International Journal of Approximate Reasoning 179 (2025) 109369

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

journal homepage: www.elsevier.com/locate/ijar

NeST: The neuro-symbolic transpiler

Viktor Pfanschilling [a,c,*], Hikaru Shindo [a], Devendra Singh Dhami [d], Kristian Kersting [a,b,c]

[a] Computer Science Department, Technical University of Darmstadt, Hochschulstr. 1, 64289, Darmstadt, Germany
[b] The Hessian Center for Artificial Intellifence (hessian.AI), Landwehrstrasse 50a, 64293, Darmstadt, Germany
[c] German Research Center for Artificial Intelligence (DFKI), Landwehrstrasse 50a, 64293, Darmstadt, Germany
[d] Department of Mathematics and Computer Science, Eindhoven University of Technology, Metaforum, Room 7.142, 5612 AZ, Eindhoven, Netherlands

ARTICLE INFO

Keywords:
Probabilistic programming
Neuro-symbolic
Large language models
Tractable probabilistic models

ABSTRACT

Tractable Probabilistic Models such as Sum-Product Networks are a powerful category of models that offer a rich choice of fast probabilistic queries. However, they are limited in the distributions they can represent, e.g., they cannot define distributions using loops or recursion. To move towards more complex distributions, we introduce a novel neurosymbolic programming language, Sum Product Loop Language (SPLL), along with the Neuro-Symbolic Transpiler (NeST). SPLL aims to build inference code most closely resembling Tractable Probabilistic Models. NeST is the first neuro-symbolic transpiler—a compiler from one high-level language to another. It generates inference code from SPLL but natively supports other computing platforms, too. This way, SPLL can seamlessly interface with e.g. pretrained (neural) models in PyTorch or Julia. The result is a language that can run probabilistic inference on more generalized distributions, reason on neural network outputs, and provide gradients for training.

1. Introduction

Sum-Product Networks (SPNs) are a deep learning architecture that support exact tractable probabilistic inference of a wide range of probabilistic queries [1]. Accordingly, they can be used to build efficient models of complex distributions. In particular, SPNs support efficient marginalization and conditional queries. Their probabilistic nature is useful when reasoning about uncertainty of the underlying distribution.

Although quite successful, unfortunately, SPNs lack the capability of handling control structures such as for and while loops, featured in probabilistic programming languages. The core idea of probabilistic programming languages is that writing down a generative description of a stochastic system is often easier, while developing a probabilistic inference strategy can be difficult and is often a custom effort for each probabilistic model. Probabilistic programming languages aim to make probabilistic inference easier for the user by encoding within the language all the mechanisms necessary for probabilistic inference. The user merely writes down how to sample from a distribution, while inference is done by the language. In this way, the developers of the language perform the difficult task of developing inference strategies once, and users only need to write the more straightforward generative description. To illustrate this, consider Fig. 1—for most purposes, the programmatic, generative description is more intuitive to reason about. From

* Corresponding author.
E-mail address: viktor.pfanschilling@cs.tu-darmstadt.de (V. Pfanschilling).

https://doi.org/10.1016/j.ijar.2025.109369
Received 15 February 2024; Received in revised form 18 January 2025; Accepted 18 January 2025

Available online 22 January 2025

```
A = coinflip(theta1)
if(A) {
  B = coinflip(theta2)
} else {
  B = A
}
```

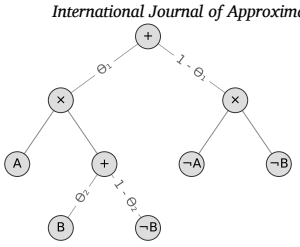

Fig. 1. A simple program with randomness (left) and a Sum-Product Network (SPN) encoding the same distribution (right). The distribution describes two booleans A and B, with B depending on A.

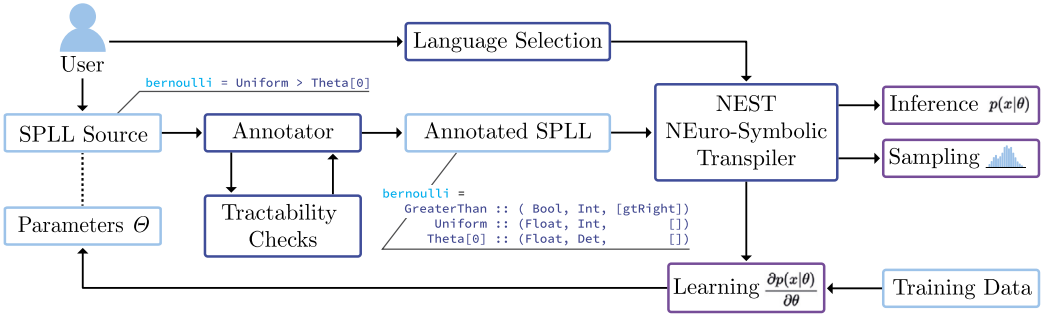

Fig. 2. An overview of NeST, showing the various modes in which it can be applied. Before transpilation, NeST checks whether the provided source program has tractable solutions. The resulting annotated representation can be transpiled to Julia or PyTorch code, or be interpreted directly. From the exposed interfaces, users can generate samples or perform probabilistic inference, i.e. compute the likelihood of a sample. Parameters in the program can be trained by maximizing the probability of a data set.

a perspective of probabilistic programming, SPNs can be seen as representing a small class of probabilistic programs that provide tractable, exact inference. Note how the example program is quite simple, while the equivalent SPN uses all essential features of SPNs. Actually, the example program in Fig. 1 has been written in the sum-product loop language (SPLL) [2]. It generalizes SPNs to more general programmatic structures while retaining the guarantees around fast inference. Recursion is of particular interest here, as either recursion or unbounded loops are a necessary component of Turing completeness. This probabilistic lens is useful for differentiating programs. A program containing hard decision boundaries and thus non-continuous behavior may not have a usable gradient. However, if it deals with uncertainties, e.g., by utilizing random number generators, then inference on the distribution is differentiable with regard to parameters of the program. In this way, SPLL can enable learning in programmatic pipelines.

Here, we expand upon SPLL by integrating neural networks and other differentiable programming methods via the Neuro-Symbolic Transpiler (NeST), see Fig. 2 for an overview. NeST builds probabilistic inference code. In order to interface easily with existing neural models, NeST transpiles (i.e. compiles from one high-level language to another) this inference code to other languages, thus integrating pre-trained models in PyTorch or Julia. NeST-generated code can also be used to train these integrated pre-trained models seamlessly or supervise new models. NeST is a significant extension of a previously published conference paper [2], showing how to make SPLL modeling more flexible. Further, we extend this previous work with a more detailed experimental study. In particular, demonstrating modular programming of large language models shows the advantages of NeST.

In summary, we make the following contributions:

1. We propose sum-product loop language (SPLL), a novel probabilistic programming language capable of handling loops in complex probabilistic code.
2. We show that SPLL is more expressive than Sum Product networks (SPNs), as it can express any SPN and can express tractable models that SPNs can not.
3. With NeST, we extend SPLL's tools to allow transpilation of SPLL into Julia and PyTorch, enabling integration with existing machine learning ecosystems.
4. In a computer vision case study, we show supervision and training of Neural Networks via SPLL.
5. In a natural language processing case study, we show that SPLL can coordinate pre-trained transformer models in a mixture of experts.

These contributions demonstrate the advantage of neuro-symbolic transpilation. We publish the source code of the transpiler at https://github.com/vektordev/haskell-dppl.

We proceed as follows. We start by discussing related work on probabilistic modeling and neuro-symbolic AI in Section 2. Then, we will give an intuitive introduction to SPLL in Section 3, followed by a detailed description of semantics, including some extensions over the state described in previous work. Section 4 then describes how NeST transpiles SPLL into either Julia or PyTorch code,

and how differentiable connections to neural networks are built. We then build a theoretical connection to the expressiveness of Sum-Product Networks in 5. Before concluding, Section 6 validates various language features in a suite of experiments.

2. Related work

Neuro-symbolic transpilation is related to several lines of research: tractable probabilistic models, probabilistic programming, neuro-symbolic AI, and differentiable programming.

2.1. Tractable probabilistic models

Probabilistic models to perform exact tractable inference have been proposed, such as Sum-Product Network (SPNs) [1] and Probabilistic Sentential Decision Diagrams (PSDDs) [3]. Exact inference is different from approximate inference in that the probability assigned to a query is deterministic and 'aligned' with the model, while approximate inference yields either an estimated probability, or the probability of a proxy problem. These models are called *Probabilistic Circuits (PCs)* [4]. However, these models are not capable of handling structured programs that contain loops or recursion. SPLL offers the capability of tractable inference and the handling of structured programs. Sum-Product Probabilistic Language (SPPL) has been proposed as a generalization of SPNs to a programming language with an extension of symbolic expressions [5]. SPPL achieves tractable exact inference. However, it does so by restricting the user to bounded loops. This is a meaningful restriction, as some functions (e.g. the Ackermann function) can not be expressed using bounded loops, only using general recursion or unbounded loops.

2.2. Probabilistic programming

Probabilistic programming languages aim to provide a toolset for describing distributions, and then run inference on these distributions. Considering the breadth of existing PPLs, we will only highlight a few: Church [6] is an example of a probabilistic programming language with similar functional notation to SPLL. More recently, the focus of the field has shifted towards Deep Probabilistic Programming Languages (DPPLs) such as DeepProblog [7], Pyro [8] and Edward [9]. These have been developed to enable users to combine probabilistic programs with deep architectures. Generally, probabilistic programming languages use approximate methods such as variational inference or sampling-based methods. Variational inference offers a appealing tradeoff of speed, expressivity and robustness, but requires the user to define guide models that approximate the target distribution. Sampling-based methods are particularly simple and expressive, but require additional effort to ensure fast inference. Sum-Product Loop Language (SPLL) takes a different approach to inference from other DPPLs. Instead of the predominant approach of variational inference, SPLL focuses on exact tractable inference instead. Naturally, not every distribution is tractable, so this is per se detrimental to expressiveness. However, SPLL can exactly capture many interesting distributions, generalizing substantially over Sum-Product Networks, and can be easily expanded to incorporate approximate or intractable methods where needed, while keeping control in the user's hands. The goal of SPLL is to use approximate or intractable methods only as much as necessary, and use exact tractable methods to simplify the problem as much as possible. In the domain of tractable probabilistic programming, DICE [10] is a noteworthy example of a probabilistic programming language that can perform exact probabilistic program inference. However, unlike SPLL, DICE supports tractable inference only on discrete distributions and programs with bounded recursion. However, extensions of DICE exist that pursue unbounded recursion [11]. In SPLL, recursion is more restricted than that, as programs are only tractable when their inference is known to converge.

