20-22 specify the rule for helix-stacks $\mathcal{H}$: given that the nucleotides at positions i and j are bondable (line 21), the interval

```
1  // enum type for nucleotides and structure tokens
2  type Nucleotide = A | C | G | U
3  type StructureToken = Hl | Hr | Ll | Lr | Lu | Eu
4
5  // input RNA seq (index mapped to nucleotide)
6  type rna(idx: usize, nuc: Nucleotide)
7
8  // probabilistic tokens extracted by RNA-FM
9  type token(idx: usize, tok: StructureToken)
10
11 // facts for nucleotide pairs that can be bonded
12 rel can_bond = {(A, U), (U, A), ...}
13
14 // Rule for paired-substructures
15 rel paired_ss(i, j) = loop(i, j) or helix(i, j)
16
17 // Rule for helix stack:
18 // - i, j must be bondable
19 // - Hl * paired_ss * Hr: production rule for helix
20 rel helix(i, j) =
21   rna(i, x_i) and rna(j, x_j) and can_bond(x_i, x_j) and
22   token(i, Hl) and paired_ss(i + 1, j - 1) and token(j, Hr)
23
24 // ... other rules for parsing RNA are omitted for brevity
```

Listing 8.1: A partial Scallop program for parsing RNA structure token sequences.

$[i, j]$ forms a valid helix if the token at position i is a left-helix marker (`Hl`), the token at j is a right-helix marker (`Hr`), and the inner span $[i+1, j-1]$ recursively forms a valid paired substructure (line 22). These declarative rules directly reflect the grammar productions shown in Figure 8.2b, highlighting the tight alignment between symbolic parsing and Scallop's logical representation.

A key advantage of Scallop is its ability to perform probabilistic reasoning over symbolic structures. In our setting, the neural network outputs, for each position i, a probability distribution over structure tokens $\mathbf{p}_i \in \Delta^{|\Sigma|}$, where Σ is the set of structure tokens (e.g., `Hl`, `Hr`, etc.). These distributions are encoded as probabilistic facts in Scallop via the `token(idx, tok)` relation, associating each token $t_i \in \Sigma$ with its corresponding probability at position i.

Rather than relying on a single hard prediction per position, Scallop parses over the entire distribution

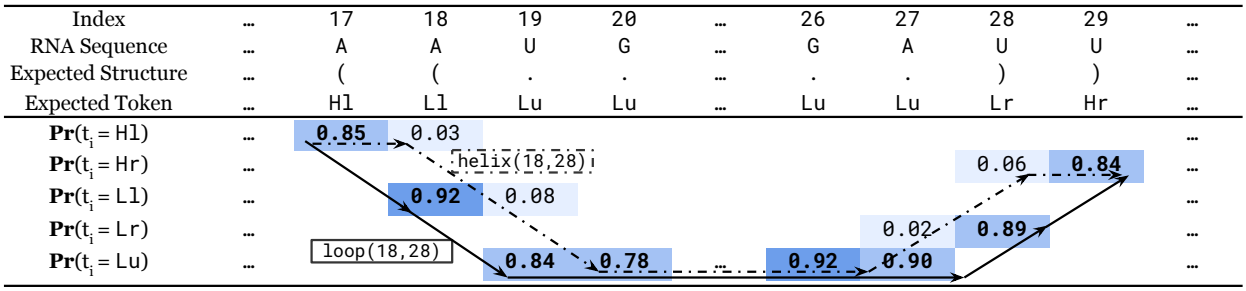


Figure 8.3: We illustrate two plausible ways of parsing the RNA subsequence `AAUG...GAUU`: one more probable (solid arrow) and one less probable (dashed arrow). The difference is whether the subsequence is parsed into a helix stack or a loop.

space, allowing it to explore and score multiple possible parses consistent with the grammar. This enables the system to compute the most probable parse, i.e., the derivation tree $\mathcal{R}$ that maximizes the joint probability under the token distributions and the grammatical constraints. As a result, the output of the parser is not just a binary decision about structure validity, but a ranked set of possible secondary structures, each corresponding to a parse derivation along with its associated likelihood. This probabilistic parsing capability allows Scallop to reconcile uncertainty in neural predictions with strict symbolic consistency, leading to robust and interpretable RNA structure predictions.

Neural component Given that Scallop can perform probabilistic parsing over a sequence of structure token distributions, our goal is to design a neural network that predicts these latent annotations, where each structure token corresponds to a likely structural role (e.g., helix start, loop unpaired, etc.). The desired neural model should be contextual and capable of capturing long-range dependencies; it should also be *fine-tunable* to adapt to training data while maintaining strong generalization. While classical sequence models such as LSTM (Hochreiter and Schmidhuber, 1997) or BiLSTM (Graves et al., 2013) could be used, we instead primarily adopt RNA-FM (Chen et al., 2022a), a pre-trained foundation model for RNA sequence representation.

RNA-FM has been trained on large-scale RNA data using masked language modeling, and as a result, it captures rich contextual embeddings that reflect both local and global structural patterns in RNA sequences. On top of RNA-FM, we add a randomly initialized classification layer that outputs, for each position i, a softmax distribution over the structure token vocabulary Σ. During training, we

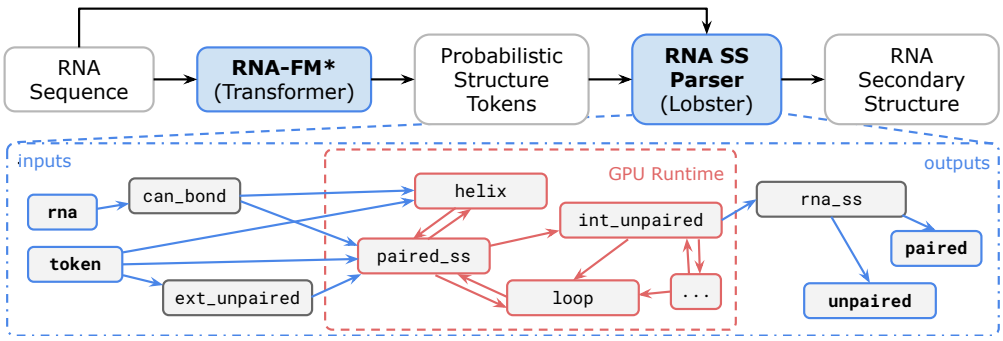


Figure 8.4: The ScallopFold pipeline. Relations in red illustrates the computation that are off-loaded to GPU accelerated Scallop runtime, Lobster.

jointly fine-tune the RNA-FM encoder and this classification layer using Scallop as the downstream parser. This end-to-end learning loop enables the neural network to adapt its predictions to better support symbolic parsing, resulting in more accurate and structurally consistent RNA secondary structure predictions.

