Exact Recursive Probabilistic Programming*

DAVID CHIANG and COLIN MCDONALD, University of Notre Dame, USA
CHUNG-CHIEH SHAN, Indiana University, USA

Recursive calls over recursive data are useful for generating probability distributions, and probabilistic programming allows computations over these distributions to be expressed in a modular and intuitive way. Exact inference is also useful, but unfortunately, existing probabilistic programming languages do not perform exact inference on recursive calls over recursive data, forcing programmers to code many applications manually. We introduce a probabilistic language in which a wide variety of recursion can be expressed naturally, and inference carried out exactly. For instance, probabilistic pushdown automata and their generalizations are easy to express, and polynomial-time parsing algorithms for them are derived automatically. We eliminate recursive data types using program transformations related to defunctionalization and refunctionalization. These transformations are assured correct by a linear type system, and a successful choice of transformations, if there is one, is guaranteed to be found by a greedy algorithm.

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; *Recursion*; • **Mathematics of computing** → *Probability and statistics.*

Additional Key Words and Phrases: probabilistic programming, recursive types, linear types

ACM Reference Format:
David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 98 (April 2023), 31 pages. https://doi.org/10.1145/3586050

1 INTRODUCTION

Probability models in natural language processing (NLP), as well as computational biology and other fields, often involve recursive computations on recursive data structures. Sequences of words (or other basic units) have been modeled, for example, using higher-order Markov chains [Chen and Goodman 1999], hidden Markov models [Rabiner 1989], finite automata and transducers [Mohri 1997], and chain-structured conditional random fields (CRFs) [Lafferty et al. 2001]. Neural models that use CRFs [Huang et al. 2015] and connectionist temporal classification [Graves et al. 2006], which are just finite transducers, are at or near the state of the art in a number of NLP tasks. Trees, used for representing syntactic structures and sometimes other linguistic structures, have been modeled using probabilistic context-free grammars (PCFGs) [Booth and Thompson 1973], tree automata and transducers, or tree-structured CRFs, again performing at or near the state of the art in constituency parsing [Stern et al. 2017].

In order to support applications such as optimization of parameters by gradient-based methods, it is important that algorithms for these models can efficiently produce exact results. Typically

*This material is based upon work supported by the National Science Foundation under Award Nos. CCF-2019266 and CCF-2019291.

Authors' addresses: David Chiang, dchiang@nd.edu; Colin McDonald, cmcdona8@nd.edu, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA; Chung-chieh Shan, ccshan@indiana.edu, Department of Computer Science, Indiana University, Bloomington, IN, USA.

2475-1421/2023/4-ART98
https://doi.org/10.1145/3586050

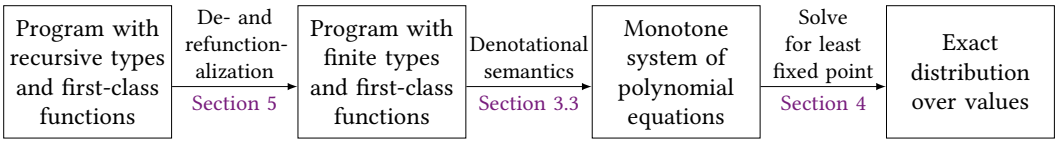

Fig. 1. How a PERPL program is compiled and evaluated. By "recursive type", we mean types such as `String` that are defined recursively and inhabited by an infinite number of values. By "finite type", we mean types such as `Bool` that are inhabited by a finite number of values.

they use dynamic programming, but more generally, they can be thought of as solving systems of linear or polynomial equations. Traditionally, these algorithms are coded by hand, which is tedious and error-prone. Worse, small changes to the model can often require nontrivial changes to the algorithm. For this reason, various toolkits have been developed, for example, for finite-state machines [Riley et al. 2009] and tree transducers [May and Knight 2006]. These provide reliable implementations of algorithms, but at the expense of flexibility: small changes to the model can often take it outside the class of model for which the toolkit was designed.

Probabilistic programming languages (PPLs) promise to avoid both manual coding and inflexible toolkits. The promise begins with an intuitive way to express a distribution, namely by describing a generative process: a run of a probabilistic program appears to generate a random outcome by making choices and scoring branches, but those probabilistic side effects are just notation for a distribution whose associated quantities—such as total weight or expected value—need to be computed. A distribution expressed in a PPL can be changed easily by editing the program, without needing to manually redesign inference algorithms.

In PPLs, recursive models such as those in NLP are naturally represented as recursive calls and data. For example, to find the probability of a string under a PCFG, it would be natural to write code that recursively generates random trees and sums the probabilities of trees yielding that string. Indeed, PCFGs served as the central example in the first PPL paper [Koller et al. 1997]. However, the example ultimately did not work out, and efficient inference on these representations remains a longstanding challenge. The general inference methods offered by PPLs today, often based on sampling, are not suitable in many applications because they take too long, don't compute gradients, or exhibit too much variance. Some PPLs do support exact inference—beginning with the "stochastic Lisp" of Koller et al. [1997], and more recently with languages like IBAL [Pfeffer 2005], Fun [Borgström et al. 2011], FSPN [Stuhlmüller and Goodman 2012], BernoulliProb [Claret et al. 2013], PSI [Gehr et al. 2016, 2020], Hakaru [Walia et al. 2019], Dice [Holtzen et al. 2020], SPPL [Saad et al. 2021], and ProbZelus [Atkinson et al. 2022]—but they impose severe limitations on recursion.

In this paper, we introduce PERPL (Probabilistic Exact Recursive Programming Language). The denotation of a PERPL program is given by a system of polynomial equations, which can be solved efficiently using numerical methods. Unlike many general-purpose PPLs, PERPL performs *exact* inference, in the sense that even if it solves the equations iteratively (by Newton's method), any desired level of accuracy is guaranteed by a known number of iterations [Stewart et al. 2015]. Moreover, it can differentiate the equations and solve for the derivatives, permitting gradient-based optimization. And unlike other PPLs that do support exact inference, PERPL can express first-class, higher-order functions, unbounded recursive calls, and some recursive data structures. For example, a PCFG parser written in PERPL appears to generate (infinitely many) trees and sum the probabilities of those (exponentially many) trees that yield a given string; yet it compiles to a cubic-sized system of equations whose solution is equivalent to the CYK algorithm. We also show programs using stacks or stacks-of-stacks for which PERPL automatically derives polynomial-time algorithms.

We demonstrate that these asymptotic speedups result in orders-of-magnitude differences in a benchmark.

The key to our approach is a linear type system [Girard 1987; Walker 2005], which enables first-class functions to be used efficiently and recursive types to be eliminated correctly using defunctionalization [Danvy and Nielsen 2001; Reynolds 1972] and refunctionalization [Danvy and Millikin 2009], even in the presence of probability effects. Figure 1 depicts our pipeline of steps that turns a naturally expressed probabilistic program into an exact distribution over return values.

Our contributions are

(1) a nonstandard denotational semantics for a PPL, in which a λ-abstraction from type τ_1 to type τ_2 denotes not a distribution over a set of functions, which has cardinality $|\tau_2|^{|\tau_1|}$, but a distribution over a set of pairs, which has cardinality $|\tau_1| \cdot |\tau_2|$ (Section 3.3);
(2) using defunctionalization and refunctionalization to eliminate recursive types (Section 5);
(3) showing that a linear type system ensures that both of the above are correct, in spite of probability effects (Theorems 3.14 and A.5);
(4) compiling programs with unbounded recursion into systems of equations and solving them; in particular, loops can be evaluated exactly and directly, not iteratively (Section 4.2);
(5) compiling natural probabilistic models into polynomial-time parsers for context-free grammars (Section 5.4), pushdown automata (Section 6.1), and their generalization to tree-adjoining grammars and embedded pushdown automata (Section 6.3);
(6) translating programs with affinely used functions to programs with only linearly used functions (Appendix B.2);
(7) an open-source implementation based on a compositional, semantics-preserving translation from PERPL to *factor graph grammars* [Chiang and Riley 2020] (Appendix C.2).

2 MOTIVATION

In this section, we present a sequence of examples with increasing demands on expressivity. First, a single coin flip (Example 2.1); second, unbounded loops and recursive calls (Examples 2.2 and 2.3); third, recursive data types (Example 2.4) and their elimination (Examples 2.5 and 2.6). The last of these will also demonstrate the need for (linearly used) first-class functions.

Example 2.1 (Probability). PERPL, like other PPLs, has a mechanism for nondeterministically sampling from probability distributions. To sample from a Bernoulli distribution, we can write:

```
define flip : Bool =
  amb (factor p in true) (factor q in false)
flip
```

where p and q are (metavariables for literal) nonnegative real weights. The distribution `flip` returns **`true`** with weight p and **`false`** with weight q. To get a Bernoulli distribution, we would require $p + q = 1$, but in general, weights do not have to sum to one. Then the final expression `flip` calls the distribution, so the value of the whole program is **`true`** with weight p and **`false`** with weight q.