2.3. Neuro-symbolic AI

Neuro-Symbolic AI is the recent effort to combine symbolic and neural methods, thus combining their respective strengths. The resulting models should ideally be composable, generalizable, interpretable and resistant to noise. We view SPLL as one way of implementing neuro-symbolic AI. We can encode symbolic constraints or operations on the output of neural networks in SPLL to define a pipeline, and then train the pipeline end-to-end. Previous work towards integrating neural and symbolic computations exist [7,12–15]. Many existing approaches in this space work with restrictions on the computations that can be encoded, mostly working with propositional or first-order logic. SPLL expands this space to computations with control flow.

2.4. Differentiable programming

Since SPLL and, in turn, NeST can be used to pass a gradient through a program, we can also compare our work to differentiable programming techniques such as Continuous Relaxations [16,17]. They smooth over discrete control flow decisions by interpolating the outcomes, weighed by a factor that corresponds to e.g. Gaussian noise on the condition. In contrast, SPLL offers exact computations, i.e., the distribution of the inference function is guaranteed to align with that of the generator. Meanwhile, Continuous Relaxations can lead to unintended program behavior, where the act of relaxing control flow conditions can lead to e.g. impossible paths in the program becoming possible. The cost of this exactness is that some programs can not be expressed in SPLL and some others scale badly in their computational cost. Meanwhile, continuous relaxations have no restriction of expressivity resulting from tractability. Instead, restrictions arise here from interpolability of outputs. Within SPLL, continuous relaxations can be used to compute gradients for parameters in places where it is not otherwise possible - i.e., deterministic expressions. For a more in-depth discussion of modes of making programs differentiable and their consequences, see also Section 5.1.

$$\begin{aligned}
\mathcal{P} \rightarrow\ & \mathcal{P}\ \mathcal{P} \mid \mathcal{F} = \mathcal{E} \\
\mathcal{E} \rightarrow\ & \mathit{if}\ \mathcal{E}\ \mathit{then}\ \mathcal{E}\ \mathit{else}\ \mathcal{E} \mid \mathcal{E} >= \mathcal{E} \mid \mathit{theta}[\mathbb{Z}] \mid \mathit{uniform} \mid \mathit{normal} \mid \\
& \mathit{constant}\ \mathcal{V} \mid \mathcal{E} * \mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid \mathit{null} \mid \mathit{cons}\ \mathcal{E}\ \mathcal{E} \mid \mathit{Var} \mid \\
& \mathit{Lambda}\ \mathit{Var}\ \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \mathit{call}\ \mathcal{F}
\end{aligned}$$

Fig. 3. The syntax for SPLL. $\mathcal{P}$ represents top-level definitions, $\mathcal{E}$ represents expressions, $\mathcal{F}$ represents function names, Var represents variable names and $\mathcal{V}$ represents values. The production rules of the grammar describe what each nonterminal ($\mathcal{P}$, $\mathcal{E}$) can expand to. Alternative options are separated by |.

3. The core of NeST: the sum-product loop language SPLL

Let us start with an intuitive introduction to SPLL. To do so, we provide a few simple programs, along with brief explanations, and their syntax. Afterward, we will provide a more formal description of the semantics.

3.1. Examples and syntax of SPLL

Our first program will linearly transform samples from $\mathcal{N}(0,1)$ using parameters *Theta* to produce a $\mathcal{N}(\mu,\sigma)$-distribution.

```
gaussian = Normal * Theta[0] + Theta[1]
```

Parameters *Theta* can be used to represent floating-point numbers that are subject to gradient descent optimization. The user is expected to initialize these to viable values, while the training process can be used to align them with the data distribution.

Bernoulli's distributions can be represented by comparing whether a uniform random sample is greater or equal to a parameter:

```
bernoulli = Uniform >= Theta[0]
```

We can combine these concepts with list construction. This example returns either an empty list or constructs a list from a single, normally distributed float and an empty tail list (`Null`).

```
main = if Uniform >= Theta[0]
  then Cons (Normal * Theta[1] + Theta[2]) Null
  else Null
```

Building on top of that, we can build a recursive generator by generating the tail list.

```
main = if Uniform >= Theta[0]
  then Cons (Normal * Theta[1] + Theta[2]) main
  else Null
```

By using Lambda expressions, we can also construct parametric functions. The probabilistic interpretation of these is a conditional distribution $p(conditional|x)$.

```
conditional x = Cons (Normal * x + Theta[2]) Null
```

More correctly, this "desugars" to:

```
conditional = Lambda x: (...)
```

The complete syntax of SPLL is shown in Fig. 3. It borrows heavily from functional programming. The definitions for $\mathcal{V}$, *Var* and $\mathcal{F}$ are left under-specified, representing values, variables, and function names, respectively. Note that not all programs derived via this grammar are valid, as the transpiler (Section 4) will reject certain intractable programs or those with type errors.

A Note on Functional Programming: Functional programming languages tend to be expression-focused. That is, the result of a function is often given as a simple expression. Statements in the form of a variable assignment are comparatively rare, though they can be expressed in ways that are not currently covered by SPLL. Furthermore, functional languages often rely strongly on recursion as an alternative to loops. This is reflected in SPLL. As a result, specific syntax for loops is not present and is instead represented by function calls.

3.2. The semantics of SPLL

Given that we know what SPLL programs look like, we will now describe the semantics of SPLL and fill in some of the details that were left out in our previous work [2]. To this end, we will use blue typewriter font to denote program code and program variables (For interpretation of the colors in this article, the reader is referred to the web version of this article.). We describe SPLL using big-step semantics, i.e. we describe with some abstraction how results are obtained, rather than giving each individual step.

We denote generative semantics as `expr` $\Downarrow$ *value*, thus describing which results *value* an expression `expr` can evaluate to and how to compute them. The probabilistic semantics, denoted $p(\texttt{expr} \Downarrow \mathit{value}) = \mathit{probability}$, give the actual probability or density of an expression evaluating to a value. The probabilistic semantics will return a density or probability, depending on the value queried. Discrete values will of course always yield a probability, while continuously distributed ones yield densities. We have found it helpful to simply count the dimensionality of a result, with 0-dimensional measures being probability masses. Integrating out dimensions then reduces dimensionality, while multiplying measures adds their dimensionalities. For any expression, the generative semantics shall yield each possible result with the probability or density yielded by the probabilistic semantics. Both semantics should thus describe the same distribution but with the goal of making different computational tasks – sampling and inference – clearer and easier. Note that we only give rules for soundly typed programs. Unsoundly typed programs will be rejected during compilation, as will be described in Section 4.

3.3. Generative semantics: the groundwork

Firstly, we will go over **generative semantics**. The first two rules describe that the constant expression, encapsulating a value, evaluates to that value, while the second rule describes that parameters evaluate to the respective position in the parameter vector θ.

$$\frac{}{\texttt{constant x} \Downarrow x} \qquad \frac{\theta_i = x}{\texttt{theta[i]} \Downarrow x}$$

The next two rules describe the ranges of primitive distributions.

$$\frac{x \sim N(0,1)}{\texttt{normal} \Downarrow x} \qquad \frac{x \sim U(0,1)}{\texttt{uniform} \Downarrow x}$$

The next two rules describe conditional statements. These can be read as 'if the condition `c` of an if-expression evaluates to True and the expression `a` evaluates to x, then the entire expression evaluates to x.' The other branch follows symmetrically.

$$\frac{\texttt{c} \Downarrow \mathit{True} \qquad \texttt{a} \Downarrow x}{\texttt{if c then a else b} \Downarrow x} \qquad \frac{\texttt{c} \Downarrow \mathit{False} \qquad \texttt{b} \Downarrow x}{\texttt{if c then a else b} \Downarrow x}$$

The next rules cover basic arithmetic and comparison operators.

$$\frac{\texttt{a} \Downarrow x \qquad \texttt{b} \Downarrow y}{\texttt{a} * \texttt{b} \Downarrow x * y} \qquad \frac{\texttt{a} \Downarrow x \qquad \texttt{b} \Downarrow y}{\texttt{a} + \texttt{b} \Downarrow x + y} \qquad \frac{\texttt{a} \Downarrow x \qquad \texttt{b} \Downarrow y \qquad x \geq y}{\texttt{a >= b} \Downarrow \mathit{True}} \qquad \frac{\texttt{a} \Downarrow x \qquad \texttt{b} \Downarrow y \qquad x < y}{\texttt{a >= b} \Downarrow \mathit{False}}$$

The symbol `null` denotes an empty list and `cons` constructs a linked list out of a single element x and a tail list y. To distinguish values from syntax, we denote lists as [] and $(a : b)$ for empty and non-empty lists.

$$\frac{}{\texttt{null} \Downarrow []} \qquad \frac{\texttt{a} \Downarrow x \qquad \texttt{b} \Downarrow y}{\texttt{cons a b} \Downarrow (x : y)}$$

For simplicity, we have thus far ignored any notion of environments, in which local variables could exist. An environment is simply a mapping of variables to values. We will use the notation $E \vdash \texttt{expr} \Downarrow \mathit{value}$ to denote that in environment E, `expr` evaluates to *value*. Where we previously omitted environments, they can be assumed to be carried over into the evaluation of subexpressions. Lambda expressions evaluate to closures, which exist to capture the surrounding environment E for use within the expression f.

Closures are resolved when applying another expression `e2` to them. The result of that second expression is inserted as the parameter into the original lambda expression that formed the closure, while the closure gives access to any variables that were captured by it

$$\frac{}{E \vdash \texttt{Lambda x f} \Downarrow (\mathit{Closure}\ x \to f,\ E)} \qquad \frac{}{E, v \mapsto x \vdash \texttt{Var v} \Downarrow x}$$

$$\frac{E_1 \vdash \texttt{e1} \Downarrow (\mathit{Closure}\ x \mapsto f,\ E_2) \qquad E_1 \vdash \texttt{e2} \Downarrow \mathit{val} \qquad E_2,\ x \mapsto \mathit{val} \vdash \texttt{f} \Downarrow r}{E_1 \vdash \texttt{e1 e2} \Downarrow r}$$

3.4. Probabilistic semantics: the tractable case

Now we have everything at hand to tackle **probabilistic semantics**. Note that we do not claim that the probabilistic semantics are complete: Many different algorithms exist that can be employed here to solve additional cases, and the ones we show are not exhaustive. The presented ones are expressive enough to go beyond sum-product networks and build interesting control flow and reasoning. Additional solutions, however, can be integrated easily into NeST. Furthermore, in some cases, algorithms with satisfactory performance criteria might not exist even when the generative semantics are simple to define.