8.2.2 Accelerating ScallopFold with Specialized Hardware

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

While custom GPU implementations have been developed for these algorithms (Yi et al., 2014), it demands specialized knowledge (e.g., parallel programming in CUDA) and hinders domain experts from focusing on functionality. Even for the problem of RNA folding, such experts have designed diverse CFGs tailored to specific biological contexts, making it impractical to implement custom GPU solutions for each CFG variant. A general purpose framework is therefore desirable for making accelerated GPU computation widely accessible.

```
@gpu
rel paired_ss(i, j) = loop(i, j) or helix(i, j)
```

Listing 8.2: Using `@gpu` attribute to tell Scallop to use the GPU runtime to accelerate relevant relations within the same stratum.

A GPU accelerated runtime With the declarative high-level language, Scallop offers a convenient abstraction to hide the underlying GPU runtime from users. We propose Lobster, a GPU-accelerated framework designed to enhance the scalability of neurosymbolic programming (Biberstein et al., 2025). The core innovation of Lobster lies in efficiently mapping an expressive subset of Scallop onto GPU architectures, for different modes of reasoning. This subset includes computationally-intensive operations like join and recursion, enabling levels of scalability unattainable with single- or multi-threaded CPUs.

Figure 8.4 illustrates the high-level pipeline of ScallopFold, along with the dependency graph of relations involved in computing RNA secondary structures. The most computationally intensive component, such as the recursive parsing based on a context-free grammar, is offloaded to a GPU-accelerated runtime to maximize performance. To enable GPU execution, programmers can annotate specific relations with the `@gpu` attribute, indicating that those computations should be offloaded. As shown in Listing 8.2, this annotation is applied to the `paired_ss` relation. All mutually dependent relations within the recursive parsing logic, such as `helix` and `loop`, are automatically identified and offloaded together, enabling efficient end-to-end symbolic computation on the GPU.

Low-level language of the GPU accelerated runtime Recall that the Scallop language compiles into an intermediate representation called RAM (Section 3.2). While RAM offers a clean and expressive abstraction for CPU execution, efficient GPU acceleration requires exposing lower-level details to better match the GPU hardware execution model. To address this, we introduce APM, a low-level, assembly-style language designed specifically for GPU execution and automatically compiled from RAM.

APM explicitly represents memory allocations and operates solely through instructions that are

```
1 rel _temp_(i,j) = token(i,Hl) and paired_ss(i+1,j-1) and token(j,Hr)
```

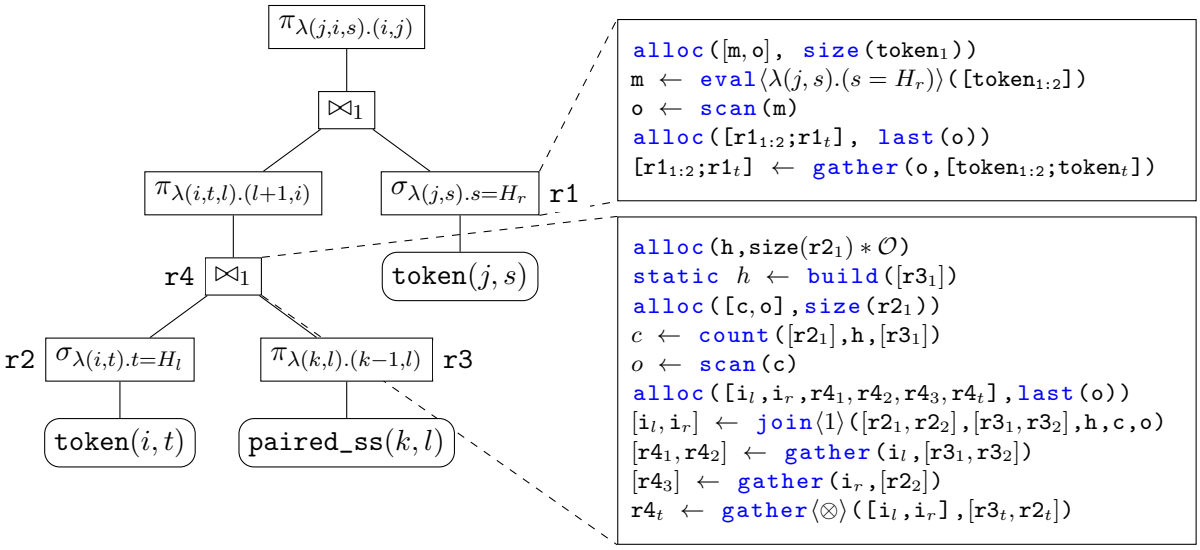
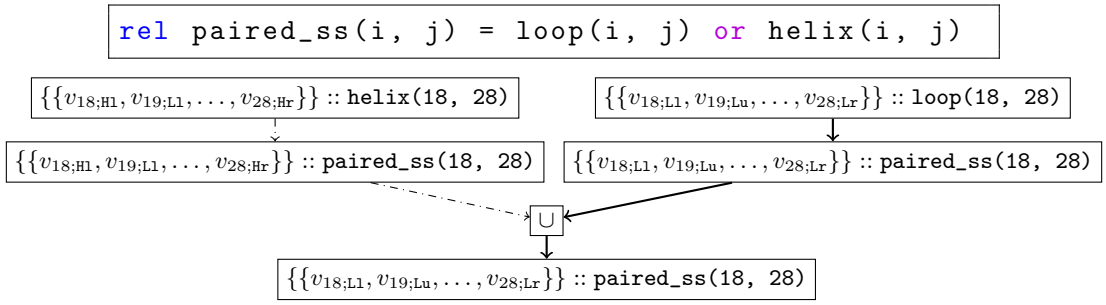


Figure 8.5: In this example, we compile a part of the rule shown in Listing 8.1 (line 22). The code block on the top shows the Scallop rule, while bottom-left illustrates the abstract syntax tree of the APM program compiled from it. We expand the node `r1` and `r4` on the right to show their low-level APM code.

amenable to massively parallel execution. Unlike general-purpose GPU programming models, APM constrains expressiveness to enforce hardware-friendly patterns. It makes GPU execution constraints explicit through its design, ensuring that any RAM program compiled into APM is guaranteed to execute efficiently on the GPU. Core APM instructions include `alloc`, `store`, `scan`, and `merge`, all of which are tailored for parallel execution. We refer the readers to Biberstein et al. (2025) for details surrounding APM.

A key design consideration in APM is the representation of relational tables in GPU memory. Since relations in Scallop have fixed schemas with a uniform number of entries per column, each relation of arity n can be represented as n parallel registers of equal size, with an additional register reserved for provenance tags. This design allows APM to treat sets of registers as tables, mirroring how RAM treats sets of facts as relations.