Example 2.2 (Loops). To simulate a fair coin using an unfair coin, flip the unfair coin twice. If the two flips disagree, take the first flip; otherwise, repeat [von Neumann 1951]. We can express this unbounded process as:

```
define fair : Bool =
  let x = flip in let y = flip in if x = y then fair else x
fair
```

PERPL turns this program into the linear equations $t = pq + (p^2 + q^2)t$ and $f = qp + (p^2 + q^2)f$, then solves for t and f. If $p + q = 1$, then the answer is $t = f = \frac{1}{2}$.

Example 2.3 (Recursive calls). The PCFG [Booth and Thompson 1973]

$$S \xrightarrow{p} SS \qquad\qquad S \xrightarrow{q} a \tag{1}$$

defines a distribution over trees, under which the total probability of all *finite* trees is the least nonnegative solution of

$$z = pz^2 + q. \tag{2}$$

If $p + q = 1$, then $z = \min\left(1, \frac{1-p}{p}\right)$. This probability is computed by the following PERPL program.

```
define gen : Unit =
  if flip then let () = gen in let () = gen in ()     -- S → SS
          else ()                                      -- S → a
gen
```

Although this program always returns `()`, the weight associated with `()` is the desired probability z.

Notably, this program makes recursive calls of unbounded depth. The denotational semantics of this program will turn out to be the equation (2), which can be solved using standard methods. (In this case, the quadratic formula gives a closed-form solution; in general, we might need to resort to iterative methods like Newton's method, which is guaranteed to converge to the correct answer.)

Example 2.4 (Recursive data). In CFG parsing, we want to know the total weight of all derivations of a CFG that yield a given string. The following program defines a type `String` for strings over the alphabet $\{a\}$, a function `gen` that generates strings from the CFG (1), and a function `equal` that tests whether the generated string is the given one. The program returns a distribution over Booleans: the weight of **`true`** is the total weight of all CFG derivations that yield the string *aaa*, and the weight of **`false`** is the total weight of all other (finite) CFG derivations.

```
data Nonterminal = S
data Terminal = A
data String = Nil | Cons Terminal String

define gen (lhs: Nonterminal) (acc: String) : String =
  case lhs of S → if flip then gen S (gen S acc) else Cons A acc

define equal (xs: String) (ys: String) : Bool =
  case xs of
    Nil → case ys of Nil → true | Cons y _ → false
    Cons x xs → case ys of Nil → false | Cons y ys → x = y and equal xs ys

equal (gen S Nil) (Cons A (Cons A (Cons A Nil)))
```

Processing pipelines like this are common, and writing them in a PPL enables intuitive expression and modular reuse. In particular, Bayesian inference is easy to express. For instance, to predict the next word conditioned on a given prefix, we can leave the `gen` function as is, change `equal` to

```
define next_word (xs: String) (ys: String) : Terminal =
  case xs of
    Nil → fail
    Cons x xs → case ys of Nil → x
                           Cons y ys → if x = y then next_word xs ys else fail
```

where **fail** denotes the zero distribution, and change the last line of code to

```
next_word (gen S Nil) (Cons A (Cons A (Cons A Nil)))
```

Because the String type is recursive and infinite, it is not trivial for PERPL to convert a program like Example 2.4 to a finite system of equations. The rest of this section shows how PERPL manages to compile this program to efficient exact inference. One way to proceed is to apply a whole-program transformation that replaces the recursive type by a nonrecursive data structure that represents all the places in the code where it is constructed. This is closely related to *defunctionalization* [Danvy and Nielsen 2001; Reynolds 1972].

Example 2.5 (Defunctionalization). In Example 2.4, there were two uses of String: for the strings generated by gen, and for the input string *aaa* to be parsed. These can be automatically distinguished. Below, we rename them to GenString and InputString, respectively. Defunctionalization changes the latter into a nonrecursive type.

```
data GenString = GenNil | GenCons Terminal GenString
data InputString = InputNil | InputCons Terminal Position
data Position = 0 | 1 | 2 | 3

define gen (lhs: Nonterminal) (acc: GenString) : GenString =
  case lhs of S → if flip then gen S (gen S acc) else GenCons A acc

define shift (ys: Position) : InputString =
  case ys of 0 → InputCons A 1 | 1 → InputCons A 2
             2 → InputCons A 3 | 3 → InputNil

define equal (xs: GenString) (ys: Position) : Bool =
  case xs of
    GenNil → case shift ys of InputNil → true | InputCons y _ → false
    GenCons x xs → case shift ys of InputNil → false
                                    InputCons y ys → x = y and equal xs ys

equal (gen S GenNil) 0
```

We introduced a new type Position with four values corresponding to the four places in the original program where an InputString was constructed, all in the last line. A Position can be thought of as a potential InputString, or a tail of the input string. It is converted into an actual InputString by the function shift.

Due to this transformation, neither Position nor InputString is recursive, although GenString still is. If we read these programs as call-by-value random generators, then the transformation has interleaved the producer and consumer of the input string and deforested it away.

Our example still uses the recursive type GenString. We can get rid of it using *refunctionalization* [Danvy and Millikin 2009; Danvy and Nielsen 2001], a whole-program transformation that replaces the recursive type by the computation that consumes it.

Example 2.6 (Refunctionalization). Example 2.5 only consumes a GenString in one place: the **case** xs expression in equal. Refunctionalization introduces functions gen_nil and gen_cons whose bodies are the branches of this **case** expression, and replaces the constructors of GenString with these functions.

```
define gen_nil : Position ⊸ Bool =
  λys: Position. case shift ys of InputNil → true | InputCons y _ → false
```

```
define gen_cons (x: Terminal) (xs: Position ⊸ Bool) : Position ⊸ Bool =
  λys: Position. case shift ys of InputNil → false
                                  InputCons y ys → x = y and xs ys
define gen (lhs: Nonterminal) (acc: Position ⊸ Bool) : Position ⊸ Bool =
  case lhs of S → if flip then gen S (gen S acc) else gen_cons A acc
(gen S gen_nil) 0
```

All recursive types are gone now. However, refunctionalization has changed values formerly of type `GenString` to type `Position ⊸ Bool`, necessitating the use of first-class functions.

The denotational semantics of this program will be a system of equations, and solving these equations is equivalent to the CYK algorithm for CFG parsing. The parser can be generalized to an arbitrary CFG, whether in Chomsky normal form, by extending `Nonterminal`, `Terminal`, and `gen`.

Not all programs of interest are structured as producer–consumer pipelines. In Section 6, we will give examples of programs that use stacks, whose production (by pushes) and consumption (by pops) follow no fixed order.

3 A PROBABILISTIC PROGRAMMING LANGUAGE

We now define PERPL more formally. It has two main distinctive features:

- probability effects, which allow a program to nondeterministically pursue multiple branches, each possibly with a different weight or probability, and
- a linear type system, in which values of certain types must be consumed exactly once.

3.1 Syntax and Typing

The syntax rules are shown in Figure 2.

3.1.1 Probability. Evaluating an expression may involve making a probabilistic choice. The expression **amb** e_1 e_2 chooses between e_1 and e_2, so, for example, (**amb true false**, **amb true false**) splits the current branch of computation into four branches, in which the pair evaluates to (**true**, **true**), (**true**, **false**), (**false**, **true**), and (**false**, **false**), each with weight 1. The expression **factor** w **in** e, where w is a nonnegative number, multiplies the weight of the current branch by w and evaluates e. The expression **fail** has weight 0, so it terminates the current branch.

3.1.2 Linearity. We use a linear type system for two reasons, both illustrated in Section 2. First, defunctionalization, by changing some computations into data structures, delays them. For example, the calls to `Cons` in Example 2.4 are delayed to inside `shift` in Example 2.5. The fact that `ys` is consumed only once ensures that any probabilistic effects in the input string would not be duplicated by the reordering.

Second, refunctionalization wraps computations inside λ-abstractions (as seen in Example 2.6), and it would be prohibitively expensive to let a λ-abstraction from type τ_1 to type τ_2 denote a distribution over $|\tau_2|^{|\tau_1|}$ functions. But if it is used only once (as `xs` and `acc` are in Example 2.6), it can denote a distribution over input–output pairs. Since there are only $|\tau_1| \cdot |\tau_2|$ such pairs, this is far more manageable.