To describe the probabilistic semantics, please note that $a \not\Downarrow x \implies p(\texttt{a} \Downarrow x) = 0$, and we will usually only describe the nontrivial cases where $a \Downarrow x$. Furthermore, we consider $p(\texttt{e} \Downarrow x) = q$ to be the probability (density) q of result x given expression `e`, i.e. p is conditioned on `e`. Consequently, continuous and discrete x respectively are normalized as follows:

$$\int_x p(\texttt{e} \Downarrow x) dx = 1 \quad \text{or} \quad \sum_x p(\texttt{e} \Downarrow x) = 1$$

For readability, we also leave any calculation of dimensionalities implicit. Firstly, we will consider the relatively simple cases: Constants and Parameters are deterministic and thus can be described like a discrete distribution.

$$\frac{}{p(\texttt{constant x} \Downarrow x) = 1} \qquad \frac{x \neq y}{p(\texttt{constant x} \Downarrow y) = 0}$$

$$\frac{\theta_i = x}{p(\texttt{theta[i]} \Downarrow x) = 1} \qquad \frac{\theta_i \neq x}{p(\texttt{theta[i]} \Downarrow x) = 0}$$

Primitive distributions can be defined via their density functions.

$$\frac{\varphi_{N(0,1)}(x) = a}{p(\texttt{normal} \Downarrow x) = a} \qquad \frac{0 \leq x < 1}{p(\texttt{uniform} \Downarrow x) = 1} \qquad \frac{x \geq 1 \vee x < 0}{p(\texttt{uniform} \Downarrow x) = 0}$$

Conditional statements evaluate both branches, and weigh the results of both branches by the probability of the condition.

$$\frac{p(\texttt{e1} \Downarrow True) = a \qquad p(\texttt{e2} \Downarrow x) = b \qquad p(\texttt{e3} \Downarrow x) = c}{p(\texttt{if e1 then e2 else e3} \Downarrow x) = a * b + (1 - a) * c}$$

List construction will be handled as follows: Null can be regarded as a constant, so it will evaluate to the empty list with probability 1. Cons meanwhile encodes an independence assumption: The probability that a cons-expression evaluates to a list x:y is the product of the probabilities that the respective subexpressions evaluate to x and y. This independence assumption is justified, as there is no random variable that can influence both subexpressions.

$$\frac{}{p(\texttt{null} \Downarrow []) = 1} \qquad \frac{p(\texttt{e1} \Downarrow x) = a \qquad p(\texttt{e2} \Downarrow y) = b}{p(\texttt{cons e1 e2} \Downarrow (x : y)) = a * b}$$

In the following rules about lambda expressions, we enforce determinism of substeps by mandating that the respective evaluations happen with probability 1. These restrictions might not be strictly necessary, but we introduce them in the interest of simplifying inference.

$$\frac{}{p(E \vdash \texttt{Lambda v f} \Downarrow (Closure\ v \mapsto f,\ E)) = 1} \qquad \frac{}{p(E,\ v \mapsto x \vdash \texttt{Var v} \Downarrow x) = 1}$$

$$\frac{p(E \vdash \texttt{e1} \Downarrow (Closure\ var \mapsto f,\ E_c)) = 1 \qquad p(E \vdash \texttt{e2} \Downarrow x) = 1 \qquad p(E_c,\ var \mapsto x \vdash \texttt{f} \Downarrow r) = a}{p(E \vdash \texttt{e1 e2} \Downarrow r) = a}$$

Arithmetic warrants some additional explanation: In the probabilistic semantics of multiplication, we assume that one side of the operation is a deterministic expression. In that case, we can undo the multiplication by way of division. We will focus on the multiplication rule on the left, but the other case follows symmetrically. a being deterministic lets us compute y. Knowing y and $x * y$, we can deduce x. We can then compute the overall probability of the result of this multiplication. The division by y is necessary for multiplication of continuous variables to ensure proper normalization. This division is inappropriate in case of addition or in case of probabilities rather than probability densities.

$$\frac{p(\texttt{e2} \Downarrow x) = a \qquad p(\texttt{e1} \Downarrow y) = 1}{p(\texttt{e1} * \texttt{e2} \Downarrow x * y) = a/y} \qquad \frac{p(\texttt{e2} \Downarrow x) = 1 \qquad p(\texttt{e1} \Downarrow y) = a}{p(\texttt{e1} * \texttt{e2} \Downarrow x * y) = a/x}$$

$$\frac{p(\texttt{e2} \Downarrow x) = a \qquad p(\texttt{e1} \Downarrow y) = 1}{p(\texttt{e1} + \texttt{e2} \Downarrow x + y) = a} \qquad \frac{p(\texttt{e2} \Downarrow x) = 1 \qquad p(\texttt{e1} \Downarrow y) = a}{p(\texttt{e1} + \texttt{e2} \Downarrow x + y) = a}$$

One last operator we need to cover is comparisons. Again we can solve this whenever one side is constant, in which case the probability is given by an integral over the density of the other side. Cases for evaluation to False and other potential operators follow the same structure as the cases given here, which cover evaluation to True with the right and left sides being constant respectively. The rules for computation of said integrals are omitted here, as they follow from the rules of integration and the previous computation of the PDF. Such integrals are possible for arithmetic and if-then-else expressions as well as primitive distributions.

$$\frac{\int_y^\infty p(\texttt{e1} \Downarrow x)\,dx = a \qquad p(\texttt{e2} \Downarrow y) = 1}{p(\texttt{e1} >= \texttt{e2} \Downarrow True) = a} \qquad \frac{p(\texttt{e1} \Downarrow y) = 1 \qquad \int_{-\infty}^y p(\texttt{e2} \Downarrow x)\,dx = a}{p(\texttt{e1} >= \texttt{e2} \Downarrow True) = a}$$

3.5. Weighted model counting: the intractable case

In some cases, we are interested in computing probabilities even if said probability is intractable. For example, consider integer addition. We can not compute the probability $p(\texttt{a} + \texttt{b} \Downarrow x)$ for all cases of a and b in a tractable manner, as the number of possible values of a and b can be made to increase exponentially in program size. However, if a and b yield integers of known ranges, we can enumerate all possible cases that sum to x and add up their probabilities:

$$\frac{p(\texttt{e1} \Downarrow i) = a_i \qquad p(\texttt{e2} \Downarrow x - i) = b_i}{p(\texttt{e1} + \texttt{e2} \Downarrow x) = \sum_i a_i * b_i}$$

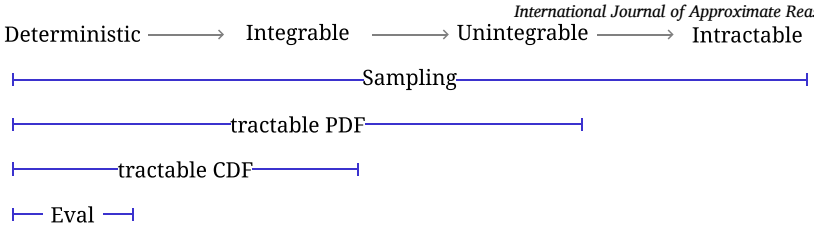

Fig. 4. SPLL Modality Lattice. The four modalities at the top describe classes of SPLL expressions. From left to right, inference becomes more restricted, as deterministic evaluation (Eval), tractable computation of the cumulative distribution function (CDF) and finally tractable computation of the probability density function (PDF) become impossible as higher-complexity language aspects are introduced into a program.

This is directly analogous to weighted model counting [18], where all assignments of values to variables that satisfy a constraint are enumerated, weighted according to their probability, and summed. Techniques for accelerating weighted model counting might also find future application here. This general procedure can be used to solve for many different operations in place of $+$.

In SPLL, a practical implementation of this requires that we know which models to count. This can be achieved by static analysis, where leaf distributions have known ranges and those ranges propagate through arithmetic expressions and conditional statements. Any discrete operation that required one deterministic operand for tractable inference can now be relaxed to merely require one side with a statically known range of values, so long as the operation is invertible. If the operation is not invertible, then we need to enumerate more models, by enumerating both sides and checking that the operation yields the desired result:

$$\frac{p(\texttt{e1} \Downarrow i) = a_i \qquad p(\texttt{e2} \Downarrow j) = b_j \qquad p(\texttt{f i j} \Downarrow x) = 1}{p(\texttt{f e1 e2} \Downarrow x) = \sum_{i,j} a_i * b_j}$$

Overall, this is an extension of the basic semantics of SPLL. The tractable basics we discussed previously take precedence, such that whenever an efficient algorithm can be found, it will also be used by NeST.

4. The neuro-symbolic transpiler NeST

Having discussed semantics, we are ready to fill most of the steps in the architecture overview previously seen in Fig. 2 with detail. Specifically, we now have everything at hand to build the neuro-symbolic transpiler NeST. We start by showing how to transpile SPLL programs in the following Section 4.1 and afterward extend it to the neuro-symbolic case in Section 4.2. The changes required to make this transpiler work well with neural models will actually be relatively minor, as long as our target language is differentiable.

4.1. Transpiling sum-product loop programs

For the sake of easy understanding, we use a simple program to explain the intermediate results and the steps taken along the way. We will also split the transpilation process into three stages:

1. **Validation**, consisting of type checking, modality checking, and static analysis,
2. **Expansion**, where the previously discussed semantics are implemented in an intermediate representation, and
3. **Translation**, where the intermediate representation is translated into another high-level language.

The big challenges in this process are soundness and semantics. Both will be made easier by relying on the semantics we described previously. In practice, we achieve this by validating the program to ensure that well-defined semantic rules exist for the program, and then expanding those rules correctly. The transpilation part at the end is comparatively straightforward. Recall the Bernoulli program from Section 3:

```
bernoulli = Uniform >= Theta[0]
```

During **validation**, NeST will initially **type check** the program and run static analyses. As a first pass, all programs that violate typing rules are rejected. In addition to regular type checking, we annotate the program with **'modalities'**, which describe which kinds of inference are possible on an expression. Since the modality of an expression naturally depends on the modality of its subexpressions, this property propagates throughout the program like type information. We use a Hindley-Milner style algorithm [19] to compute types and modalities, i.e. we collect constraints on types and modalities, until we arrive at a single most general elaboration of the program, or run into an error of contradiction or ambiguity. These constraints encode type information such as 'addition of integers accepts two integers and yields an integer', or modality information such as 'if the left operand of addition is deterministic, then the result is integrable if the right operand is integrable.' The type constraints are unchanged from those found in regular programming languages. The modality constraints correspond to the implicit assumptions about subexpressions made in the semantics.