The compilation from RAM to APM involves flattening the RAM program, which is structured as a directed acyclic graph (DAG), into a sequential list of APM instructions. This is done via a



(a) According to the rule shown on top, there are two ways to derive the fact `paired_ss(18, 28)`. Notice how the facts carry the tag of top-1-proof, where a proof represent a concrete trace of structure tokens for the parse. Since we only keep the top-1 proof, only the proof on the right is propagated through the union ($\cup$) operation because it has a higher probability. Note that boolean variable such as $v_{18;\texttt{Ll}}$ denotes the variable corresponding to the probability $\Pr(t_{18} = \texttt{Ll})$.

i	j	empty	len	top-1-proof						
...	...	...	...	...	...	...	...	...	...	...
17	29	0	13	17;Hl	18;Hl	...	27;Lu	28;Lr	29;Hr	X
18	28	0	11	18;Hl	19;Lu	...	28;Lr	X	X	X
...	...	...	...	...	...	...	...	...	...	...

(b) The memory layout for `paired_ss` relation on GPU, where `i` and `j` are the two columns for the relation and the rest is the table for tags. The `empty` bit is to represent whether there exists a proof, while the `len` represents the number of literals within the proof. We use `18;Ll` to denote the variable ID (an integer) for the corresponding probability.

Figure 8.6: An illustration of the top-1-proof provenance in action. Consider the RNA sequence adapted from our motivating example in Figure 8.1 and Figure 8.3. While (a) shows the derivation process of the top-1-proof for the fact `paired_ss(18,28)`, (b) illustrates the corresponding memory layout after derivation.

recursive `ram-to-apm` compiler that traverses the DAG, emitting a sequence of low-level instructions and tracking the registers where each intermediate table is stored. High-level relational operations in RAM, such as join, projection, and filtering, are decomposed into multiple APM instructions that execute efficiently in parallel.

Figure 8.5 demonstrates this full compilation pipeline with an example Scallop rule for parsing RNA secondary structure, its corresponding RAM representation, and selected snippets of the compiled APM code. In summary, the introduction of APM and its compiler from RAM provides a general-purpose mechanism for GPU acceleration across all Scallop programs. Computationally intensive tasks—such as the recursive parsing required in RNA secondary structure prediction—stand to benefit significantly from this architecture.

	Method	PPV	SEN	F1
	CONTRAfold	0.607	0.679	0.638
	RNAFold	0.565	0.627	0.592
	E2EFold	0.738	0.665	0.690
	MxFold2	0.856	0.896	0.874
	ContextFold	0.873	0.821	0.842
	XTransformer	0.806	0.604	0.703
ScallopFold w/	RNAErnie	0.812	0.720	0.751
	RNA-FM	**0.923**	**0.928**	**0.924**

Table 8.1: Performance of ScallopFold on a subset of ArchiveII dataset compared to baselines.

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

Without loss of generality, we illustrate the top-1-proof provenance in Figure 8.6a, using the RNA secondary structure parsing application. It is a special case of top-k-proofs proposed in Huang et al. (2021) and is sufficiently effective in practice. In general, the proof tracks one conjunction of the corresponding boolean variables for facts used to derive the current fact. During disjunction, the provenance picks the more likely one from the two proofs by computing the probabilities of the two proofs. For conjunction, the provenance merges the two proofs while ensuring that no conflict is seen. Figure 8.6b further details the memory layout for the top-1-proof semiring, where we use extra `empty` and `len` fields to track the structure of the proof.

8.3 Empirical Evaluation

Experimental setup We evaluate ScallopFold on a subset of the ArchiveII (Sloma and Mathews, 2016) dataset, which consists of RNA sequences with lengths up to 200 nucleotides. For baseline

comparison, we include several models: CONTRAFold (Do et al., 2006), RNAFold (Lorenz et al., 2011), MxFold2 (Sato et al., 2021), E2EFold (Chen et al., 2020a), and ContextFold (Zakov et al., 2011). For ScallopFold, we experiment with three different neural backbones: RNA-Ernie (Chen et al., 2022b), RNA-FM (Chen et al., 2022a), and XTransformer. These models are used to predict token distributions, which are then parsed by Scallop's symbolic component.

Evaluation metrics The evaluation metrics used in our experiments are Positive Predictive Value (PPV), Sensitivity (SEN) and F1 Score. Specifically,

$$\text{PPV} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}, \tag{8.2}$$

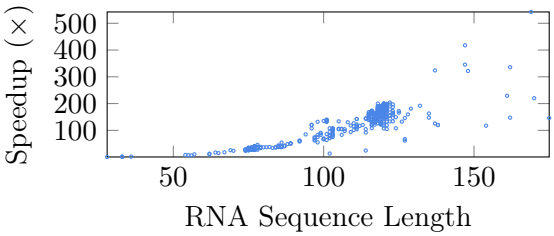
$$\text{SEN} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}, \tag{8.3}$$

$$\text{F1} = 2 \cdot \frac{\text{PPV} \cdot \text{SEN}}{\text{PPV} + \text{SEN}}. \tag{8.4}$$

Here, a predicted nucleotide pair $(i, \hat{j})$ is counted as a true positive if the ground truth also contains a pair (i, j) and $\hat{j} = j$. A true negative occurs when the nucleotide i is correctly predicted to be unpaired, matching the ground truth. A false positive arises when i is predicted to pair with $\hat{j}$, but either the ground truth has i unpaired or paired with a different nucleotide $j \neq \hat{j}$. A false negative happens when i is predicted to be unpaired, but it is actually paired in the ground truth.

Secondary structure prediction performance As shown in Table 8.1, ScallopFold equipped with RNA-FM achieves the best overall accuracy, outperforming all evaluated baselines. This result highlights the effectiveness of our neurosymbolic approach: by combining the strengths of powerful pre-trained RNA foundation models with a structured, grammar-based parser, ScallopFold is able to produce more accurate and biologically consistent RNA secondary structure predictions.

Scalability improvements with Lobster We advance the state-of-the-art by solving neurosymbolic programming's scalability challenges via GPU acceleration. Figure 8.7 shows the speedup of Lobster over the CPU baseline, Scallop, on a subset of the ArchiveII dataset (Sloma and Mathews, 2016). Scallop suffers noticeable performance drops as sequence length increases, while Lobster



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

Figure 8.7: The speedup of Lobster across different RNA sequence lengths relative to Scallop.

scales much more gracefully, parsing all 475 sequences in 6 minutes. Speedups of 100× are easily achieved on the median RNA sequence length of 120. This efficiency gain is enabled by mapping a significant fragment of Scallop to the GPU programming paradigm, which entails careful design decisions involving how to represent relations, how to parallelize relational operators, and how to schedule computation.