For both of these reasons (detailed below in Sections 5.3 and 3.3, respectively), we decree that functions must be used *linearly* [Girard 1987; Walker 2005], in that, once introduced, they must be used exactly once. For example, the following program is not well-typed, because it calls `f` twice.

```
let f = λx. amb x (not x) in (f true, f true)
```

$$
\begin{array}{lll}
\text{Programs} & p ::= e \mid \textbf{define}\ x = e\ ;\ p & \\
\text{Expressions} & e ::= x \mid \lambda x : \tau.\, e \mid e\ e \mid \textbf{amb}\ e\ e \mid \textbf{fail} \mid \textbf{factor}\ w\ \textbf{in}\ e \mid (e, \ldots, e) \mid \langle e, \ldots, e \rangle & \\
& \quad \mid \textbf{let}\ (x, \ldots, x) = e\ \textbf{in}\ e \mid e.i \mid c\ e \mid \textbf{case}\ e\ \textbf{of}\ c\ x \to e \mid \cdots \mid c\ x \to e & \\
\text{Types} & \tau ::= \tau \multimap \tau \mid \textbf{Unit} \mid \tau \otimes \cdots \otimes \tau \mid \tau\ \&\ \cdots\ \&\ \tau \mid c\ \tau \oplus \cdots \oplus c\ \tau & \\
\text{Constructors} & c \in C \qquad C \text{ is an infinite set of constructors} & \\
\text{Weights} & w \in \mathbb{Q}_{\geq 0} &
\end{array}
$$

Syntactic sugar

$$
\begin{array}{c}
\textbf{let}\ x_1 = e_1\ \textbf{in}\ e' \equiv (\lambda x_1.e')\ e_1 \\
\textbf{Bool} \equiv \texttt{True}\ \textbf{Unit} \oplus \texttt{False}\ \textbf{Unit} \qquad \textbf{true} \equiv \texttt{True}\ () \qquad \textbf{false} \equiv \texttt{False}\ () \\
\textbf{if}\ e\ \textbf{then}\ e_1'\ \textbf{else}\ e_2' \equiv \textbf{case}\ e\ \textbf{of}\ \texttt{True}\ u \to \textbf{let}\ () = u\ \textbf{in}\ e_1' \mid \texttt{False}\ u \to \textbf{let}\ () = u\ \textbf{in}\ e_2' \\
e_1\ \textbf{and}\ e_2 \equiv \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else false} \qquad \textbf{not}\ e \equiv \textbf{if}\ e\ \textbf{then false else true}
\end{array}
$$

Fig. 2. Syntax. We use a taller vertical bar (|) to separate different right-hand sides of a BNF production, and a shorter vertical bar (|) to separate the branches of a **case** expression.

3.1.3 Algebraic Data Types. Because of linearity, there are two kinds of tuples [Abramsky 1993]. In a *multiplicative* tuple $(e_1, \ldots, e_n)$, of type $\tau_1 \otimes \cdots \otimes \tau_n$, all components are computed regardless of demand, and they are all consumed together. In the common case where $n = 0$, we write $()$ for the empty tuple and **Unit** for its type. In an *additive* tuple $\langle e_1, \ldots, e_n \rangle$, of type $\tau_1\ \&\ \cdots\ \&\ \tau_n$, just one component is computed depending on which one the context demands, and only the demanded one is consumed. Additive tuples also allow $n = 0$, but we don't need to notate that case in this paper.

Disjoint union types $\tau ::= c\ \tau \oplus \cdots \oplus c\ \tau$ and terms $e ::= c\ e$ are tagged with *constructors* c drawn from an infinite set C. In our example programs, **data** type declarations can be understood as introducing aliases for unions of multiplicative tuples. In Example 2.5 for instance, `Position` is an alias for $\texttt{0}\ \textbf{Unit} \oplus \texttt{1}\ \textbf{Unit} \oplus \texttt{2}\ \textbf{Unit} \oplus \texttt{3}\ \textbf{Unit}$, inhabited by terms like `2 ()`. In turn, `InputString` is an alias for $\texttt{InputNil}\ \textbf{Unit} \oplus \texttt{InputCons}\ (\texttt{Terminal} \otimes \texttt{Position})$, inhabited by terms like `InputCons (A (), 2 ())`. Here the constructors are `0`, `1`, `2`, `3`, `InputNil`, `InputCons`, and `A`.

3.1.4 Global Definitions. The **define** keyword introduces a global definition, which can recursively use any global variable. Because global variables are nonlinear, they allow us to express recursion, as Examples 2.2 to 2.6 demonstrated. Global variables are 'call-by-name' in the sense that they are bound to computations rather than values: each use of a global variable evaluates to a fresh copy of its definition's right-hand side. So the following program is well-typed, because each use of `f` creates a new λ-expression that is used once:

```
define f = λx. amb x (not x); (f true, f true)
```

But the following is not well-typed, because `g` is used twice.

```
define f = λx. amb x (not x); let g = f in (g true, g true)
```

And in the following program, `b` samples from **amb true false** each time it is evaluated (in other words, its nondeterminism is "hot"), so the program has four branches, not two:

```
define b = amb true false; (b, b)
```

3.1.5 Typing. The typing rules are shown in Figure 3. Typing judgements for expressions e are of the form $\Gamma; \Delta \vdash e : \tau$, where Γ and Δ are typing contexts for nonlinear and linear bindings, respectively. Nonlinear (or *intuitionistic*) bindings can be used any number of times, whereas linear bindings must each be used exactly once [Barber 1996]. In these typing rules, global definitions always

Variables

$$\frac{}{\Gamma, x : \tau; \cdot \vdash x : \tau} \qquad \frac{}{\Gamma; x : \tau \vdash x : \tau}$$

Functions

$$\frac{\Gamma; \Delta_0, x_1 : \tau_1 \vdash e' : \tau'}{\Gamma; \Delta_0 \vdash \lambda x_1.e' : \tau_1 \multimap \tau'} \qquad \frac{\Gamma; \Delta_0 \vdash e_0 : \tau_1 \multimap \tau' \qquad \Gamma; \Delta_1 \vdash e_1 : \tau_1}{\Gamma; \Delta_0, \Delta_1 \vdash e_0\ e_1 : \tau'}$$

Distributions

$$\frac{\Gamma; \Delta \vdash e_1 : \tau \qquad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathbf{amb}\ e_1\ e_2 : \tau} \qquad \frac{}{\Gamma; \Delta \vdash \mathbf{fail} : \tau} \qquad \frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{factor}\ w\ \mathbf{in}\ e : \tau}$$

Tuples

$$\frac{\Gamma; \Delta_i \vdash e_i : \tau_i}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash (e_1, \ldots, e_n) : \tau_1 \otimes \cdots \otimes \tau_n} \qquad \frac{\Gamma; \Delta \vdash e_i : \tau_i}{\Gamma; \Delta \vdash \langle e_1, \ldots, e_n \rangle : \tau_1 \,\&\, \cdots \,\&\, \tau_n}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 \otimes \cdots \otimes \tau_n \qquad \Gamma; \Delta', x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathbf{let}\ (x_1, \ldots, x_n) = e\ \mathbf{in}\ e' : \tau'} \qquad \frac{\Gamma; \Delta \vdash e : \tau_1 \,\&\, \cdots \,\&\, \tau_n}{\Gamma; \Delta \vdash e.i : \tau_i}$$

Unions

$$\frac{\Gamma; \Delta \vdash e_i : \tau_i}{\Gamma; \Delta \vdash c_i\ e_i : c_1\ \tau_1 \oplus \cdots \oplus c_n\ \tau_n} \qquad \frac{\Gamma; \Delta \vdash e : c_1\ \tau_1 \oplus \cdots \oplus c_n\ \tau_n \qquad \Gamma; \Delta', x_i : \tau_i \vdash e_i' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathbf{case}\ e\ \mathbf{of}\ c_1\ x_1 \to e_1' \mid \cdots \mid c_n\ x_n \to e_n' : \tau'}$$

Programs

$$\frac{\Gamma; \cdot \vdash e : \tau \qquad \Gamma \vdash p : \Gamma'; \tau'}{\Gamma \vdash (\mathbf{define}\ x = e\ ;\ p) : (x : \tau, \Gamma'); \tau'} \qquad \frac{\Gamma; \cdot \vdash e : \tau}{\Gamma \vdash e : \cdot; \tau} \qquad \frac{\Gamma \vdash p : \Gamma; \tau}{p : \tau}$$

Fig. 3. Typing rules. We write $\cdot$ for an empty typing context.

populate the nonlinear context, whereas local bindings only enter the linear context. It is useful in practical applications (including the examples in Section 2) to relax the linearity requirement to allow linear bindings to be used affinely (zero times or once), or to allow local nonlinear bindings under certain circumstances. Please see Appendix B for more details.

An **amb** expression can be used in any linear context, and both branches are type-checked in that linear context. Since **fail** is like **amb** but with 0 branches, it can also be used in any linear context.

The difference between multiplicative and additive tuples is reflected in their distinct typing rules. Whereas the components of an additive tuple $\langle e_1, \ldots, e_n \rangle$ are all type-checked in the same linear context Δ, the components of a multiplicative tuple $(e_1, \ldots, e_n)$ are type-checked by partitioning the linear context $\Delta_1, \ldots, \Delta_n$. Thus, the empty tuple $()$ requires the empty linear context.