For example, the following rule assumes that e1 is in modality deterministic (on account of it yielding y with probability 1) and that e2 is in modality unintegrable or better (as e2's PDF gets queried). In turn, the rule then yields a PDF, thus putting e1 ∗ e2 into modality unintegrable if those assumptions are met.

$$\frac{p(\texttt{e2} \Downarrow x) = a \qquad p(\texttt{e1} \Downarrow y) = 1}{p(\texttt{e1} * \texttt{e2} \Downarrow x * y) = a/y}$$

This constraint - e1 ∗ e2 being unintegrable given the above conditions - is added to the constraints pool when validating the program. Modalities exist on a **lattice** (see Fig. 4) that describes whether e.g. queries of the CDF or PDF are possible on a program. Along this lattice exists a hierarchy, where all simpler queries are automatically supported by a program that supports more complex queries. For example, a program supporting CDF queries (i.e. one in *Integrable*) implies that it must also support PDF queries. Due to the lattice structure of modalities, these constraints are often (chained) inequality constraints, e.g. that e1 ∗ e2 is of a modality no better than e2. The purpose of using constraints and resolutions is that even cyclical dependencies can be expressed, which appear in recursive programs. In resolving constraints and creating the modality annotations, we determine which algorithm is appropriate for each expression, if there are multiple choices. For example, we choose which probabilistic semantics of >= to apply, depending on which subexpression is deterministic, thus informing the modality of the compound expression. The static analyses include the range of outputs of an expression, if required for weighted model counting.

Using this system of modalities, we can for example express that Theta[0] is deterministic, Uniform can be integrated (i.e. has a CDF), and thus the probability of Uniform >= Theta[0] can be inferred via its PDF and its CDF.

Following the above Bernoulli's distribution, the type annotations will look as follows, containing types, modalities, and any further annotations. In this case, that is the choice of inference strategy for >=. This is also where ranges of outcomes will be annotated in case Weighted Model Counting is needed.

```
bernoulli =
    GreaterThan :: (Bool,  Integrable, [greaterThanRight])
        Uniform :: (Float, Integrable, [])
        Theta[0]:: (Float, Deterministic, [])
```

The second step, **expansion**, uses the annotated SPLL program to produce an intermediate representation (IR). This IR is a target language-agnostic representation of computations. During conversion to IR, we also resolve the inference strategy choice of the previous step into a concrete implementation. The result is a representation that could be interpreted as is, without additional knowledge about the probabilistic semantics of SPLL. Additionally, we introduce two different entry points here, representing generative and probabilistic semantics:

1. The first entry point samples from the underlying distribution, potentially conditioned on any function arguments.
2. The second entry point is for inference and computes the probability corresponding to a sample, likewise potentially conditioned on any function arguments.

The arguments of the sampling function match those of the SPLL code, while the inference entry point additionally accepts the queried sample. The inference function is implemented by expanding the previously selected algorithms into an implementation. This implementation is directly derived from the semantics discussed in Section 3.4.

Using the `greaterThanRight` inference rule, we can proceed in our example as follows.

$$\frac{\int_y^\infty p(\texttt{e1} \Downarrow x)\,dx = a \qquad p(\texttt{e2} \Downarrow y) = 1}{p(\texttt{e1} >= \texttt{e2} \Downarrow True) = a}$$

This rule calls for a deterministic sample of the right-hand side operand and integrates the PDF of the left-hand side with appropriate bounds. The l_3 and l_4 variables below represent the upper limit of the integral over the uniform distribution and the integral itself. The if-expression then transforms the integral to reflect whether the sample was given as true or false. The below IR implementation of Bernoulli's distribution was edited for readability, as the IR notation is not intended for human interaction.

```
bernoulli_prob sample =
    let l_3_high =
        if 1 > Theta[0] then Theta[0] else 1
    let l_4_lhs_integral =
        if 0 > l_3_high then 0 else l_3_high - 0
    in if sample == True
        then 1 - l_4_lhs_integral
        else l_4_lhs_integral
bernoulli_gen = Uniform >= Theta[0]
```

Finally, we **translate** the IR representation into code of the target language. The IR code from the last step already makes the complexities of inference explicit, so this last step is just a matter of translating the IR code to the target language. Since the IR is merely a straightforward computation without additional semantics, this is quite straightforward. By deferring target language specific decisions until now, we can add other languages relatively simply, by adding a code generator from this IR to the new target language. Code generators for PyTorch and Julia have been implemented. Shown here is the resulting PyTorch implementation.

```
class Main(Module):
    def forward(self, sample):
        if (1.0 >= self.thetas[0]):
            l_3_high = self.thetas[0]
        else:
            l_3_high = 1.0
        if (0.0 >= l_3_high):
            l_4_lhs_integral = 0.0
        else:
            l_4_lhs_integral = (l_3_high - 0.0)
        if (true == sample):
            return (1.0 - l_4_lhs_integral)
        else:
            return l_4_lhs_integral

    def generate(self):
        return (rand() >= self.thetas[0])
```

Note that parameters θ are owned by the module here, changing the way parameters are shared. This can be adjusted easily to make the parameter vector θ shared globally. A Julia translation can be found in Appendix A.

One major feature that this example did not cover is that of recursive programs. In such cases, NeST generates a recursive inference strategy. A more in-depth discussion of termination and tractability of this inference follows in Section 5.3

4.2. Extension to the neural case

A major goal of the Neuro-Symbolic Transpiler NeST that we have not covered so far is the combination of neural networks (or differentiable programs in general) with SPLL. For this, we have to be a little bit more careful and make some additions to SPLL to ensure meaningful gradients propagate to the neural network:

1. We will consider what we want such a conjunction to do: We want neural networks to produce symbolic variables, which SPLL can then reason about. In order to make this conform with existing SPLL constructs, we require the outputs of neural networks to be distributional, such as when using a softmax output layer on a classifier. The neural architecture is then essentially forward-declared in SPLL and fully implemented by the user in the target language. From there, the probabilistic semantics of a neural network call is exactly the output of the network, assigning a probability to each outcome. Consequently, sampling from this distribution is just sampling from the categorical distribution encoded by the softmax output.
2. It is essential that gradients from training the SPLL program are passed on to the neural network.
3. The computation of the probability function needs to seamlessly integrate with any attached neural networks, such that gradients of the probability function can be used to train the neural network. This informs the choice of target languages we support for the transpilation presented in Section 4, as both strong neural network ecosystems and expressive automatic differentiation tools make this integration technically simpler.
4. Furthermore, since we are not always interested in learning the joint probability distribution of a data set, we introduce conditional distributions. Where a parameterless SPLL function models a joint distribution of all its outputs, a parametric function models a conditional distribution. When generating a sample, these parameters are considered as given, much like in a conditional distribution. For example, the task of MNIST-addition can be implemented as

`mnistAdd img1 img2 = read img1 + read img2`,

encoding the distribution $p(sum|img1, img2)$ - the complete joint probability of $p(sum, img1, img2)$ is not modeled.

With this, neural networks can directly be supported within SPLL and in turn NeST as long as everything is differentiable. In order to technically facilitate this integration, we expect the user to provide a forward declaration of the neural network's interface, i.e. its input and outputs. This can then be used to ensure the type soundness of the program and to ensure the output of such networks is enumerated properly by Weighted Model Counting. With soundness ensured, neural network integration is as simple as calling the specified neural network and transforming outputs as appropriate.

5. Faithful differentiability and model expressiveness

5.1. On the differentiability of programs

NeST is, among other things, motivated by a desire to pass gradients through arbitrary programs. This is useful for neuro-symbolic AI in various ways: For one, when a known algorithm can be used to generate the output, a preceding neural network can be trained to simply handle input transformation and does not need to deal with the often-times complicated control flow needed to replicate the algorithm. As an example, consider pathfinding based on image data: By explicitly using a pathfinding algorithm, the neural network need not deal with the complexity of emulating the control flow thereof. Secondly, the algorithm makes assumptions about the representation of its inputs. A neural network that complies with these assumptions can be used to generate this representation also for use in other tasks. Such a neural network can then effortlessly be reused in other programs. As an example of this, consider a program that uses a neural network to convert a natural language to subject-predicate-object-relations, for knowledge graph building. Such a neural network can easily be reused to also provide such relations for use in question answering.

Further generalization of these principles results in pipelines that freely intermix neural and programmatic modules, where neural modules are used to black-box model intricate transformations with simple control flow, while programmatic modules handle white-box transformations, incorporate expert knowledge, and provide advanced control flow. Put simply, we envision a language in which use, training, and reuse of neural components of a program is as natural as other programming tasks.

We will first inspect the current state of differentiable programming as it applies to programs generally by looking at some existing tools. We will exemplarily look at Julia's Zygote [20] and Haskell's AD library [21]. They and many similar tools have in common that code written in the host language can be made differentiable with relatively little effort. As a result, writing down a regular computation graph, e.g. a neural network, is as simple as implementing the forward pass. Additionally, we can also write more complex programs with loops, recursion, and symbolic operations and still get a gradient. However, here we are no longer guaranteed a *useful* gradient. While the gradient is mathematically correct, it does not exhibit the qualities we require to base gradient descent on it. Control flow and symbolic operations will often exhibit a gradient that behaves like a step function: Zero gradient up to the decision boundary, then a discontinuity as the output shifts abruptly. This is amplified whenever control flow builds on top of other control flow to construct more complex behavior. Thus, if the programs we write using the aforementioned tools shall be trained using gradients, we must be careful whenever we implement discontinuous program behavior.

For example, consider a program such as:

```
step x = if x > 0
  then 1
  else -1
```

In this program, the gradient of x is always zero or undefined, and any gradient passed through the program is therefore useless. That is, we can not use gradient descent to optimize x such that `step x` returns a desired result.

Discontinuities, however, are a fundamental property of Turing-complete programs, and no (modified) gradient can be constructed that gives useful results in all cases. However, this has not stopped the field from engineering solutions that address large parts of the problem. Consider that neural networks were originally designed around step functions as non-linearities [22]. Nowadays, we use ReLUs or sigmoids instead, due to their useful gradients. Likewise, we can approximate step functions in our differentiable programs with sigmoids, interpolating between the outcomes of the above example according to the sigmoid [17]. As a result, we can repair gradients for large classes of programs that include control flow such as loops and conditions.

This comes, however, with trade-offs: For one, the behavior of such approximations can become unpredictable and even useless in more complicated programs:

```
two_step x = if x < 5
  then 1
  else if x < 10
    then 2
    else 0
```

Here, regular gradients are again useless. In this example, interpolating over results does not offer great results either. Consider $x = 10$ for example. We would interpolate mostly over the second conditional, resulting in an output of $two\_step\ 10 \approx 1$. This might not be

acceptable: Suppose the desired output would be 1. The proper optimization step would be to shift x s.t. $x < 5$, but in relaxed form, gradient descent makes no changes to x, as the modified function outputs 1, which is the desired result already.