8.4 Related Work

Traditional methods for RNA secondary structure prediction Early computational efforts for RNA secondary structure prediction were dominated by thermodynamic, energy-based models. These methods rely on the principle of free energy minimization, assuming that the most stable structure of an RNA molecule is the one with the minimum free energy (MFE) under thermodynamic equilibrium (Zuker and Stiegler, 1981; Mathews et al., 1999). Algorithms such as those implemented in RNAfold (Lorenz et al., 2011), RNAstructure (Reuter and Mathews, 2010), and LinearFold (Huang et al., 2019) use dynamic programming and the nearest-neighbor model (Turner and Mathews, 2010), which decomposes structures into energetically parameterized motifs like hairpins, bulges, and multiloops. While these approaches are fast, well-established, and effective for relatively simple and short RNAs, they face limitations when handling long-range interactions, pseudoknots, and novel RNAs (Rivas and Eddy, 1999). Their accuracy is fundamentally bound by the quality and completeness of experimentally derived thermodynamic parameters, which have remained relatively static over the past decade (Kierzek et al., 2022; Szabat et al., 2022).

Deep learning methods for RNA secondary structure prediction To overcome the limitations of energy-based models, machine learning (ML) and deep learning (DL) methods have emerged as powerful alternatives. These models learn directly from data, bypassing the need for handcrafted energy parameters or explicit evolutionary conservation. Early examples like CONTRAfold (Do et al., 2006) used probabilistic discriminative models, while more recent methods such as SPOT-RNA (Singh et al., 2019), SPOT-RNA2 (Singh et al., 2021), E2Efold (Chen et al., 2020b), UFold (Fu et al., 2022), KnotFold (Gong et al., 2024), and REDFold (Chen and Chan, 2023) leverage convolutional, recurrent, and transformer-based neural networks to predict base pairing probabilities or contact maps. These neural methods show strong performance across diverse benchmarks, especially in capturing noncanonical base pairs, pseudoknots, and long-range dependencies. However, their performance often drops on out-of-distribution sequences, especially in low-data regimes, due to overfitting and limited generalization—a known challenge in bioinformatics when working with scarce, biased RNA structural data (Justyna et al., 2023; Zablocki et al., 2025).

Hybrid approaches for RNA secondary structure prediction In response to the complementary strengths and weaknesses of symbolic and neural methods, hybrid approaches have gained traction. These methods combine data-driven learning with biophysically grounded models to improve both accuracy and robustness. For example, MXFold (Akiyama et al., 2017) integrates thermodynamic scoring with structured SVMs, while MXFold2 (Sato et al., 2021) replaces the SVM with a deep neural network, achieving superior performance through thermodynamic regularization. Foundation models like RNA-FM (Chen et al., 2022a), RNAErnie (Wang et al., 2024b), and MOEFold2D (Qiu, 2025) go a step further by incorporating pre-training on massive RNA sequence corpora with fine-tuning for structure prediction, often using symbolic parsing or thermodynamic modules to guide decoding, similar to ScallopFold. These systems embody a form of neurosymbolic reasoning, balancing flexibility with interpretability, and represent a promising direction for generalizing across RNA families and sequence lengths. Their success illustrates the growing importance of combining biological priors with data-intensive learning, especially in domains constrained by experimental data availability (Zablocki et al., 2025).

CHAPTER 9

CONCLUSIONS

This dissertation presented Scallop, a programming language and system designed to support the development of neurosymbolic applications—where logical reasoning meets statistical learning. Through the design, implementation, and evaluation of Scallop, we explored how symbolic logic programming can be harmonized with neural components to build interpretable, expressive, and trainable systems across diverse domains.

We began by introducing the language constructs of Scallop and its reasoning backend, grounded in a unified provenance framework based on provenance semiring. These constructs not only generalize traditional logic programming but also enable differentiable and probabilistic reasoning, opening the door to learning from examples, labels, and gradients. We extended the system with a foreign interface for neural and foundation models, allowing Scallop programs to interoperate with large pre-trained foundation models in a modular and controllable fashion.

To evaluate the practicality and expressiveness of Scallop, we benchmarked it on a suite of classical and modern neurosymbolic tasks. These experiments demonstrated Scallop's flexibility in combining symbolic rules with model-based perception, while preserving performance and interpretability.

We then explored three advanced applications that highlight Scallop's potential to handle complex, real-world problems. In computer vision, Scallop provided a clean abstraction for scene graph generation over videos by incorporating symbolic structure into spatio-temporal reasoning. In cybersecurity, we introduced IRIS-Scallop, a neurosymbolic static analysis framework that combines LLM-assisted taint specification inference with symbolic reachability analysis. In bioinformatics, we demonstrated how Scallop supports efficient and extensible parsing for RNA secondary structure prediction by expressing grammar-based logic rules.

Across all these efforts, a recurring theme is the power of composability: Scallop allows symbolic rules, data abstractions, and neural components to be composed in a unified programming model. This

composability enables rapid experimentation and customization across domains with fundamentally different reasoning demands.

Ultimately, this work takes a step toward a broader vision: making neurosymbolic programming programmable. That is, programmable, scalable, and applicable. Scallop provides a foundation for building systems that reason, learn, and generalize in ways that neither symbolic nor neural approaches can achieve alone.

9.1 Limitations

While Scallop offers a powerful framework for neurosymbolic programming, several limitations remain. Its support for numerical values is limited, restricting applicability to tasks like low-level robotics control that require continuous reasoning. The current design of tag-symbol interaction is application-specific and lacks a generalized, systematic mechanism for bridging the neural and symbolic world. Scallop also does not yet support direct fine-tuning of modern neural models, such as LLMs and Diffusion Models, which limits its learning capabilities in more complex scenarios. On the systems side, scalability remains a challenge, particularly as the provenance engine must be extended to support broader learning paradigms. Finally, Scallop requires more infrastructure, including libraries, training pipelines, and tooling, to support the growing ecosystem of neurosymbolic building-blocks such as neurosymbolic program synthesis.

9.2 Future Work

Looking ahead, we envision Scallop as a foundation for the next generation of programmable intelligence systems that can not only learn from data but also reason with structure, generalize with abstraction, and adapt through interaction. To realize this vision, several exciting directions lie ahead.