For a program to type check under the global context Γ, the **define**s of the program must provide global variables with types exactly as promised in Γ. To enforce this consistency, the judgement for a program p takes the form $\Gamma \vdash p : \Gamma'; \tau$ to track the assumed context Γ and the provided context Γ' separately, and the judgement for a complete program $p : \tau$ requires Γ and Γ' to be identical.

Next, we give a standard operational semantics and a slightly nonstandard denotational semantics for PERPL, then prove that they agree thanks to the linear use of functions and additive tuples.

3.2 Operational Semantics

Our operational semantics is defined by a one-step reduction judgement $\gamma \vdash e \Longrightarrow E'$, shown in Figure 4. The expression being reduced is e. The reduction result E' is a distribution over expressions, written as a set of weight-expression pairs; this is used to reduce **amb**, **fail**, and **factor**. The global environment $\gamma = (x_1 = e_1, \ldots)$ maps global names to their definitions; this is used to reduce global names in (4a). Our evaluation contexts allow evaluation under λ and on both sides of application, so the order of probabilistic choices is left unspecified; this is benign, as denotations are preserved regardless of the order (Theorem 3.14).

$$\gamma \vdash x \Longrightarrow \{(1, e)\} \quad \text{if } (x = e) \in \gamma \tag{4a}$$

$$\gamma \vdash (\lambda x_1.e')\ v_1 \Longrightarrow \{(1, e'\{x_1 := v_1\})\} \tag{4b}$$

$$\gamma \vdash \mathtt{amb}\ e_1\ e_2 \Longrightarrow \{(1, e_1), (1, e_2)\} \tag{4c}$$

$$\gamma \vdash \mathtt{fail} \Longrightarrow \emptyset \tag{4d}$$

$$\gamma \vdash \mathtt{factor}\ w\ \mathtt{in}\ e \Longrightarrow \{(w, e)\} \tag{4e}$$

$$\gamma \vdash \mathtt{let}\ (x_1, \ldots, x_n) = (v_1, \ldots, v_n)\ \mathtt{in}\ e' \Longrightarrow \{(1, e'\{x_1 := v_1, \ldots, x_n := v_n\})\} \tag{4f}$$

$$\gamma \vdash \langle e_1, \ldots, e_n \rangle.i \Longrightarrow \{(1, e_i)\} \tag{4g}$$

$$\gamma \vdash \mathtt{case}\ c_i\ v\ \mathtt{of}\ c_1\ x_1 \to e_1' \mid \cdots \mid c_n\ x_n \to e_n' \Longrightarrow \{(1, e_i'\{x_i := v\})\} \tag{4h}$$

$$\frac{\gamma \vdash e \Longrightarrow E'}{\gamma \vdash C[e] \Longrightarrow \{(w, C[e']) \mid (w, e') \in E'\}} \tag{4i}$$

Syntactic values $v ::= \lambda x.e \mid (v, \ldots, v) \mid c\ v \mid \langle e, \ldots, e \rangle$

Evaluation contexts $C ::= [\cdot] \mid \lambda x : \tau.\, C \mid C\ e \mid e\ C \mid \mathtt{factor}\ w\ \mathtt{in}\ C$
$\mid (e, \ldots, e, C, e, \ldots, e) \mid \mathtt{let}\ (x, \ldots, x) = C\ \mathtt{in}\ e \mid \mathtt{let}\ (x, \ldots, x) = e\ \mathtt{in}\ C$
$\mid C.i \mid c\ C \mid \mathtt{case}\ C\ \mathtt{of}\ c_1\ x \to e \mid \cdots \mid c_n\ x \to e$

Fig. 4. Operational semantics.

If E is a distribution over expressions, then we can reduce E by reducing any element of E:

Definition 3.1. If $E = \{(w, e)\} + E_0$ and $\gamma \vdash e \Longrightarrow E'$, then $\gamma \vdash E \Longrightarrow w \cdot E' + E_0$ (where $\cdot$ and $+$ denote scaling and addition of distributions). We also write $\Longrightarrow^*$ for a sequence of zero or more reductions.

For example, starting with the expression $\mathtt{factor}\ w_1\ \mathtt{in}\ \mathtt{factor}\ w_2\ \mathtt{in}\ e$, we can apply Definition 3.1 to (4e) twice to get

$$\gamma \vdash \{(1, \mathtt{factor}\ w_1\ \mathtt{in}\ \mathtt{factor}\ w_2\ \mathtt{in}\ e)\} \Longrightarrow \{(w_1, \mathtt{factor}\ w_2\ \mathtt{in}\ e)\} \Longrightarrow \{(w_1 w_2, e)\}. \tag{3}$$

We conclude this section with a proof sketch of type soundness.

Lemma 3.2 (Linear substitution preserves typing). *Suppose* $\Gamma; \Delta_0, x_1 : \tau_1 \vdash e' : \tau'$ *and* $\Gamma; \Delta_1 \vdash e_1 : \tau_1$. *Then* $\Gamma; \Delta_0, \Delta_1 \vdash e'\{x_1 := e_1\} : \tau'$.

Proof. By induction on the typing derivation of e'. □

Definition 3.3. We say that a global environment $\gamma = (x_1 = e_1, \ldots)$ is *well-typed* for a context $\Gamma = (x_1 : \tau_1, \ldots)$ iff $\Gamma; \cdot \vdash e_i : \tau_i$ for all i.

Proposition 3.4 (Reduction preserves typing). *Let* γ *be well-typed for* Γ. *If* $\Gamma; \Delta \vdash e : \tau$ *and* $\gamma \vdash e \Longrightarrow E'$, *then* $\Gamma; \Delta \vdash e' : \tau$ *for all* $(w, e') \in E'$.

Proof. By induction on the derivation of $\gamma \vdash e \Longrightarrow E'$. Case (4a) uses the well-typedness of γ. Cases (4b), (4f), (4h) use Lemma 3.2. Case (4i) uses the induction hypothesis. □

Proposition 3.5 (Progress). *Let* γ *be well-typed for* Γ. *If* $\Gamma; \cdot \vdash e : \tau$ *then either* e *is a value or* $\gamma \vdash e \Longrightarrow E'$ *for some* E'.

Proof. By induction on the typing derivation of e. The benign nondeterminism mentioned above is not used. □

3.3 Denotational Semantics

In this section, we define the denotation of a complete program $p : \tau$, which is a distribution over the set denoted by the type τ. First we define a *distribution* over a set X to be a mapping $X \to [0, \infty]$. This makes sense in measure theory whenever X is countable; for us, X is always finite. Note that weights can be irrational or infinite (unlike in the syntax and operational semantics). If χ_1, χ_2 are distributions over X, we define the (complete) partial order $\chi_1 \leq \chi_2$ iff for all $x \in X$, $\chi_1(x) \leq \chi_2(x)$.

We handle recursive calls in our denotational semantics in a standard way. Without the global environment (our sole source of recursive calls), we would just let each expression e denote a mapping from environments to distributions over semantic values, where an environment maps each free variable of e to its semantic value. With the global environment, we have to first define this denotation relative to an interpretation of each global name as a distribution. Thus, the denotation of a program's global definitions maps an interpretation of each global name to another interpretation of each global name. Finally, we take the least fixed point of this monotonic map [Kozen 1981].

Definition 3.6. A type τ denotes a finite set $[\![\tau]\!]$ of *semantic values*, defined by induction on τ:

$$\begin{aligned}
[\![\tau_1 \otimes \cdots \otimes \tau_n]\!] &= [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \\
[\![c_1\ \tau_1 \oplus \cdots \oplus c_n\ \tau_n]\!] &= \{\, c_i\ v \mid 1 \leq i \leq n,\ v \in [\![\tau_i]\!] \,\} \\
[\![\tau_1 \multimap \tau_2]\!] &= \{\, v_1 \mapsto v_2 \mid v_1 \in [\![\tau_1]\!],\ v_2 \in [\![\tau_2]\!] \,\} \\
[\![\tau_1 \,\&\, \cdots \,\&\, \tau_n]\!] &= \{\, i : v \mid 1 \leq i \leq n,\ v \in [\![\tau_i]\!] \,\}
\end{aligned}$$

The first two cases are unsurprising: the tuple type $\tau_1 \otimes \cdots \otimes \tau_n$ denotes a set of tuples, and the union type $c_1\ \tau_1 \oplus \cdots \oplus c_n\ \tau_n$ denotes a disjoint union, whose elements $c_i\ v$ are just pairs of c_i and v.