Furthermore, continuous relaxations also restrict the set of valid programs, as one must interpolate between the different outcomes of the subprograms.

```
step x = if x > 0
  then "A"
  else "B"
```

In this example, interpolation between the outcomes is no longer viable. This might not be a fundamental problem, as we can build powerful programs in spite of this particular limitation, but we take many tools away from the programmer.

SPLL takes a different approach to differentiation: We observe that while the outcome x of a program might not be smooth in its parameters θ, the probability of an outcome of a program $p(x|prog, \theta)$ is smooth in the parameters θ. Thus, wherever we can compute this probability, we can also compute a gradient that accurately reflects the underlying program for the sake of gradient-based optimization. A constraint imposed by this is that the result of a program needs to be probabilistic to some degree. This can be because a deep probabilistic model decomposes the input into a distribution of possible symbolic representations, or because the SPLL program itself invokes random numbers. The downside of this approach is that the probability of an outcome can be difficult to compute. One of the goals of SPLL is to clarify the boundary between programs for which computing this probability is easy or hard.

5.2. Comparing expressiveness of SPNs and SPLL

In this section, we establish two key theoretical results regarding the expressive power of SPLL: (1) SPLL can express any Sum-Product Network (SPN), and (2) SPLL can express structures beyond the capabilities of SPNs.

Theorem 1. *Any Sum-Product Network can be expressed in SPLL.*

Proof sketch. We can express any SPN in SPLL by encoding its sampling procedure. The key is to show that SPN nodes map directly to SPLL constructs:

- **Sum Nodes** are equivalent to if-then-else statements in SPLL with a random condition. Formally, a sum node associating node A with weight p and node B with weight $(1 - p)$ is equivalent to

  `if Bernoulli(p) then A else B`

- **Product Nodes** map to list concatenation in SPLL. Both assume independence and multiply the probabilities of their components.
- **Leaf Nodes** are represented by primitive distributions in SPLL (e.g., Normal distribution). Since leaves in SPNs include parametric normal distributions, and SPLL includes merely the standard normal distribution, we have to add linear transformations. □

A minimal grammar for SPLL programs covering SPNs, where linear transformations are folded into the probabilistic primitive, would be:

$$Expr \rightarrow if\ (bernoulli\ p)\ then\ Expr\ else\ Expr$$

$$Expr \rightarrow normal\ \mu\ \sigma$$

$$Expr \rightarrow null$$

$$Expr \rightarrow cons\ Expr\ Expr$$

We restricted this proof to only nodes with two children. SPN nodes with more than two children can always be split into multiple nodes with two children each.

Theorem 2. *SPLL can express probabilistic structures that cannot be represented by SPNs.*

Proof sketch. Consider a recursive process generating variable-length lists from a Gaussian mixture distribution:

```
main = if Uniform >= Theta[0]
  then Null
  else Cons
    Normal * Theta[1] + Theta[2]
    main
```

This SPLL program generates lists of variable and unbounded length, where longer lists are exponentially less likely. SPNs cannot model such a distribution due to the variable length of results: SPNs require a fixed set of variables, and variable-length outputs violate completeness. SPNs also have a fixed structure and cannot represent potentially infinite recursive processes. This SPLL program can however be processed by the compiler, and the resulting density function is accurate.

Thus, SPLL can express distributions beyond the capabilities of SPNs. □

These theorems establish that SPLL is strictly more expressive than SPNs, capable of representing all SPN structures and beyond. This increased expressivity allows SPLL to model more complex probabilistic processes while still maintaining the ability to leverage the tractability guarantees of SPNs when appropriate.

5.3. *Recursive inference and tractability*

The above list-generating program is not only inferred correctly, but inference is also tractable. Generally, any program that generates structure with every recursion (such as the list elements here) will have a structurally recursive inference pass. Likewise, any time the generated structure determines the taken branch, the inference engine can know the trace of the generator and thus does not need to branch. Specifically, in the example program above, the list length informs the inference engine's choice to evaluate the recursive or non-recursive branch. Thus, a query can be answered in as many recursive steps as there are list elements. On the flip side, a similar function that adds normally distributed numbers instead of accumulating them in a list is intractable. While this can syntactically be represented in SPLL, the compiler will refuse to compile this, as no tractable inference strategy is implemented for this.

Note also that we do not claim tractable inference for all recursive programs. There are potentially edge cases where NeST generates recursive inference code that does not terminate or is not tractable, though such behavior is considered a bug.

6. Experimental validation

Our intention here is to demonstrate capabilities enabled by NeST in five experiments. First, we empirically substantiate the previous theoretical arguments that showed that NEST's expressivity strictly exceeds SPNs. Second, we demonstrate that NEST can easily organize multiple neural models into a mixture of experts. In a variant of this experiment, we show that multiple SPLL programs can cooperate to train the same set of experts in different ways. In another experiment, we show that probabilistic reasoning on neural network outputs is easily possible using SPLL. Finally, we demonstrate that NeST can be used for training and inference in large language models. Typically this requires access to large compute clusters. Instead, we use NeST to combine different parts of the model that are independently trained on different subsets of the data, reducing the need for multi-node training or inference.

More concretely, we aim to answer the following questions: **(Q1)** Does NeST cover SPNs? **(Q2)** Moving beyond SPNs, can NeST additionally deal with neural networks? **(Q3)** Can we switch learning settings, e.g. from supervised to semi-supervised learning, simply by reprogramming? **(Q4)** Does NeST support reasoning on the outputs of neural networks? **(Q5)** Can NeST help in orchestrating large pre-trained models and train the resulting combined model?

We note that the pre-trained models and neural architectures used here are not chosen to maximize performance. We intend to show how complex inference pipelines can be built from simple SPLL programs. The neural components can be exchanged easily. All experiments except for the large language models experiment were run on a single consumer-grade GPU.

6.1. ***(Q1)*** *SPLL covers sum-product networks*

Because SPNs offer good guarantees of e.g. tractability, it is desirable to demonstrate that SPLL can fall back on those guarantees if the problem at hand is sufficiently simple. We can express any SPN in our language simply by writing out its sampling procedure. Given identical sampling procedures, SPLL's derived inference will also be identical. On the back of solid theory on this topic, we will only show a simple experiment here for illustrative purposes. In SPLL, we can make the independence assumption of an SPN product node by concatenating two lists of variables. Likewise, we can mix two distributions, i.e. sample from one or the other, by using an if-expression.

We can thus model a simple mixture of Gaussians using an SPN and using SPLL. We used a synthetic dataset consisting of two two-dimensional Gaussians. For this we use the SPLL code shown below. We will compare this to an equivalent SPN using SPFlow [23].

```
main = if Uniform >= Theta[0]
  then [Normal * Theta[1] + Theta[2],
        Normal * Theta[3] + Theta[4]]
  else [Normal * Theta[5] + Theta[6],
        Normal * Theta[7] + Theta[8]]
```

We find that in both cases, the parameters of the respective structures match those found in the distribution: SPN and SPLL both get the mixture ratios and the underlying parameters of the normal distribution right to within a small approximation error. Please refer also to Fig. 5, where we display the probability density for both models. The resulting figures line up to the point of being indistinguishable.

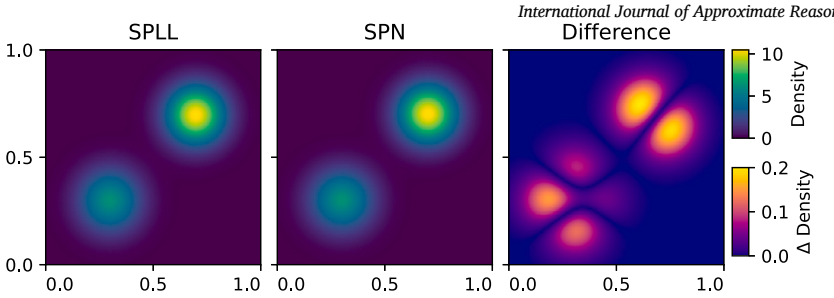

Fig. 5. Comparison of the learned density functions of synthetic data of an SPN (left) and SPLL (center). Absolute difference between the plot on the right (note the scale). We can see that the density functions learned by both are identical save for some noise, thus showing empirically that SPLLs are at least as powerful as SPNs.

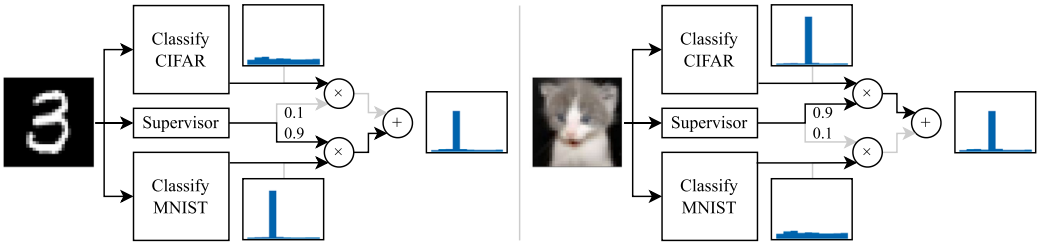

Fig. 6. Illustration of the inference pass of the expert system.

6.2. ***(Q2)*** *differentiable mixture of expert programs*

Having established that we can fall back to the guarantees of SPNs whenever appropriate, we will now tie neural networks into a model. To this end, we will show that NeST can be used to ensemble multiple neural networks into an expert model. We use MNIST and CIFAR10. Both datasets were brought to the same dimensionality by introducing color channels to MNIST and cropping the margin on CIFAR. We used the following SPLL code:

```
expert im = if isMNIST im
    then classifyMNIST im
    else classifyCIFAR im
```

Which transpiles to the following Julia model:

```
function expert_prob(sample, img)
    return (
        l_1_cond = isMNIST(img)[2];
        ((l_1_cond * classifyMNIST(img)[1 + sample])
        + ((1 - l_1_cond) * classifyCIFAR(img)[1 + sample]))
    )
end

function expert_gen(img)
    return if randomchoice(isMNIST(img)) == 2
        randomchoice(classifyMNIST(img))
    else
        randomchoice(classifyCIFAR(img))
    end
end
```

Here, isMNIST is a 2-way, and classifyMNIST and classifyCIFAR are 10-way classifiers. We use convolutional and dense networks, but the architecture and hyperparameters of these networks are largely interchangeable. A graphical illustration of the inference pass can be seen in Fig. 6.

We then pre-train classifyMNIST briefly on only MNIST, to ensure that the expert system aligns with our notion that each inner classifier specializes in the respective dataset. After that, we train the full model to completion, providing only image-classification pairs. That is, we train the entire expert system on a mixture of MNIST and CIFAR images. At the very beginning of this second training phase, we freeze the weights of classifyMNIST. In this pipeline, isMNIST is supervised in a very loose manner, and its training objective is basically to pick the better classifier.