First, we aim to extend Scallop's reasoning capabilities to continuous and real-time domains, enabling applications in robotics, control, and scientific modeling. This will require new abstractions

that seamlessly integrate symbolic logic with differentiable dynamics and real-valued computation. Another key frontier is the development of a unified interface between neural representations and symbolic programs. Generalizing the communication between tags and symbolic values will allow models to fluidly combine perception, language, and reasoning in a way that mirrors human cognition. As foundation models continue to evolve, Scallop can serve as a scaffold for training and fine-tuning large neural components with structured supervision, enabling agentic and embodied systems that are grounded, controllable, and interpretable.

Ultimately, Scallop is a step toward a broader goal: to empower developers, scientists, and researchers with tools that unify logic and learning—not as separate paradigms, but as complementary dimensions of intelligent behavior. We believe such systems will be essential in tackling the most ambitious challenges in AI, from scientific discovery to trustworthy autonomy.

BIBLIOGRAPHY

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Pearson, 1st edition, 1994.

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

Manato Akiyama, Kengo Sato, and Yasubumi Sakakibara. A max-margin training of rna secondary structure prediction integrated with the thermodynamic model. *Bioinformatics*, 33(22):3373–3379, 2017.

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

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

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

Gianluca Apriceno, Andrea Passerini, and Luciano Serafini. A neuro-symbolic approach for real-world event recognition from weak supervision. In *29th International Symposium on Temporal Representation and Reasoning*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

Sarah M Assmann, Hong-Li Chou, and Philip C Bevilacqua. Rock, scissors, paper: How rna structure informs function. *The Plant Cell*, 35(6):1671–1707, 2023.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming*, 2016.

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

Paul Biberstein, Ziyang Li, Joseph Devietti, and Mayur Naik. Lobster: A gpu-accelerated framework for neurosymbolic programming, 2025.

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

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

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

Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. Improving Java deserialization gadget chain mining via overriding-guided object generation. In *International Conference on Software Engineering*, 2023.

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

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48: 3280–3296, 2020.

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

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

Mark Chavira and Adnan Darwiche. Compiling bayesian networks with local structure. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1306–1312. Morgan Kaufmann, 2003.

Checker Framework, 2024. https://checkerframework.org/.

Chun-Chi Chen and Yi-Ming Chan. Redfold: accurate rna secondary structure prediction using residual encoder-decoder network. *BMC bioinformatics*, 24(1):122, 2023.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021.

Siyu Chen, Jian Liu, Yifan Gong, Jiarui Wang, Qiang Zhang, Dong Zhan, Min Jiang, and Shuaicheng Wang. Rna-ernie: A pre-trained language model for rna understanding. *Bioinformatics*, 38(22): 5094–5100, 2022b.

Xinshi Chen, Yu Li, Ramzan Umarov, Xin Gao, and Le Song. Rna secondary structure prediction by learning unrolled algorithms, 2020a.

Xinshi Chen, Yu Li, Ramzan Umarov, Xin Gao, and Le Song. Rna secondary structure prediction by learning unrolled algorithms. In *International Conference on Learning Representations*, 2020b. URL https://openreview.net/forum?id=S1eALyrYDH.

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

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 24185–24198, 2024.

Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *International Symposium on Software Testing and Analysis*, 2022.

Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. Scalable taint specification inference with big code. In *Conference on Programming Language Design and Implementation*, 2019.

Petr Cintula, Petr Hájek, and Carles Noguera. *Handbook of Mathematical Fuzzy Logic*, volume 1. College Publications, 2011.

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

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

Code Checker, 2023. https://github.com/Ericsson/codechecker.

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

CPPCheck, 2023. https://cppcheck.sourceforge.io/.

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

Dima Damen, Hazel Doughty, Giovanni Maria Farinella, Sanja Fidler, Antonino Furnari, Evangelos Kazakos, Davide Moltisanti, Jonathan Munro, Toby Perrett, Will Price, and Michael Wray. Scaling egocentric vision: The epic-kitchens dataset. In *European Conference on Computer Vision (ECCV)*, 2018.

Dima Damen, Hazel Doughty, Giovanni Maria Farinella, Antonino Furnari, Jian Ma, Evangelos Kazakos, Davide Moltisanti, Jonathan Munro, Toby Perrett, Will Price, and Michael Wray. Rescaling egocentric vision: Collection, pipeline and challenges for epic-kitchens-100. *International Journal of Computer Vision (IJCV)*, 130:33–55, 2022.

Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. Semiring provenance for fixed-point logic. In *Conference on Computer Science Logic (CSL)*, 2021. doi: 10.4230/LIPIcs.CSL.2021.17.

Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-143.

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

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Li Ding and Chenliang Xu. Weakly-supervised action segmentation with iterative soft boundary assignment. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6508–6516, 2018.

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

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

Chuong B. Do, Daniel A. Woods, and Serafim Batzoglou. Contrafold: Rna secondary structure prediction without physics-based models. *Bioinformatics*, 22(14):e90–e98, 07 2006. ISSN 1367-4803.

Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2015.

Saikat Dutta, Diego Garbervetsky, Shuvendu K Lahiri, and Max Schäfer. Inspectjs: Leveraging code similarity and user-feedback for effective taint specification inference for Javascript. In *International Conference on Software Engineering: Software Engineering in Practice (SEIP) Track*, 2022.

Nikita Dvornik, Isma Hadji, Konstantinos G. Derpanis, Animesh Garg, and Allan D. Jepson. Drop-dtw: Aligning common signal between sequences while dropping outliers. *CoRR*, abs/2108.11996, 2021.

Debidatta Dwibedi, Yusuf Aytar, Jonathan Tompson, Pierre Sermanet, and Andrew Zisserman. Temporal cycle-consistency learning. *CoRR*, abs/1904.07846, 2019.

Thomas E Edwards, Daniel J Klein, and Adrian R Ferré-D'Amaré. Riboswitches: small-molecule recognition by gene regulatory rnas. *Current opinion in structural biology*, 17(3):273–279, 2007.

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

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

FB Infer, 2023. https://fbinfer.com/.

FlawFinder, 2023. https://dwheeler.com/flawfinder.

Laiyi Fu, Yingxin Cao, Jie Wu, Qinke Peng, Qing Nie, and Xiaohui Xie. Ufold: fast and accurate rna secondary structure prediction with deep learning. *Nucleic acids research*, 50(3):e14–e14, 2022.

Michael Fu and Chakkrit Tantithamthavorn. LineVul: A transformer-based line-level vulnerability prediction. In *International Conference on Mining Software Repositories*, 2022.

Tsu-Jui Fu, Linjie Li, Zhe Gan, Kevin Lin, William Yang Wang, Lijuan Wang, and Zicheng Liu. Violet: End-to-end video-language transformers with masked visual-token modeling. *arXiv preprint arXiv:2111.12681*, 2021.