The last two cases are where our denotational semantics is nonstandard. One might expect $[\![\tau_1 \multimap \tau_2]\!]$ to be the set of functions $[\![\tau_2]\!]^{[\![\tau_1]\!]}$. But the denotation of an expression of type $\tau_1 \multimap \tau_2$ will involve a distribution over $[\![\tau_1 \multimap \tau_2]\!]$, and that would have $|[\![\tau_2]\!]|^{|[\![\tau_1]\!]|}$ weights and lead to programs that require exponential time and space. Linearity permits a more efficient way. Intuitively, when a function is created, it nondeterministically guesses what (one) argument value it will receive. Later, applying the function (once) is just a matter of unifying the actual argument with this guess. The semantic value $v_1 \mapsto v_2$ is just a suggestively notated pair of v_1 and v_2. Hence, the cardinality $\big|[\![\tau_1 \multimap \tau_2]\!]\big|$ is only $\big|[\![\tau_1]\!]\big| \cdot \big|[\![\tau_2]\!]\big|$, enabling PERPL to handle many programs in polynomial time.

Similarly, when an additive tuple is created, it nondeterministically guesses which (one) component will be demanded. The semantic value $i : v$ is just a pair of i and v, and the cardinality $\big|[\![\tau_1 \,\&\, \cdots \,\&\, \tau_n]\!]\big|$ is not $\big|[\![\tau_1]\!]\big| \cdots \big|[\![\tau_n]\!]\big|$ but only $\big|[\![\tau_1]\!]\big| + \cdots + \big|[\![\tau_n]\!]\big|$.

Example 3.7. Since **Unit** is the type of 0-tuples, the set $[\![\textbf{Unit}]\!]$ is $\{()\}$. Since **Bool** is syntactic sugar for `True` **Unit** $\oplus$ `False` **Unit**, the set $[\![\textbf{Bool}]\!]$ is $\{\texttt{True}\ (), \texttt{False}\ ()\}$.

Recall from Section 3.1.3 that `Position` = `0` **Unit** $\oplus$ `1` **Unit** $\oplus$ `2` **Unit** $\oplus$ `3` **Unit**, so $[\![\texttt{Position}]\!] = \{\texttt{0}\ (), \texttt{1}\ (), \texttt{2}\ (), \texttt{3}\ ()\}$. Then $[\![\texttt{Position} \multimap \texttt{Position}]\!]$ has $4 \cdot 4 = 16$ members, not $4^4 = 256$:

$$\begin{aligned}
[\![\texttt{Position} \multimap \texttt{Position}]\!] = \big\{&(\texttt{0}\ (), \texttt{0}\ ()), (\texttt{0}\ (), \texttt{1}\ ()), (\texttt{0}\ (), \texttt{2}\ ()), (\texttt{0}\ (), \texttt{3}\ ()), \\
&(\texttt{1}\ (), \texttt{0}\ ()), (\texttt{1}\ (), \texttt{1}\ ()), (\texttt{1}\ (), \texttt{2}\ ()), (\texttt{1}\ (), \texttt{3}\ ()), \\
&(\texttt{2}\ (), \texttt{0}\ ()), (\texttt{2}\ (), \texttt{1}\ ()), (\texttt{2}\ (), \texttt{2}\ ()), (\texttt{2}\ (), \texttt{3}\ ()), \\
&(\texttt{3}\ (), \texttt{0}\ ()), (\texttt{3}\ (), \texttt{1}\ ()), (\texttt{3}\ (), \texttt{2}\ ()), (\texttt{3}\ (), \texttt{3}\ ())\big\}.
\end{aligned}$$

Furthermore, $[\![(\texttt{Position} \multimap \texttt{Position}) \multimap \texttt{Position}]\!]$ has $4 \cdot 4 \cdot 4 = 64$ members, not $4^{4^4} \approx 10^{154}$.

$$[\![x]\!]_\eta(\delta, v) = \begin{cases} \eta(x)(v) & x \in \operatorname{dom}\eta \\ \mathbb{I}[v = \delta(x)] & x \in \operatorname{dom}\delta \end{cases} \tag{5a}$$

$$[\![\lambda x_1.e']\!](\delta_0, v_1 \mapsto v') = [\![e']\!](\delta_0 \cup \{(x_1, v_1)\}, v') \tag{5b}$$

$$[\![e_0\ e_1]\!](\delta_0 \cup \delta_1, v') = \sum_{v_1 \in [\![\tau_1]\!]} [\![e_0]\!](\delta_0, v_1 \mapsto v') \cdot [\![e_1]\!](\delta_1, v_1) \tag{5c}$$

$$[\![\texttt{amb}\ e_1\ e_2]\!](\delta, v) = [\![e_1]\!](\delta, v) + [\![e_2]\!](\delta, v) \tag{5d}$$

$$[\![\texttt{fail}]\!](\delta, v) = 0 \tag{5e}$$

$$[\![\texttt{factor}\ w\ \texttt{in}\ e]\!](\delta, v) = w \cdot [\![e]\!](\delta, v) \tag{5f}$$

$$[\![(e_1, \ldots, e_n)]\!](\delta_1 \cup \cdots \cup \delta_n, (v_1, \ldots, v_n)) = [\![e_1]\!](\delta_1, v_1) \cdots [\![e_n]\!](\delta_n, v_n) \tag{5g}$$

$$[\![\texttt{let}\ (x_1, \ldots, x_n) = e\ \texttt{in}\ e']\!](\delta \cup \delta', v') = \sum_{v_1, \ldots, v_n} [\![e]\!](\delta, (v_1, \ldots, v_n)) \cdot [\![e']\!](\delta' \cup \{(x_1, v_1), \ldots, (x_n, v_n)\}, v') \tag{5h}$$

$$[\![\langle e_1, \ldots, e_n\rangle]\!](\delta, i : v) = [\![e_i]\!](\delta, v) \tag{5i}$$

$$[\![e.i]\!](\delta, v) = [\![e]\!](\delta, i : v) \tag{5j}$$

$$[\![c\ e]\!](\delta, c'\ v) = \begin{cases} [\![e]\!](\delta, v) & c' = c \\ 0 & c' \neq c \end{cases} \tag{5k}$$

$$[\![\texttt{case}\ e\ \texttt{of}\ c_1\ x_1 \to e'_1 \mid \ldots \mid c_n\ x_n \to e'_n]\!](\delta \cup \delta', v') = \sum_{i=1}^{n} \sum_{v} [\![e]\!](\delta, c_i\ v) \cdot [\![e'_i]\!](\delta' \cup \{(x_i, v)\}, v') \tag{5l}$$

Fig. 5. Denotational semantics. In $[\![e]\!](\delta, v)$, the domain of δ is always the set of free variables in e, so in (5c), the left-hand side uniquely determines δ_0 and δ_1 on the right-hand side, and similarly in (5g), (5h), and (5l).

A global typing context Γ denotes the set of all mappings from variables in Γ to distributions over semantic values. That is, $[\![\Gamma]\!]$ contains all possible η such that for each $x : \tau \in \Gamma$, we have a distribution $\eta(x) : [\![\tau]\!] \to [0, \infty]$. Thus, $[\![\Gamma]\!]$ is generally uncountable.

A local typing context Δ denotes the set of all mappings from variables in Δ to semantic values. That is, $[\![\Delta]\!]$ contains all possible δ such that if $x : \tau \in \Delta$, then $\delta(x) \in [\![\tau]\!]$. Thus, $[\![\Delta]\!]$ is a Cartesian product of finite sets and itself finite.

Example 3.8. In Example 2.3, the global typing context is $\Gamma = (\texttt{flip} : \textbf{Bool}, \texttt{gen} : \textbf{Unit})$. Since **Bool** has two values and **Unit** has one, each $\eta \in [\![\Gamma]\!]$ stores 3 weights:

$$\eta(\texttt{flip})(\texttt{True}\ ()) \in [0, \infty] \qquad \eta(\texttt{flip})(\texttt{False}\ ()) \in [0, \infty] \qquad \eta(\texttt{gen})(()) \in [0, \infty]$$

There are no local variables in Example 2.3, so the local typing context Δ is everywhere empty, and the unique $\delta \in [\![\Delta]\!]$ is also empty.

A typing judgement $\Gamma; \Delta \vdash e : \tau$ denotes a mapping that takes an $\eta \in [\![\Gamma]\!]$ to a distribution over $[\![\Delta]\!] \times [\![\tau]\!]$. We write this distribution as $[\![\Gamma; \Delta \vdash e : \tau]\!]_\eta$, but as $[\![e]\!]_\eta$ or even $[\![e]\!]$ for short, and we define it compositionally on the typing derivation, using the equations in Figure 5. The Iverson bracket $\mathbb{I}[\cdot]$ is 1 if its contents are true, 0 otherwise.