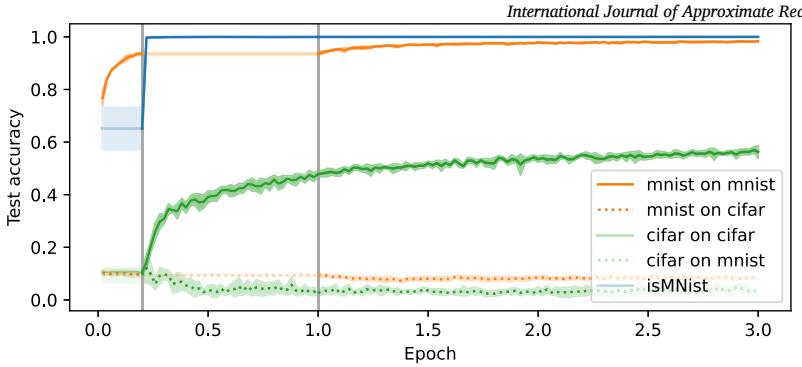

Fig. 7. Test performance during training epochs of the neural networks contained in the expert system. Displayed is mean and variance of 5 runs. For 0.2 epochs, we train only classifyMNIST on MNIST data. From this point on, samples are only labeled with their class label and not their dataset label. We freeze the weights of classifyMNIST, and train the rest of the expert system for 0.8 epochs. In the final two epochs, we train all parameters. The specializations in the expert system stabilize as desired. (Best viewed in color).

Results. The results can be seen in Fig. 7. As one can see, mixtures of neural network experts can be trained. Indeed, one can find that a little bias, e.g. via the training schedule, ensures that the mixture of experts converges towards its intended role. Furthermore, the results demonstrate that we can exert effective and fine-grained control over which components actively learn and what they learn. Meanwhile, SPLL provides useful gradients throughout the pipeline.

The key benefit of the above program is that it arranges different models with different structures and trained on different data sets to jointly handle a more complex problem. Each component adheres to a properly defined and semantically meaningful role. As a result, the components of this pipeline can be reused in other pipelines, which we will explore further in the next experiment.

6.3. ***(Q3)*** *semi-supervised learning via dynamic dispatching*

We now modify the previous experiment's SPLL code such that it annotates each sample with an integer indicating the classifier that was used to solve the instance:

```
expert_annotate im = if isMNIST im
    then [0, classifyMNIST im]
    else [1, classifyCIFAR im]
```

If we use this program to train the contained networks, then it also expects samples to be annotated with the dataset they belong to. Since the contained neural networks can share parameters between both programs, we can freely choose which program –annotating or not– to train or evaluate on. We can thus train on unannotated or sparsely annotated data and evaluate to annotate data. That is: by dynamically dispatching samples that do not have the dataset annotation to the `expert` inference function and dispatching those with that annotation to the `expert_annotate` inference function, we can train from partially labeled data. Samples with dataset annotations will provide gradients for `isMNIST` as well as the respective expert, as if directly supervised. In experiments, we can show that even a small fraction of thusly annotated data is sufficient to make the neural networks converge to their intended goals. This is an alternative to the previously described pre-training and weight freezing. In the absence of such a mechanism, the neural networks will not align with their respective goals.

Results. can be seen in Fig. 8, where subnetworks might be trained on different data, or be trained on an unintended objective. In about half of all runs with 0% annotated data, the assignment of datasets to experts is generally correct. Shown here is a case where it is not.

6.4. ***(Q4)*** *reasoning on network outputs*

We will demonstrate reasoning on the task of MNIST-addition. That is, the task is to learn to classify MNIST from samples such as (5, 7, 12), where each pair of training images is labeled with its sum. The challenge lies in reasoning through the addition operation and reconstructing probable classifications of the images from there. The SPLL code used for this experiment is simply

```
classifyMNist :: Symbol -> Int ([0..9])
mNISTAdd im1 im2 = classifyMNIST im1 + classifyMNIST im2
```

where Symbol denotes a type which SPLL needs not handle, merely pass on. This transpiles to the following PyTorch module:

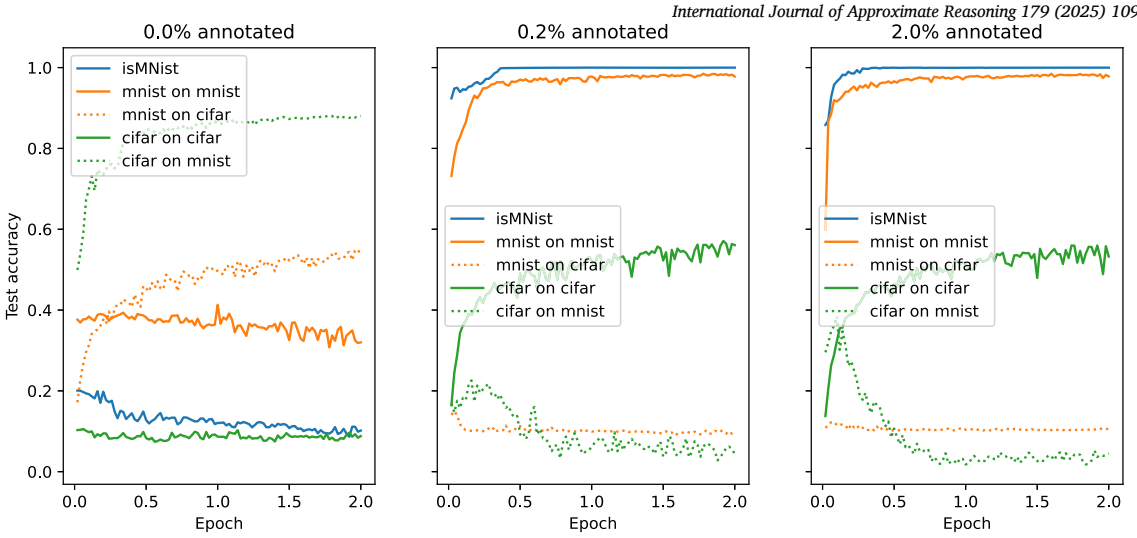

Fig. 8. Test performance during training epochs of the neural networks contained in the expert system. A fraction (0%, 0.2%, 2%) of the data has dataset annotations and is used to train `expert_annotate`, the rest is used to train `expert`. While 0% annotations can lead to a misaligned network, already a very small number of annotated samples provides strong guidance and aligns each neural network with its prescribed task. (Best viewed in color).

```
class MNISTAdd(Module):
  def generate(self, im1, im2):
    return (
      randomchoice(classifyMNIST(im1))
      + randomchoice(classifyMNIST(im2)))

  def forward(self, sample, im1, im2):
    return sum(
      map(
        (lambda l_1_enum: (
          (
            classifyMNIST(im1)[l_1_enum] *
            classifyMNIST(im2)[(sample - l_1_enum)]
          ) if contains(
            (sample - l_1_enum),
            [0,1,2,3,4,5,6,7,8,9]) else 0.0)),
        [0,1,2,3,4,5,6,7,8,9]))
```

This transpilation employs the previously discussed intractable extension (see Section 3.5) and neural network bindings. For the sake of generating valid model counting code, the neural network here must be forward-declared with the range of values it outputs. For a reference of the used functions here, see also Appendix B. As with previous experiments, the network architecture and hyperparameters are not the focus of this experiment. We train this program on a sample (im1 =  , im2 =  , 12) by maximizing $p(12|\texttt{mNISTAdd im1 im2})$, adjusting the parameters of the neural network accordingly.

Please note that SPLL makes it easy to change what operation we do on the digit classifications. For example, we could easily exchange addition for multiplication. SPLL can also reason through noisy labels. For example, we can contaminate part of the data by adding 1 to the label. To reason about this noise, we implement an SPLL program that generates the same kind of noise:

```
mNISTAddNoise im1 im2 = if theta[0] >= uniform
    then classifyMNIST im1 + classifyMNIST im2
    else 1 + (classifyMNIST im1 + classifyMNIST im2)
```

Here, training samples could for example be ( ,  , 12) or (, , 13). SPLL can still learn in this noisy environment, though of course slightly slower than without noise. The resulting training curves can be seen in Fig. 9. Naturally, we can also incorporate more complex reasoning. Trivial extensions of the above are multiplication, addition of multiple numbers, or addition of numbers with multiple digits.

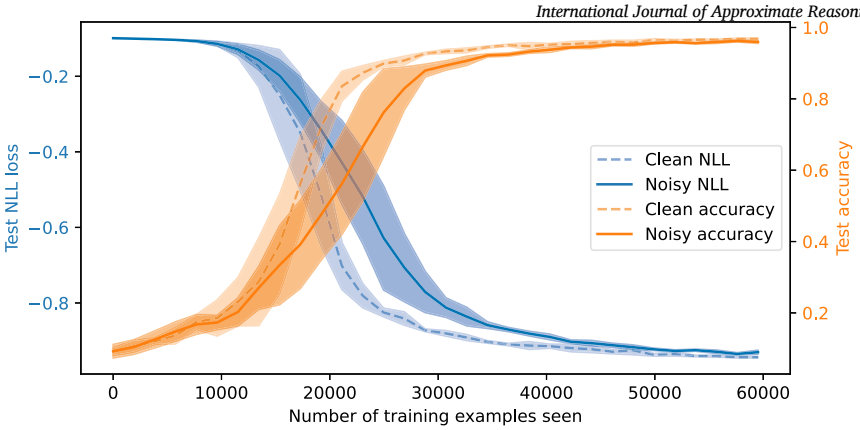

Fig. 9. Test negative log-likelihood and accuracy of neural networks trained via SPLL and MNIST-Addition. Aggregated across 5 runs. To ease comparisons, test performance is measured on 10-way classification.

Table 1
Negative log likelihoods (lower is better) of the domain-specific expert models and the combined expert system on different datasets. Expert 1 was trained on eli5, expert 2 on stack. No model saw dialogsum during training. The supervisor was trained on an unlabeled mix of eli5 and stack. Note how the combined model consistently scores very close to the better expert, meaning the supervisor consistently identifies the correct model for the task, even on the third dataset.

	Eli5	Stack	Dialogsum
expert1	4.139	4.419	4.330
expert2	5.439	2.240	4.806
NeST	4.150	2.286	4.368

6.5. ***(Q5)*** *modular programming of large language models*

There is growing interest in gaining better insight and control over the competencies of large language models. For example, it is interesting to restrict answers to certain queries to only be derived from parts of the training data. This could be done to increase modularity and thus scalability [24] or in order to maintain information security. We show that SPLL can effectively be used with such larger models and provide this modularity. The goal is to verify that larger models can also be ensembled this way and that supervision works as intended. For this experiment, we build an expert system based on DistilGPT2 [25]. The model consists of two instances of DistilGPT2 that were fine-tuned on different datasets and one instance of a DistilGPT2-based binary classifier as a supervisor. The first expert was trained on the /r/askscience section of ELI5 [26], a dataset that contains natural-language questions and answers on scientific topics. The second expert was trained on a small subset of The Stack [27], a dataset that contains program code. Thus, the two experts are assumed to be pre-trained in a supervised fashion on their respective domains. In principle, they can be replaced by any other embedding-compatible models.