Xiang-Dong Fu. Non-coding rna: a new frontier in regulatory biology. *National science review*, 1(2):

190–204, 2014.

Laura R Ganser, Megan L Kelly, Daniel Herschlag, and Hashim M Al-Hashimi. The roles of structural dynamics in the cellular functions of rnas. *Nature reviews Molecular cell biology*, 20(8):474–489, 2019.

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

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

Ross B. Girshick. Fast R-CNN, 2015.

GitHub. Github advisory database, 2024a. https://github.com/advisories.

GitHub. Github security advisories, 2024b. https://github.com/github/advisory-database.

Tiansu Gong, Fusong Ju, and Dongbo Bu. Accurate prediction of rna secondary structure including pseudoknots through solving minimum-cost flow with learned potentials. *Communications Biology*, 7(1):297, 2024.

Raghav Goyal, Samira Ebrahimi Kahou, Vincent Michalski, Joanna Materzynska, Susanne Westphal, Heuna Kim, Valentin Haenel, Ingo Fründ, Peter Yianilos, Moritz Mueller-Freitag, Florian Hoppe, Christian Thurau, Ingo Bax, and Roland Memisevic. The "something something" video database for learning and evaluating visual common sense. *CoRR*, abs/1706.04261, 2017.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika

Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan

Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024.

Kristen Grauman, Andrew Westbury, Eugene Byrne, Zachary Chavis, Antonino Furnari, Rohit Girdhar, Jackson Hamburger, Hao Jiang, Miao Liu, Xingyu Liu, Miguel Martin, Tushar Nagarajan, Ilija Radosavovic, Santhosh Kumar Ramakrishnan, Fiona Ryan, Jayant Sharma, Michael Wray, Mengmeng Xu, Eric Zhongcong Xu, Chen Zhao, Siddhant Bansal, Dhruv Batra, Vincent Cartillier, Sean Crane, Tien Do, Morrie Doulaty, Akshay Erapalli, Christoph Feichtenhofer, Adriano Fragomeni, Qichen Fu, Christian Fuegen, Abrham Gebreselasie, Cristina González, James Hillis, Xuhua Huang, Yifei Huang, Wenqi Jia, Weslie Khoo, Jáchym Kolár, Satwik Kottur, Anurag Kumar, Federico Landini, Chao Li, Yanghao Li, Zhenqiang Li, Karttikeya Mangalam, Raghava Modhugu, Jonathan Munro, Tullie Murrell, Takumi Nishiyasu, Will Price, Paola Ruiz Puentes, Merey Ramazanova, Leda Sari, Kiran Somasundaram, Audrey Southerland, Yusuke Sugano, Ruijie Tao, Minh Vo, Yuchen Wang, Xindi Wu, Takuma Yagi, Yunyi Zhu, Pablo Arbeláez, David Crandall, Dima Damen, Giovanni Maria Farinella, Bernard Ghanem, Vamsi Krishna Ithapu, C. V. Jawahar, Hanbyul Joo, Kris Kitani, Haizhou Li, Richard A. Newcombe, Aude Oliva, Hyun Soo Park, James M. Rehg, Yoichi Sato, Jianbo Shi, Mike Zheng Shou, Antonio Torralba, Lorenzo

Torresani, Mingfei Yan, and Jitendra Malik. Ego4d: Around the world in 3, 000 hours of egocentric video. *CoRR*, abs/2110.07058, 2021.

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

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

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

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

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

Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter learning in probabilistic databases: A least squares approach, 09 2008.

Tengda Han, Weidi Xie, and Andrew Zisserman. Temporal alignment networks for long-term video. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2906–2916, 2022.

Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *International Conference on Mining Software Repositories*, 2014.

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

Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing, Verification and Validation*, 2009.

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

David Hin, Andrey Kan, Huaming Chen, and Muhammad Ali Babar. Linevd: Statement-level

vulnerability detection using graph neural networks. In *International Conference on Mining Software Repositories*, 2022.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

De-An Huang, Li Fei-Fei, and Juan Carlos Niebles. Connectionist temporal modeling for weakly supervised action labeling. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 137–153. Springer, 2016.

Jiani Huang, Calvin Smith, Osbert Bastani, Rishabh Singh, Aws Albarghouthi, and Mayur Naik. Generating programmatic referring expressions via program synthesis. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4495–4506. PMLR, 13–18 Jul 2020.

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

Jiani Huang, Ziyang Li, Mayur Naik, and Ser-Nam Lim. LASER: A neuro-symbolic framework for learning spatio-temporal scene graphs with weak supervision. In *The Thirteenth International Conference on Learning Representations*, 2025.

Liang Huang, He Zhang, Dezhong Deng, Kai Zhao, Kaibo Liu, David A Hendrix, and David H Mathews. Linearfold: linear-time approximate rna folding by 5'-to-3'dynamic programming and beam search. *Bioinformatics*, 35(14):i295–i304, 2019.

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

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, and Bo Zheng. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, September 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Jingwei Ji, Ranjay Krishna, Li Fei-Fei, and Juan Carlos Niebles. Action genome: Actions as compositions of spatio-temporal scene graphs. In *Proceedings of the IEEE/CVF conference on*

computer vision and pattern recognition, pages 10236–10247, 2020.

Jingwei Ji, Rishi Desai, and Juan Carlos Niebles. Detecting human-object relationships in videos. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 8106–8116, 2021.

Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc V. Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision. *CoRR*, abs/2102.05918, 2021.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-Bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, 2013.

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

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

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with LLMs. In *AAAI Conference on Artificial Intelligence*, 2023.

Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *International Static Analysis Symposium*, 2005.

Marek Justyna, Maciej Antczak, and Marta Szachniuk. Machine learning for rna 2d structure prediction benchmarked on experimental data. *Briefings in Bioinformatics*, 24(3):bbad153, 2023.

Hong Jin Kang, Khai Loong Aw, and David Lo. Detecting false alarms from automatic static analysis tools: How far are we? In *International Conference on Software Engineering*, 2022.

Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, et al. The kinetics human action video dataset. *arXiv preprint arXiv:1705.06950*, 2017.

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

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169*, 2023.

Elzbieta Kierzek, Xiaoju Zhang, Richard M Watson, Scott D Kennedy, Marta Szabat, Ryszard Kierzek, and David H Mathews. Secondary structure prediction for rna sequences including n6-methyladenosine. *Nature communications*, 13(1):1271, 2022.

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

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

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

Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper R. R. Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. The open images dataset V4: unified image classification, object detection, and visual relationship detection at scale. *CoRR*, abs/1811.00982, 2018.

Luigi Lavazza, Davide Tosi, and Sandro Morasca. An empirical study on the persistence of spotbugs issues in open-source software evolution. In *Quality of Information and Communications Technology*, 2020.