Example 3.9. Consider the expression **amb** (**factor** p **in true**) (**factor** q **in false**) from Example 2.1. We build up the denotation of this expression as follows.

$$[\![()]\!](\emptyset, ()) \overset{(5g)}{=} 1$$

$$[\![\textbf{true}]\!](\emptyset, c\,()) \overset{(5k)}{=} \mathbb{I}[c = \texttt{True}] \qquad [\![\textbf{false}]\!](\emptyset, c\,()) \overset{(5k)}{=} \mathbb{I}[c = \texttt{False}]$$
$$[\![\textbf{factor}\ p\ \textbf{in true}]\!](\emptyset, c\,()) \overset{(5f)}{=} p \cdot \mathbb{I}[c = \texttt{True}] \qquad [\![\textbf{factor}\ q\ \textbf{in false}]\!](\emptyset, c\,()) \overset{(5f)}{=} q \cdot \mathbb{I}[c = \texttt{False}]$$

$$[\![\textbf{amb}\ (\textbf{factor}\ p\ \textbf{in true})\ (\textbf{factor}\ q\ \textbf{in false})]\!](\emptyset, c\,()) \overset{(5d)}{=} \begin{cases} p & c = \texttt{True} \\ q & c = \texttt{False}. \end{cases} \tag{6}$$

Example 3.10. Moving on to Example 2.3, assume that η is given.

$$[\![\texttt{flip}]\!]_\eta(\emptyset, c\,()) \overset{(5a)}{=} \eta(\texttt{flip})(c\,())$$
$$[\![\texttt{gen}]\!]_\eta(\emptyset, ()) \overset{(5a)}{=} \eta(\texttt{gen})(())$$
$$[\![\textbf{let}\ () = \texttt{gen}\ \textbf{in}\ ()]\!]_\eta(\emptyset, ()) \overset{(5h)}{=} \eta(\texttt{gen})(()) \cdot 1$$
$$[\![\textbf{let}\ () = \texttt{gen}\ \textbf{in let}\ () = \texttt{gen}\ \textbf{in}\ ()]\!]_\eta(\emptyset, ()) \overset{(5h)}{=} \eta(\texttt{gen})(()) \cdot \eta(\texttt{gen})(()) \cdot 1$$
$$[\![\textbf{if}\ \texttt{flip}\ \textbf{then let}\ () = \texttt{gen}\ \textbf{in let}\ () = \texttt{gen}\ \textbf{in}\ ()\ \textbf{else}\ ()]\!]_\eta(\emptyset, ()) \overset{(5l)}{=} \eta(\texttt{flip})(\texttt{True}\,()) \cdot \eta(\texttt{gen})(()) \cdot \eta(\texttt{gen})(()) \cdot 1 + \eta(\texttt{flip})(\texttt{False}\,()) \cdot 1. \tag{7}$$

Given a set of global definitions $\gamma = (x_1 = e_1 : \tau_1, \ldots, x_n = e_n : \tau_n)$, we define its denotation $[\![\gamma]\!]$ to be the global environment η that is the least solution to the equations

$$\eta(x_i)(v_i) = [\![e_i]\!]_\eta(\emptyset, v_i) \tag{8}$$

for each $i = 1, \ldots, n$ and each v_i in $[\![\tau_i]\!]$. Finally, the program $\textbf{define}\ x_1 = e_1\ ;\ \ldots\ ;\ \textbf{define}\ x_n = e_n\ ;\ e$ denotes the distribution $[\![e]\!]_{[\![\gamma]\!]}(\emptyset, v)$ over semantic values v.

Example 3.11. For Examples 2.1 and 2.3, the equations are

$$\eta(\texttt{flip})(c\,()) \overset{(8)}{=} [\![\textbf{amb}\ (\textbf{factor}\ p\ \textbf{in true})\ (\textbf{factor}\ q\ \textbf{in false})]\!](\emptyset, c\,()) \tag{9}$$
$$\overset{(6)}{=} \begin{cases} p & c = \texttt{True} \\ q & c = \texttt{False} \end{cases} \tag{10}$$
$$\eta(\texttt{gen})(()) \overset{(8)}{=} [\![\textbf{if}\ \texttt{flip}\ \textbf{then let}\ () = \texttt{gen}\ \textbf{in let}\ () = \texttt{gen}\ \textbf{in}\ ()\ \textbf{else}\ ()]\!](\emptyset, ()) \tag{11}$$
$$\overset{(7)}{=} \eta(\texttt{flip})(\texttt{True}\,()) \cdot \eta(\texttt{gen})(()) \cdot \eta(\texttt{gen})(()) \cdot 1 + \eta(\texttt{flip})(\texttt{False}\,()) \cdot 1. \tag{12}$$

In all, there are three equations in three unknowns, $\eta(\texttt{flip})(\texttt{True}\,())$, $\eta(\texttt{flip})(\texttt{False}\,())$, and $\eta(\texttt{gen})(())$. The whole program in Example 2.3, of type **Unit**, denotes a single weight, which we write as z for short:

$$z = [\![\texttt{gen}]\!](\emptyset, ()) \overset{(5a)}{=} \eta(\texttt{gen})(()). \tag{13}$$

Putting (10), (12), and (13) together, we get Equation (2) as promised:

$$z = pz^2 + q. \tag{14}$$

3.4 Soundness

We justify our denotational semantics by showing that it is consistent with our standard operational semantics, thanks to the linear type system.

Definition 3.12. We say that the denotation $[\![\Gamma; \Delta \vdash e : \tau]\!]_\eta$ is *deterministic* if, for each $\delta \in [\![\Delta]\!]$, it assigns weight 1 to just one $v \in [\![\tau]\!]$ and weight 0 to all other $v \in [\![\tau]\!]$.

Lemma 3.13 (Linear substitution preserves denotation).
Suppose $\Gamma; \Delta_0, x_1 : \tau_1 \vdash e' : \tau'$ *and* $\Gamma; \Delta_1 \vdash e_1 : \tau_1$ *(as in Lemma 3.2). Then*

$$[\![e'\{x_1 := e_1\}]\!]_\eta(\delta_0 \cup \delta_1, v') = \sum_{v_1 \in [\![\tau_1]\!]} [\![e']\!]_\eta(\delta_0 \cup \{(x_1, v_1)\}, v') \cdot [\![e_1]\!]_\eta(\delta_1, v_1) \tag{15}$$

for all $\eta \in [\![\Gamma]\!]$, $\delta_0 \in [\![\Delta_0]\!]$, $\delta_1 \in [\![\Delta_1]\!]$, *and* $v' \in [\![\tau']\!]$.

Proof. By induction on the typing derivation of e'. See Appendix A.1 for more details. □

Theorem 3.14 (Reduction preserves denotation). *If $\Gamma; \Delta \vdash e : \tau$ and $\gamma \vdash e \Longrightarrow E'$ in the operational semantics, then in the denotational semantics, for all $\delta \in [\![\Delta]\!]$ and $v \in [\![\tau]\!]$,*

$$[\![e]\!]_{[\![\gamma]\!]}(\delta, v) = \sum_{(w,e') \in E'} w \cdot [\![e']\!]_{[\![\gamma]\!]}(\delta, v).$$

Proof. By induction on the reduction derivation of e. We show the case $\gamma \vdash (\lambda x_1.e')\, e_1 \Longrightarrow \{(1, e'\{x_1 := e_1\})\}$ where e_1 is a syntactic value:

$$\begin{aligned}
[\![(\lambda x_1.e')\, e_1]\!](\delta_0 \cup \delta_1, v') &\overset{(5c)}{=} \sum_{v_1} [\![\lambda x_1.e']\!](\delta_0, v_1 \mapsto v') \cdot [\![e_1]\!](\delta_1, v_1) \\
&\overset{(5b)}{=} \sum_{v_1} [\![e']\!](\delta_0 \cup \{(x_1, v_1)\}, v') \cdot [\![e_1]\!](\delta_1, v_1) \\
&= 1 \cdot [\![e'\{x_1 := e_1\}]\!](\delta_0 \cup \delta_1, v') \text{ by Lemma 3.13.}
\end{aligned}$$ □

4 INFERENCE

The denotation of a program under the above semantics is the least fixed point of a system of equations, which has a particular form that has been well studied in other contexts.

Definition 4.1. A *monotone system of polynomial equations*, or MSPE [Esparza et al. 2008; Etessami and Yannakakis 2009], is a system of equations

$$\begin{aligned}
z_1 &= P_1(z_1, \ldots, z_n) \\
&\vdots \\
z_n &= P_n(z_1, \ldots, z_n)
\end{aligned}$$

where the z_i are called *weight variables* (to distinguish them from variables in PERPL) and each P_i is a polynomial with nonnegative real coefficients. We write $\mathbf{z}$ for a vector in $[0, \infty]^n$ of assignments to the weight variables. If $\mathbf{z}, \mathbf{z}'$ are such assignments, we write $\mathbf{z} \le \mathbf{z}'$ iff for all i, $z_i \le z'_i$. We write $\mathbf{P}(\mathbf{z})$ for the vector $[P_i(z_1, \ldots, z_n)]_{i=1}^n \in [0, \infty]^n$. Thus the MSPE can be written succinctly as $\mathbf{z} = \mathbf{P}(\mathbf{z})$.

4.1 Constructing an MSPE

The first step of inference is to instantiate Figure 5 and Equation (8) for all the expressions of the program, which has the form of an MSPE.