We train the supervisor by training the entire expert system on data from the two data sets, updating only the supervisor parameters. Thus, supervisor weights are shifted such that the correct expert is selected. This process is fast and was done on a single Nvidia V100 GPU. We evaluate the combined, as well as the two contained generative models, by computing their average negative log-likelihood on test sets of each of the training datasets, as well as a third dataset, DialogSum [28]. DialogSum contains natural language dialogue interactions on varied topics. It is thus expected that this task is more strongly represented by the expert that was fine-tuned on natural language. However, since this dataset is distinct from the /r/askscience dataset, the supervisor needs to generalize beyond the training domain.

The results summarized in Table 1 clearly show that the supervisor was consistently capable of picking the right expert. Even on the third, unseen domain, the supervisor was capable of picking the right model, as the log-likelihood of the combined model almost matches that of the better expert. Concrete examples of prompts and the answers generated by all three models can be seen in Fig. 10. We show more generated samples from the expert model in the appendix in Table 2.

To summarize, our empirical evidence shows that all questions **(Q1)**-**(Q5)** can be answered affirmatively: NeST is a useful transpiler for formulating (conditional) distributions with recursive, looping, and differentiable properties.

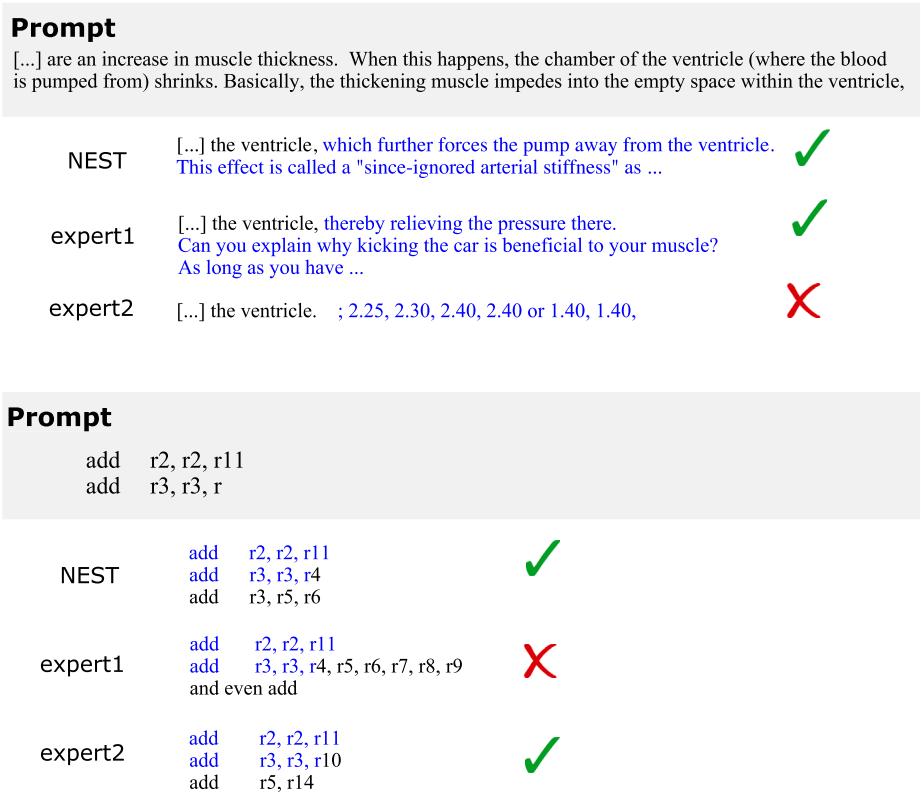

Fig. 10. NeST merges multiple experts to deal with multiple domain inputs. Comparisons of text generation on the ELI5 and The Stack datasets. Both experts fail to generate texts in the domain they are not trained in.

7. Discussion and conclusion

In this final section, we will discuss design decisions and limitations, draw conclusions and present possible avenues for future work.

7.1. Discussion

Design Decisions. Building NeST and SPLL required some design decisions that deserve discussion. One is the use of an intermediate representation (IR) within NeST. Using an IR allows the compiler developer to put a layer of abstraction between the derivation of inference strategies and their implementation within the target language. As a result, adding more inference strategies is made independent of the number of supported target languages. However, it comes with downsides, making careful consideration of the resulting tradeoffs necessary. One aspect we found crucial is the level of abstraction used within the IR. For example, during development we added several high-complexity, abstract operations to the IR that did specific tasks very well. They also generate compact and relatively readable target language code. However, these constructs are too specialized to be reused. A more productive approach here was to add smaller operations that combine readily into more complex units of meaning. Simpler operations are also easier to transpile to the target language. A disadvantage here is that some complex inference strategies need to be translated into many IR operations, which tends to hurt readability of the resulting code. From our experience implementing NeST, using a simpler IR appears to be the preferable choice.

Limitations. We would also like to give a broad overview of the expressive limitations of SPLL. When introducing the syntax and semantics of SPLL, we incorporated several restrictions to ensure sound and fast inference. The first one is that SPLL does not allow for use of variables. In conventional programming, this might not always be a meaningful restriction, as variables can be inlined at the cost of redundant computations. In probabilistic programming, this is more restrictive, as the result of a random process can then not be shared in two different parts of the computation. Inlining in probabilistic programming is semantically meaningful and thus not an alternative. Furthermore, we restricted all functions to be non-parametric or only use deterministic parameters. This makes inference easy, but is restrictive. We are confident that a more permissive restriction here would be sufficient to ensure fast inference, though deterministic parameters map neatly onto conditional probabilities. It is also likely that parametric functions adversely impact exact inference through the same mechanism as variables, as said parameters create information flow much like variables. As a last noteworthy restriction on expressivity, SPLL currently does not allow the use of tuples and sum types. While lists can be used to

emulate some of their respective properties in SPLL, tuples and sum types would both elevate the theoretical expressiveness as well as practical usability. Determining which of these constraints are fundamentally necessary to exact tractable inference and which are resolvable through continued development remains an open question.

Densities or Probabilities? Previously, we switched between densities and probabilities ad hoc. However, it is generally prudent to properly distinguish between the two. Continuous random variables are modeled with PDFs(Probability Density Functions) where $p(X = x)$ is a probability density and discrete random variables are modeled with PMFs (Probability Mass Functions) where $P(X = x)$ is a probability mass. We noted that our preferred implementation is to annotate each probability measure with its dimensionality, such that masses are considered dimension 0, while densities have positive dimensionality. Intuitively, this appears to be equivalent to matching densities with continuous return types and probabilities with discrete return types. However, we can build some examples that behave in interesting ways and make dynamic counting necessary. For example, we can use a discrete and continuous type each in a tuple (product type) or a tagged union (sum type)—represented here with a list. Consider the following programs, which construct a sum and product type.

```
list = if theta[0] > Uniform
  then Null
  else Cons Normal Null
```

```
tuple =
  (if theta[0] > Uniform
    then True
    else False,
  Normal)
```

In both cases, the resultant type contains both discrete and continuous types. For the `tuple` example, it makes sense to treat this as a continuous variable, as for any value (a, b) we can find values (a, b') that are arbitrarily close. Therefore, a product type containing a continuous variable is again continuous. To be precise, the dimensions of each contained type are added. In the case of a discrete-continuous tuple, this is known as a mixed distribution.

The same process does not work for the `list` example. Here, the value `Null` does not have a continuous neighborhood. Instead, this value is completely discrete. We should therefore treat it as such, assigning probabilities to `Null` while assigning densities to `[x]`. Consequently, whether densities or probabilities are the correct language for expressing the probability of a return value is a run time property that depends not on the type of the expression, but on each value individually. Similar constructs are known as mixed random variables, though they are underexplored in tractable models. The dimensionality thus being a run time property means that statically determining the kind of measure is unviable, hence we count dimensions at run time.

With this distinction in mind, we can adjust the equations for normalized distributions as such:

$$\int_x p(\mathtt{e} \Downarrow x) dx + \sum_y p(\mathtt{e} \Downarrow y) = 1$$

where x and y are continuous and discrete outcomes, respectively. We consider it future work to provide an actual implementation of a general system that can deal with mixed random variables and keep densities and probabilities properly separate.

7.2. Conclusions

We introduced Sum-Product Loop Language (SPLL), a probabilistic programming language that combines useful features of Sum-Product Networks, such as tractability and exactness, but can capture more complex distributions. By automatically generating inference code, the language provides inference to the user for low mental overhead. We also introduced the Neuro-Symbolic Transpiler (NeST), a tool to translate SPLL into high-performance differentiable computing ecosystems such as PyTorch or Julia. This way, NeST can integrate probabilistic programs with complex control flow into environments that support neural networks. We show easy integration of neural networks via experiments: Our probabilistic programs can supervise and train neural networks and even coordinate mixture-of-experts of large pre-trained models.

7.3. Future work

Our work provides several interesting avenues for future work on SPLL and NeST.

Parameter Passing. While it is currently already possible to define large models in SPLL, handling the parameters for all these is still a largely manual task of keeping track of indexes into the θ-vector. However, this could be drastically simplified by providing a more comprehensive data structure that can more adequately reflect which parameters are relevant where in the program. For example, if a tree structure was chosen, within each function call only the appropriate part of the parameter space would be passed on to the called function. For example, we could define a sum node in an SPN like so:

```
sum_node theta layer = if Uniform > (root theta)
  then product_node (subforest theta !! 0) (layer - 1)
  else product_node (subforest theta !! 1) (layer - 1)
-- "list !! index" is notation for list indexing.
```

This notation has the advantage of eliminating code duplication. Both product nodes can use the same code because their differences are defined via the passed subforest of the theta structure. However, users need to reason about which weights to pass. As a simpler, but less general supplement to such an approach, it could also be useful to have completely local, anonymous parameters. These can easily be used ad-hoc to represent values that are not shared in any other part of the program. Likewise, for readability, it could be useful to have named parameters that represent clearly identified concepts, e.g. the coefficient of friction in a mechanical device.