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

Jae Hee Lee, Michael Sioutis, Kyra Ahrens, Marjan Alirezaie, Matthias Kerzel, and Stefan Wermter. Neuro-symbolic spatio-temporal reasoning. In *Compendium of Neurosymbolic Artificial Intelligence*, pages 410–429. IOS Press, 2023.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International Conference on Software Engineering*, 2023.

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

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2024a.

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

Junnan Li, Ramprasaath R. Selvaraju, Akhilesh Deepak Gotmare, Shafiq R. Joty, Caiming Xiong, and Steven C. H. Hoi. Align before fuse: Vision and language representation learning with momentum distillation. *CoRR*, abs/2107.07651, 2021a.

Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. Comparison and evaluation on static application security testing (SAST) tools for Java. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023a.

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

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

Xinghang Li, Di Guo, Huaping Liu, and Fuchun Sun. Embodied semantic scene graph generation. In Aleksandra Faust, David Hsu, and Gerhard Neumann, editors, *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 1585–1594. PMLR, 08–11 Nov 2022b.

Yi Li, Shaohua Wang, and Tien Nhut Nguyen. Vulnerability detection with fine-grained interpretations. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021b.

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

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19:2244–2258, 2021c.

Zhuguo Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19:2821–2837, 2020b.

Ziang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities, 2025.

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

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

Ziyang Li, Jiani Huang, Jason Liu, and Mayur Naik. *Neurosymbolic Programming in Scallop: Principles and Practice*, volume 8 of *Foundations and Trends in Programming Languages*. now Publishers, 2024b. doi: 10.1561/2500000059.

Ziyang Li, Jiani Huang, Jason Liu, Felix Zhu, Eric Zhao, William Dodds, Neelay Velingker, Rajeev Alur, and Mayur Naik. Relational programming with foundational models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(9):10635–10644, Mar. 2024c.

Nanhao Liang, Yong Liu, Wenfang Sun, Yingwei Xia, and Fan Wang. Ckt-rcm: Clip-based knowledge transfer and relational context mining for unbiased panoptic scene graph generation. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3570–3574. IEEE, 2024.

Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *International Symposium on Software Testing and Analysis*, 2022.

Hengyue Liu, Ning Yan, Masood Mortazavi, and Bir Bhanu. Fully convolutional scene graph generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11546–11556, June 2021.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Qing Jiang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, and Lei Zhang. Grounding dino: Marrying dino with grounded pre-training for open-set object detection, 2024.

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

Benjamin Livshits, Aditya V. Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Conference on Programming Language Design and Implementation*, 2009.

Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. Viennarna package 2.0. *Algorithms for molecular biology*, 6:1–14, 2011.

Cewu Lu, Ranjay Krishna, Michael S. Bernstein, and Li Fei-Fei. Visual relationship detection with

language priors. *CoRR*, abs/1608.00187, 2016.

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

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

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298, 2021. doi: 10.1016/j.artint.2021.103504.

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

Joanna Materzynska, Tete Xiao, Roei Herzig, Huijuan Xu, Xiaolong Wang, and Trevor Darrell. Something-else: Compositional action recognition with spatial-temporal interaction networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1049–1059, 2020.

David H Mathews, Jeffrey Sabina, Michael Zuker, and Douglas H Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of rna secondary structure. *Journal of molecular biology*, 288(5):911–940, 1999.

David H Mathews, Walter N Moss, and Douglas H Turner. Folding and finding rna secondary structure. *Cold Spring Harbor perspectives in biology*, 2(12):a003665, 2010.

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

Antoine Miech, Jean-Baptiste Alayrac, Lucas Smaira, Ivan Laptev, Josef Sivic, and Andrew Zisserman. End-to-end learning of visual representations from uncurated instructional videos. *CoRR*, abs/1912.06430, 2019.

Toki Migimatsu and Jeannette Bohg. Grounding predicates through actions. In *2022 International Conference on Robotics and Automation*, pages 3498–3504. IEEE, 2022.

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

Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel.

Learning reasoning strategies in end-to-end differentiable proving. In *ICML*, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

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

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

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

Michael O'Connell, Guanya Shi, Xichen Shi, Kamyar Azizzadenesheli, Anima Anandkumar, Yisong Yue, and Soon-Jo Chung. Neural-fly enables rapid learning for agile flight in strong winds. *Science Robotics*, 7(66), May 2022. ISSN 2470-9476.

OpenAI. Chatgpt plugins, 2023a. URL https://openai.com/index/chatgpt-plugins/. Accessed: 2024-10-27.

OpenAI. Gpt-4 technical report, 2023b.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich,

Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *International Conference on Software Engineering*, 2024.

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

Xiangyun Qiu. Robust rna secondary structure prediction with a mixture of deep learning and physics-based experts. *Biology Methods and Protocols*, 10(1):bpae097, 2025.

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

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

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

Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. Sam 2: Segment anything in images and videos, 2024.

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

Joris Renkens, Guy Van den Broeck, and Siegfried Nijssen. k-optimal: A novel approximate inference algorithm for ProbLog. *Machine Learning*, 89(3):215–231, July 2012.

Jessica S Reuter and David H Mathews. Rnastructure: software for rna secondary structure prediction and analysis. *BMC bioinformatics*, 11:1–9, 2010.

Alexander Richard, Hilde Kuehne, Ahsan Iqbal, and Juergen Gall. Neuralnetwork-viterbi: A framework for weakly supervised video learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7386–7395, 2018.

Elena Rivas and Sean R Eddy. A dynamic programming algorithm for rna structure prediction including pseudoknots. *Journal of molecular biology*, 285(5):2053–2068, 1999.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965. ISSN 0004-5411. doi: 10.1145/321250.321253. URL https://doi.org/10.1145/321250.321253.

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

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

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

Kengo Sato, Manato Akiyama, and Yasubumi Sakakibara. Rna secondary structure prediction using deep learning with thermodynamic integration. *Nature Communications*, 12(1):941, 2021. ISSN 2041-1723. doi: 10.1038/s41467-021-21194-4.

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

use tools, 2023.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *International Conference on Compiler Construction (CC)*, 2016.

Semgrep. The Semgrep platform. https://semgrep.dev/, 2023.

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

Xindi Shang, Donglin Di, Junbin Xiao, Yu Cao, Xun Yang, and Tat-Seng Chua. Annotating objects and relations in user-generated videos. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 279–287. ACM, 2019.

Hikaru Shindo, Manuel Brack, Gopika Sudhakaran, Devendra Singh Dhami, Patrick Schramowski, and Kristian Kersting. Deisam: Segment anything with deictic prompting. *arXiv preprint arXiv:2402.14123*, 2024.