Proposition 4.2. *The denotation of a program is a distribution whose weight values are components of the least solution (in $[0, \infty]^n$) of an MSPE.*

Proof. The MSPE has two kinds of weight variables:

(1) For every subexpression $\Gamma; \Delta \vdash e : \tau$, every $\delta \in [\![\Delta]\!]$, and every $v \in [\![\tau]\!]$, there is a weight variable $[\![e]\!]_\eta(\delta, v)$. Figure 5 gives an equation whose left-hand side is this weight variable and whose right-hand side is a polynomial in the weight variables with nonnegative coefficients. Examples of such equations are (6), (7), and (13).
(2) For every global variable $x : \tau$ and every $v \in [\![\tau]\!]$, there is a weight variable $\eta(x)(v)$. Equation (8) gives an equation whose left-hand side is this weight variable and whose right-hand side is a weight variable of the first kind. Examples of such equations are (9) and (11).

The denotation of a program **define** $x_1 = e_1$; ... ; **define** $x_n = e_n$; e with type τ is the distribution that assigns to each $v \in [\![\tau]\!]$ the weight $[\![e]\!](\emptyset; v)$, which is a weight variable of the first kind above.

All that needs to be shown is that the MSPE has finite size. Under Definition 3.6, $[\![\tau]\!]$ is finite for every τ, which can be shown by induction on the structure of τ. Also, $[\![\Delta]\!]$ is finite, being a finite Cartesian product of finite sets. So the number of weight variables is finite. □

Even though the number of weight variables is finite, it can be exponential in the size of the program, because $[\![\Delta]\!]$ is the product of as many sets as Δ has variables. If only for this reason, PERPL inference is intractable in general. This blowup is not surprising given that, even without any recursive calls or data, PERPL can easily express all discrete Bayes nets [Cooper 1990] and conjunctive queries [Chandra and Merlin 1977], and inherits their intractability.

Even for a family of programs—such as the typical parser—for which the number of weight variables is polynomial in the size of the program, an unwisely chosen dependency can dramatically increase the degree of the polynomial; this is a concern as well for the polynomial-time dynamic-programming inference algorithms that we are trying to recover automatically. As with Bayes nets and conjunctive queries, finding the optimal inference strategy (tree decomposition) is NP-hard, but many heuristics help in practice. Also, in practice the weight variables can be packed into a tensor to take advantage of vectorized parallelism.

4.2 Solving the MSPE

The general strategy for solving an MSPE automatically is to decompose it into smaller MSPEs. If z_1, z_2 are weight variables, we write $z_1 \prec z_2$ if there is an equation whose left-hand side is z_2 and whose right-hand side contains z_1. Then find the strongly connected components (SCCs) of $\prec$. In Example 3.11,

$$\eta(\texttt{flip})\big(\texttt{True}\,()\big) \prec \eta(\texttt{gen})\big(()\big) \quad \eta(\texttt{flip})\big(\texttt{False}\,()\big) \prec \eta(\texttt{gen})\big(()\big) \quad \eta(\texttt{gen})\big(()\big) \prec \eta(\texttt{gen})\big(()\big)$$

so each of the three variables forms its own SCC. We visit each SCC in topological order, solving it and substituting its solution into the remaining equations.

The easiest, and most common, case is when an SCC has just one weight variable, whose equation (after substituting earlier weight variables) must be of the form $z = w$ where w is a constant. Many existing PPLs with exact inference handle only this case, that is, when $\prec$ is acyclic.

The next easiest case is when an SCC has more than one weight variable but (after substituting earlier weight variables) its equations are all linear. These can be solved *directly*—as opposed to iteratively—in the semiring $[0, \infty]$ by one of the algorithms of Lehmann [1977], such as Gaussian elimination. This case includes all loops (Example 2.2) and arises often in practice; for example, PCFG rules of the form $(X \to X)$ can be used an unbounded number of times. Most existing parsers limit how many times such rules can be applied [e.g., Collins 1999; Finkel et al. 2008; Taskar et al. 2004], but PERPL makes it easy to write code that handles such situations exactly.

The most general case is when the equations for the SCC are nonlinear. An example is parsing using a PCFG with rules of the form $(X \to \epsilon)$. Again, implementations usually resort to arbitrary limits on how such rules can be applied [e.g., Cai et al. 2011]. But these cases can be solved using a generalization of Newton's method to ω-continuous semirings [Esparza et al. 2007, 2010], which is guaranteed to converge to the least solution of an MSPE:

$$\begin{aligned}\mathbf{z}^{(0)} &= \mathbf{0}\\ \mathbf{z}^{(i+1)} &= \mathbf{z}^{(i)} + \left(\tfrac{\partial \mathbf{P}}{\partial \mathbf{z}}\left(\mathbf{z}^{(i)}\right)\right)^{*}\left(\mathbf{P}(\mathbf{z}^{(i)}) - \mathbf{z}^{(i)}\right)\end{aligned}$$

where $\frac{\partial \mathbf{P}}{\partial \mathbf{z}}$ is the *formal* derivative of $\mathbf{P}$ (that is, defined only using the sum and product rules, not using limits). For any matrix $\mathbf{A}$, the *closure* $\mathbf{A}^* = \sum_{i=0}^{\infty} \mathbf{A}^i$ can be computed by an algorithm of Lehmann [1977]. And $\infty - \infty$ can be defined to be anything (say, zero).

Although the iterates of Newton's method are in general only approximations, there is a known number of iterations that is guaranteed to reach any desired level of accuracy [Stewart et al. 2015]. This is also true of, for example, logarithms, and in the context of machine learning, such computations are generally considered "exact," in contrast to methods based on random sampling.

Example 4.3. In Equation (14), let $p = \frac{2}{3}$ and $q = \frac{1}{3}$. Then $P(z) = \frac{2}{3}z^2 + \frac{1}{3}$ and $\frac{\partial P}{\partial z}(z) = \frac{4}{3}z$, and using Newton's method to solve $z = P(z)$ gives

$$\begin{aligned}
z^{(0)} &= 0 \\
z^{(1)} &= 0 + 0^* \left(\tfrac{1}{3} - 0\right) = \tfrac{1}{3} \approx 0.33333 \\
z^{(2)} &= \tfrac{1}{3} + \left(\tfrac{4}{9}\right)^* \left(\tfrac{11}{27} - \tfrac{1}{3}\right) = \tfrac{7}{15} \approx 0.46667 \\
z^{(3)} &= \tfrac{7}{15} + \left(\tfrac{28}{45}\right)^* \left(\tfrac{323}{675} - \tfrac{7}{15}\right) = \tfrac{127}{255} \approx 0.49804 \\
z^{(4)} &= \tfrac{127}{255} + \left(\tfrac{508}{765}\right)^* \left(\tfrac{97283}{195075} - \tfrac{127}{255}\right) = \tfrac{32767}{65536} \approx 0.49999
\end{aligned}$$

and so on. But suppose $p = q = 1$. Then $P(z) = z^2 + 1$ and $\frac{\partial P}{\partial z}(z) = 2z$, and Newton's method gives

$$\begin{aligned}
z^{(0)} &= 0 \\
z^{(1)} &= 0 + 0^*(1 - 0) = 1 \\
z^{(2)} &= 1 + 2^*(2 - 1) = \infty \\
z^{(3)} &= \infty + \infty^*(\infty - \infty) = \infty.
\end{aligned}$$

Accordingly, our implementation actually outputs `inf`.

In machine learning applications, it is often useful to optimize a quantity involving the distribution computed by a program, by adjusting parameters that the program depends on. In Example 2.4 for instance, having computed the likelihood L of some observed strings, we might want to maximize it by adjusting p. To adjust p by gradient descent, we can differentiate the MSPE denoted by the program with respect to p and then solve the resulting (linear) system of equations for $\frac{\mathrm{d}L}{\mathrm{d}p}$.

4.3 Factor Graph Grammars

Our implementation actually does not compile PERPL programs directly to MSPEs, but to *factor graph grammars* (FGGs) [Chiang and Riley 2020], whose inference algorithm constructs an MSPE in turn and solves it while taking advantage of vectorized parallelism. FGGs are a formalism for describing probability models on recursive structures that is more general than factor graphs, case–factor diagrams [McAllester et al. 2008] and sum–product networks [Poon and Domingos 2011]. Details of the translation to FGGs are in Appendix C.

5 RECURSIVE TYPES

In this section, we extend the language to allow recursive types (Section 5.1), and then, because our denotational semantics does not include recursive types and because known exact inference methods do not support recursive types, we show how to eliminate them (Sections 5.2 to 5.5).

5.1 Definitions

As shown in Figure 6, we add iso-recursive types [Pierce 2002] with introduction and elimination forms **`fold`** and **`unfold`** (also sometimes called **`roll`** and **`unroll`**; they do not mean catamorphism and anamorphism). Our definition of recursive types has two slightly unusual features. First, **`unfold`** expressions have a scope e', which will be used below (Section 5.2.2) when eliminating recursive types. Invariably, e' is a **`case`** expression, so we often write **`case unfold`** e **`of`** ... as shorthand for **`unfold`** $x = e$ **`in case`** x **`of`** ....