Invertible Transfomations and Memory. Currently, SPLL does not have good support for local variables. This is because local variables substantially complicate probabilistic inference. The general case of a local variable looks like this:

```
stateful = let x = e1 in e2
```

In this general case, the completely general solution for inference is

$$p(y|\texttt{stateful}) = \int p(x|\texttt{e1})p(y|\texttt{e2}, x)dx\,,$$

however this integral is rarely easily solved. While we can not solve all instances of the class of programs described by `stateful`, we can nonetheless solve many useful instances. For example, if x is of a small set of discrete values, we can easily enumerate and solve similar to the strategy previously described in subsection 3.5. More interestingly though: If we know from the mathematical structure of `e2` that x can be computed from y, i.e. $\exists f : f(y) = x$, then the integral above collapses into a very straightforward computation: $p(y|\texttt{stateful}) = p(x|\texttt{e1})p(y|\texttt{e2}, x)$. Under this constraint, there is a connection to normalizing flow models, which make use of invertible functions to build probabilistic models. To achieve this, it would be necessary to build an understanding of the invertibility of expressions into the language.

Queries. Currently, SPLL only has explicit support for simple probabilistic inference. However, we know that SPNs support fast evaluation of more complicated queries. The implementations of such queries can be applied to SPLL too, thus enabling such queries for SPLL. Ideally, SPLL would support more advanced queries by adding tools to formulate such queries within the language, thus adding more tools to the language that can also be used in other situations. For example, it is already possible to query the cumulative distribution function of a distribution by using the `>=` operator. Adding more such tools to build up to the set of queries supported by SPNs should improve the expressiveness of the language while remaining closely informed by prior work on SPN inference. Additionally, such work on queries would make it possible to train from partial data by maximizing the probability of answering a marginal or conditional query correctly. For example, building on top of the ideas in Section 6.3, we could maximize $p([\_, 3]|\texttt{expert\_annotate im})$ to maximize the probability of classifying the image as belonging to class 3, without deciding which expert should make that decision.

Parallelization. Since both the probabilistic semantics and the generative semantics can imply the evaluation of if-then-else expressions, running either code in parallel comes with extra challenges. The target language implementations shown in this paper are completely sequential, but an order of magnitude of performance or more is available via parallelization. The focus should be on probabilistic semantics here, as this is necessary for fitting parameters.

CRediT authorship contribution statement

Viktor Pfanschilling: Writing – original draft, Software, Investigation, Conceptualization. **Hikaru Shindo:** Writing – review & editing, Visualization, Software. **Devendra Singh Dhami:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Kristian Kersting:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We gratefully acknowledge support by the German Research Center for AI (DFKI) and the Hessian Ministry of Higher Education, Research and the Arts (HMWK). The work was also supported by the Federal Ministry for Economic Affairs and Climate Action (BMWK) AI lighthouse project “SPAICER" (01MK20015E), the EU ICT-48 Network of AI Research Excellence Center “TAILOR" (EU Horizon 2020, GA No 952215) and the Collaboration Lab “AI in Construction” (AICO) with Nexplore/HochTief. The work has also benefited from EU Horizon project TANGO (101120763), the Federal Ministry of Education and Research (BMBF) Competence Center for AI and Labour (“KompAKI”, FKZ 02L19C150), and the HMWK cluster projects “The Third Wave of AI” and “The Adaptive Mind”. The Eindhoven University of Technology authors received support from their Department of Mathematics and Computer Science and the Eindhoven Artificial Intelligence Systems Institute. The funding sources had no involvement in the conduct of the research or preparation of this article.

Appendix A. Compiled SPLL programs

Below is the Julia compilation result of the example program shown in 4.1, preceded by another copy of the relevant SPLL source code.

```
bernoulli = Uniform >= Theta[0]
```

```
function bernoulli_prob(thetas, sample)
  return (
      l_4_lhs_integral = (
        l_3_high = if (1.0 >= thetas[0])
          thetas[0]
        else
          1.0
        end
        if (0.0 >= l_3_high)
          0.0
        else
          (l_3_high - 0.0)
        end
      )
      if (true == sample)
        (1.0 - l_4_lhs_integral)
      else
        l_4_lhs_integral
      end
    )
end
function bernoulli_gen(thetas)
  return (rand() >= thetas[0])
end
```

Appendix B. NeST python library

Below is the implementation of common functions used by NeST in the implementation of SPLL inference:

```
import torch
import numpy as np

def rand():
    return np.random.rand()

def randn():
    return np.random.randn()

def hcat(head, tail):
    return [head] + tail

def density_Normal(x):
    return (1 / torch.sqrt(torch.tensor(2 * 3.14159)))
      * torch.exp(-0.5 * x * x)

def contains(elem, list):
    return elem in list

def randomchoice(vector):
    return np.random.choice(np.arange(len(vector)), p=vector / sum(vector))
```

Appendix C. Further LLM expert system samples

Table 2
Text samples generated by the combined expert model. The supervisor reliably picks the correct model to sample from.

Prompt - Generated text
[..] The main problems with using renewables like solar and wind is that they can't be 'dialed' up all the time. We need to have a fixed, lower load and this is absolutely correct and efficient. [..]
[..] U.S. energy consumption can largely be broken down into two main types: stationary and portable. The electrical power grid would be the main use of stationary power, while cars, trucks and the like would be the main use. Since cars are relatively limited in size they can pose a lot of problems with standards capacity, [..]
[..] The beam dumps are long, water-cooled rods of carbon encased in steel, designed to absorb the light of they are being pulled out of the suspended silicon beam hexagonal shape. [..]

`ON_CHANGE ; change only if needed`
`call SBDWriteFMReg`
`add al, DISP_FROM_LEVEL_TO_ATTACK ; al <- attack address of mod`
`mov al, DR1`
`add al, TRGE_STICK_INIT_TIMEOUT ;`

`nop`
`dec %rcx`
`lea addresses_normal_ht+0x18f3e, %r12`
`add %r9, %r7`
`mov (%rbx), %r11`
`map`

`ymm4, %xmm4`
`vpextrq $1, %xmm4, %rdi`
`nop`
`nop`
`cmp %rcx, %rcx`
`lea addresses_D_ht+0x4830e, %r9`
`nop`
`movups (%r11, %rsi)`
`nop`
`lea addresses_`

Data availability

The software used for this article is available under an Open-Source license.

References

[1] H. Poon, P. Domingos, Sum-Product Networks: A New Deep Architecture, UAI, 2011.
[2] V. Pfanschilling, H. Shindo, D.S. Dhami, K. Kersting, Sum-product loop programming: from probabilistic circuits to loop programming, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, vol. 19, 2022, pp. 453–462.
[3] D. Kisa, G. Van den Broeck, A. Choi, A. Darwiche, Probabilistic sentential decision diagrams, in: KR, 2014.
[4] Y. Choi, A. Vergari, G. Van den Broeck, Probabilistic circuits: a unifying framework for tractable probabilistic models, uCLA, http://starai.cs.ucla.edu/papers/ProbCirc20.pdf, Oct 2020.
[5] F.A. Saad, M.C. Rinard, V.K. Mansinghka, SPPL: probabilistic programming with fast exact symbolic inference, in: ICPLI, 2021.
[6] N.D. Goodman, V.K. Mansinghka, D.M. Roy, K.A. Bonawitz, J.B. Tenenbaum, Church: a language for generative models, in: UAI, 2008.
[7] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Deepproblog: neural probabilistic logic programming, Adv. Neural Inf. Process. Syst. 31 (2018).
[8] E. Bingham, J.P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, N.D. Goodman, Pyro: deep universal probabilistic programming, J. Mach. Learn. Res. 20 (1) (2019) 973–978.
[9] D. Tran, M.D. Hoffman, R.A. Saurous, E. Brevdo, K. Murphy, D.M. Blei, Deep probabilistic programming, 2017.
[10] S. Holtzen, G. Van den Broeck, T. Millstein, Scaling exact inference for discrete probabilistic programs, Proc. ACM Program. Lang. (2020).
[11] M. Torres-Ruiz, R. Piedeleu, A. Silva, F. Zanasi, On iteration in discrete probabilistic programming, in: 9th International Conference on Formal Structures for Computation and Deduction, 2024, p. 1.
[12] Z. Yang, A. Ishay, J. Lee, Neurasp: embracing neural networks into answer set programming, in: IJCAI, 2020.
[13] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, J. Artif. Intell. Res. (2018).
[14] T. Rocktäschel, S. Riedel, End-to-end differentiable proving, NeurIPS, 2017.
[15] J. Mao, C. Gan, P. Kohli, J.B. Tenenbaum, J. Wu, The neuro-symbolic concept learner: interpreting scenes, words, and sentences from natural supervision, in: ICLR, 2019.
[16] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, F. Bach, Learning with differentiable pertubed optimizers, in: NeurIPS, 2020.
[17] F. Petersen, C. Borgelt, H. Kuehne, O. Deussen, Learning with algorithmic supervision via continuous relaxations, Adv. Neural Inf. Process. Syst. 34 (2021) 16520–16531.
[18] M. Chavira, A. Darwiche, On probabilistic inference by weighted model counting, Artif. Intell. 172 (6–7) (2008) 772–799.
[19] L. Damas, R. Milner, Principal type-schemes for functional programs, in: R. DeMillo (Ed.), Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '82, ACM Press, New York, New York, USA, 1982, pp. 207–212.

[20] M. Innes, Don't unroll adjoint: differentiating ssa-form programs, arXiv:1810.07951, 2019.
[21] E. Kmett, Automatic differentiation, https://github.com/ekmett/ad. (Accessed 12 January 2024).
[22] F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain, Psychol. Rev. 65 (6) (1958) 386.
[23] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N.D. Mauro, P. Poupart, K. Kersting, Spflow: an easy and extensible library for deep probabilistic learning using sum-product networks, arXiv:1901.03704, 2019.
[24] S. Gururangan, M. Lewis, A. Holtzman, N.A. Smith, L. Zettlemoyer, Demix layers: disentangling domains for modular language modeling, arXiv preprint, arXiv: 2108.05036, 2021.
[25] V. Sanh, L. Debut, J. Chaumond, T. Wolf, Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, arXiv preprint, arXiv:1910.01108, 2019.
[26] A. Fan, Y. Jernite, E. Perez, D. Grangier, J. Weston, M. Auli, ELI5: long form question answering, in: A. Korhonen, D.R. Traum, L. Màrquez (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, in: Long Papers, vol. 1, Association for Computational Linguistics, 2019, pp. 3558–3567.
[27] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, H. de Vries, The stack: 3 tb of permissively licensed source code, Preprint, 2022.
[28] Y. Chen, Y. Liu, L. Chen, Y. Zhang, DialogSum: a real-life scenario dialogue summarization dataset, in: C. Zong, F. Xia, W. Li, R. Navigli (Eds.), Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, Association for Computational Linguistics, 2021, pp. 5062–5074, Online.