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

Jaswinder Singh, Jack Hanson, Kuldip Paliwal, and Yaoqi Zhou. Rna secondary structure prediction using an ensemble of two-dimensional deep neural networks and transfer learning. *Nature communications*, 10(1):5407, 2019.

Jaswinder Singh, Kuldip Paliwal, Tongchuan Zhang, Jaspreet Singh, Thomas Litfin, and Yaoqi Zhou. Improved rna secondary structure and tertiary base-pairing prediction using evolutionary profile, mutational coupling and two-dimensional transfer learning. *Bioinformatics*, 37(17):2589–2600, 2021.

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

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

Snyk.io, 2024. https://snyk.io.

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

SonarQube, 2024. https://www.sonarsource.com/products/sonarqube.

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

Robert C Spitale and Danny Incarnato. Probing the dynamic rna structurome and its functions. *Nature Reviews Genetics*, 24(3):178–196, 2023.

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

Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. *arXiv preprint arXiv:2212.08108*, 2023.

Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 2024.

Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 7464–7473, 2019.

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

Yu Sun, Qian Bao, Wu Liu, Tao Mei, and Michael J. Black. Trace: 5d temporal regression of avatars with dynamic cameras in 3d environments. *arXiv preprint arXiv:2306.02850*, 2023.

Marta Szabat, Martina Prochota, Ryszard Kierzek, Elzbieta Kierzek, and David H Mathews. A test and refinement of folding free energy nearest neighbor parameters for rna including n6-methyladenosine. *Journal of Molecular Biology*, 434(18):167632, 2022.

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

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

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur,

Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Sebastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. Gemma 2: Improving open language models at a practical size, 2024.

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

Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 59(2):64–73, 2016.

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

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

Douglas H Turner and David H Mathews. Nndb: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic acids research*, 38(suppl_1): D280–D282, 2010.

Guy Van den Broeck, Wannes Meert, and Adnan Darwiche. Skolemization for weighted first-order model counting. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1319–1325. AAAI Press, 2013.

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

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. LLMDFA: Analyzing dataflow in code with large language models. In *Neural Information Processing Systems*, 2024a.

Ning Wang, Jiang Bian, Yuchen Li, Xuhong Li, Shahid Mumtaz, Linghe Kong, and Haoyi Xiong. Multi-purpose rna language modelling with motif-aware pretraining and type-guided fine-tuning. *Nature Machine Intelligence*, pages 1–10, 2024b.

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

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards, 1989.

Wenhao Wu, Zhun Sun, and Wanli Ouyang. Revisiting classifier: Transferring vision-language models for video recognition, 2023a.

Wenhao Wu, Xiaohan Wang, Haipeng Luo, Jingdong Wang, Yi Yang, and Wanli Ouyang. Bidirectional cross-modal knowledge exploration for video recognition with pre-trained vision-language models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6620–6630, 2023b.

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

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *International Conference on Software Engineering*, 2023.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all:

Universal fuzzing with large language models. In *International Conference on Software Engineering*, 2024.

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

Hu Xu, Gargi Ghosh, Po-Yao Huang, Dmytro Okhonko, Armen Aghajanyan, Florian Metze, Luke Zettlemoyer, and Christoph Feichtenhofer. Videoclip: Contrastive pre-training for zero-shot video-text understanding. *arXiv preprint arXiv:2109.14084*, 2021.

Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss function for deep learning under weak supervision. In *Neural Information Processing Systems*, 2017.

Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International Conference on Machine Learning (ICML)*, 2018.

Li Xu, Haoxuan Qu, Jason Kuen, Jiuxiang Gu, and Jun Liu. Meta spatio-temporal debiasing for video scene graph generation. *arXiv preprint arXiv:2207.11441*, 2022a.

Ziwei Xu, Yogesh S Rawat, Yongkang Wong, Mohan Kankanhalli, and Mubarak Shah. Don't pour cereal into coffee: Differentiable temporal logic for temporal action segmentation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022b.

Aidan ZH Yang, Ruben Martins, Claire Le Goues, and Vincent J Hellendoorn. Large language models for test-free fault localization. *arXiv preprint arXiv:2310.01726*, 2023a.

Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. Graph r-cnn for scene graph generation. In *Proceedings of the European conference on computer vision*, pages 670–685, 2018a.

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

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

Yuan Yao, Ao Zhang, Xu Han, Mengdi Li, Cornelius Weber, Zhiyuan Liu, Stefan Wermter, and Maosong Sun. Visual distant supervision for scene graph generation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15816–15826, October 2021.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-

symbolic vqa: Disentangling reasoning from vision and language understanding. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

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

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

Zhou Yu, Dejing Xu, Jun Yu, Ting Yu, Zhou Zhao, Yueting Zhuang, and Dacheng Tao. Activitynet-qa: A dataset for understanding complex web videos via question answering. *CoRR*, abs/1906.02467, 2019.

Luciano I Zablocki, Leandro A Bugnon, Matias Gerard, Leandro Di Persia, Georgina Stegmayer, and Diego H Milone. Comprehensive benchmarking of large language models for rna secondary structure prediction. *Briefings in Bioinformatics*, 26(2):bbaf137, 2025.

Shay Zakov, Yoav Goldberg, Michael Elhadad, and Michal Ziv-ukelson. Rich parameterization improves rna structure prediction. *Journal of Computational Biology*, 18(11):1525–1542, 2011. doi: 10.1089/cmb.2011.0184. PMID: 22035327.

Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. Sigmoid loss for language image pre-training, 2023.

Yong Zhang, Yingwei Pan, Ting Yao, Rui Huang, Tao Mei, and Chang-Wen Chen. Learning to generate language-supervised and open-vocabulary scene graph using pre-trained visual-semantic space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2915–2924, 2023.

Yuanhan Zhang, Jinming Wu, Wei Li, Bo Li, Zejun Ma, Ziwei Liu, and Chunyuan Li. Video instruction tuning with synthetic data, 2024.

Yaqin Zhou, Shangqing Liu, J. Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Neural Information Processing Systems*, 2019.

Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. TYGR: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1.

Guangming Zhu, Liang Zhang, Youliang Jiang, Yixuan Dang, Haoran Hou, Peiyi Shen, Mingtao Feng, Xia Zhao, Qiguang Miao, Syed Afaq Ali Shah, et al. Scene graph generation: A comprehensive

survey. *arXiv preprint arXiv:2201.00443*, 2022.

Michael Zuker and Patrick Stiegler. Optimal computer folding of large rna sequences using thermodynamics and auxiliary information. *Nucleic acids research*, 9(1):133–148, 1981.