Second, the superscript t is a *tag* drawn from any infinite set such as $\mathbb{N}$. It is used to distinguish recursive types that would otherwise be considered equal, which is sometimes necessary for eliminating them (illustrated in Section 5.4 below). As with nominal typing, we consider two

$$\begin{array}{ll}
\text{Syntax} & e ::= \mathbf{fold}_{\mu^t\alpha.\tau}\ e \mid \mathbf{unfold}_{\mu^t\alpha.\tau}\ x = e\ \mathbf{in}\ e' \qquad \tau ::= \mu^t\alpha.\,\tau \mid \alpha \\
\text{Typing} & \dfrac{\Gamma;\Delta \vdash e : \tau\{\alpha := \mu^t\alpha.\tau\}}{\Gamma;\Delta \vdash \mathbf{fold}_{\mu^t\alpha.\tau}\ e : \mu^t\alpha.\,\tau} \qquad \dfrac{\Gamma;\Delta \vdash e : \mu^t\alpha.\,\tau \qquad \Gamma;\Delta', x : \tau\{\alpha := \mu^t\alpha.\tau\} \vdash e' : \tau'}{\Gamma;\Delta,\Delta' \vdash \mathbf{unfold}_{\mu^t\alpha.\tau}\ x = e\ \mathbf{in}\ e' : \tau'} \\
\text{Reduction} & \gamma \vdash \mathbf{unfold}_{\mu^t\alpha.\tau}\ x = (\mathbf{fold}_{\mu^t\alpha.\tau}\ v)\ \mathbf{in}\ e' \Longrightarrow \{(1, e'\{x := v\})\} \qquad (16) \\
\text{Syntactic values} & v ::= \mathbf{fold}_{\mu^t\alpha.\tau}\ v \\
\text{Evaluation contexts} & C ::= \mathbf{fold}\ C \mid \mathbf{unfold}\ x = C\ \mathbf{in}\ e \mid \mathbf{unfold}\ x = e\ \mathbf{in}\ C
\end{array}$$

Fig. 6. Syntax and semantics of recursive types.

recursive types that differ only in their tag to be two different types. Additionally, we require that different recursive types must have different tags; that is, if the program contains both $\mu^t\alpha.\tau_1$ and $\mu^t\alpha.\tau_2$, then $\tau_1 = \tau_2$. (This requirement is easy to satisfy, and used in Lemma A.12.) Tags can be inferred by using a different tag variable for each occurrence of μ, and unifying tag variables during type checking as necessary. (Our implementation of PERPL also infers tag polymorphism in global definitions and datatypes, by a straightforward generalization of Hindley–Milner–Damas type inference. It then eliminates this inferred polymorphism by monomorphization.)

Example 5.1. In Examples 5.2 and 5.3, we will transform the following simple program to eliminate the recursive type $\texttt{Nat} \equiv \mu^t\alpha.\,(\texttt{Zero}\ \mathbf{Unit} \oplus \texttt{Succ}\ \alpha)$. This program samples a natural number n from an exponential distribution (by flipping a coin repeatedly and counting how many flips are **true** before the first **false**), and tests whether n is odd.

```
data Nat = Zero | Succ Nat
define sample : Nat = if flip then fold (Succ sample) else fold Zero
define odd (n: Nat) : Bool =
  unfold n′ = n in case n′ of Zero ⟶ false | Succ m ⟶ not (odd m)
odd sample
```

5.2 Eliminating Recursive Types

Recursive types such as `Nat` above cannot be handled by the conversion to an MSPE (Section 4) because they would give rise to an infinite number of weight variables. However, they can often be transformed away. In this section, we define these transformations and show how they work on Example 5.1. In Section 5.3 and Appendix A.3, we argue that they preserve program meaning, even in the presence of probabilistic effects. In Section 5.5, we provide a greedy algorithm that finds a successful sequence of transformations whenever one exists.

Our transformations are closely related to defunctionalization and refunctionalization, which are well-known. However, our formulation is slightly nonstandard, and our usage of them in a language with probabilistic effects is novel. Most notably, whereas traditional defunctionalization often introduces a recursive type, our usage of it eliminates a recursive type.

5.2.1 Defunctionalization. Suppose we want to eliminate a recursive type $\sigma = \mu^t\alpha.\,\bar{\sigma}$ from a given program. Let the n occurrences of $\mathbf{fold}_\sigma$ in the program be $\mathbf{fold}_\sigma\ m_1, \ldots, \mathbf{fold}_\sigma\ m_n$. For each i, let the nonglobal free variables of m_i form the tuple $\vec{y}_i : \vec{\varphi}_i$. We define a transformation $\mathcal{D}_\sigma[\![\cdot]\!]$ that changes terms of type σ to type $\varphi = \texttt{Fold}_1\ \vec{\varphi}_1 \oplus \cdots \oplus \texttt{Fold}_n\ \vec{\varphi}_n$, where $\texttt{Fold}_1, \ldots, \texttt{Fold}_n$ are constructors. Because our purpose is to eliminate σ, it doesn't help to replace σ by another type that contains σ, and replacing σ in φ would cause infinite regress, so we require that φ not contain σ.

On types, our transformation $\mathcal{D}_\sigma[\![\cdot]\!]$ just changes occurrences of σ to φ, so it leaves $\vec{\varphi}_i$ unchanged. On programs, it adds a global function u_σ (u stands for unfold), of type $\varphi \multimap \bar{\sigma}\{\alpha := \varphi\}$:

$$\textbf{define}\ u_\sigma = \lambda x\colon \varphi.\ \textbf{case}\ x\ \textbf{of}\ \mathtt{Fold}_1\ \vec{y}_1 \to \mathcal{D}_\sigma[\![m_1]\!] \mid \cdots \mid \mathtt{Fold}_n\ \vec{y}_n \to \mathcal{D}_\sigma[\![m_n]\!] \tag{17}$$

On terms, it postpones from $\textbf{fold}_\sigma$ to $\textbf{unfold}_\sigma$ the work done by u_σ:

$$\mathcal{D}_\sigma[\![\textbf{fold}_\sigma\ m_i]\!] = \mathtt{Fold}_i\ \vec{y}_i \qquad \mathcal{D}_\sigma[\![\textbf{unfold}_\sigma\ x = e\ \textbf{in}\ e']\!] = \textbf{let}\ x = u_\sigma\ \mathcal{D}_\sigma[\![e]\!]\ \textbf{in}\ \mathcal{D}_\sigma[\![e']\!] \tag{18}$$

The other cases are uninteresting.

If the m_i are all abstractions, and if the e' in each $\textbf{unfold}_\sigma\ x = e\ \textbf{in}\ e'$ is an application of x, then the $\mathcal{D}_\sigma$ transformation is equivalent to defunctionalization [Danvy and Nielsen 2001; Reynolds 1972], with u_σ usually called $\mathtt{apply}_\sigma$. Although the original purpose of defunctionalization was to get rid of λ-abstractions by postponing them to their applications, in general it can be used to get rid of any introduction forms by postponing them to their elimination forms; here, we use it to get rid of **fold**s by postponing them to their **unfold**s.

Example 5.2. In Example 5.1, there are two occurrences of **fold**, inside `sample`. Since neither of them has any free variables, $\varphi = \mathtt{Fold}_1\ \textbf{Unit} \oplus \mathtt{Fold}_2\ \textbf{Unit}$ and so we can defunctionalize `Nat`. Create new types `NatF` and `NatU`, which take the roles of φ and $\bar{\sigma}\{\alpha := \varphi\}$ above, respectively:

```
data NatF = Fold₁ | Fold₂
data NatU = Zero | Succ NatF
```

The bodies of the **fold** expressions move into a new function `u_Nat`, which plays the role of u_σ:

```
define u_Nat (n: NatF) : NatU =
  case n of Fold₁ → Succ sample | Fold₂ → Zero
```

The rest of the program is transformed by Equation (18):

```
define sample : NatF = if flip then Fold₁ else Fold₂
define odd (n: NatF) : Bool =
  let n′ = u_Nat n in case n′ of Zero → false | Succ m → not (odd m)
odd sample
```

5.2.2 Refunctionalization. Again suppose we want to eliminate a recursive type $\sigma = \mu^t \alpha.\bar{\sigma}$ from a given program. Instead of moving the work done at **fold**ing, we can move the work done at **unfold**ing. Without loss of generality, assume that every $\textbf{unfold}_\sigma$ expression binds the same variable name x, and let the n occurrences of $\textbf{unfold}_\sigma$ in the program be $\textbf{unfold}_\sigma\ x = \cdots\ \textbf{in}\ m_i$, where $1 \le i \le n$. For each i, let the nonglobal free variables of the body $m_i : \varphi_i$, except x, form the tuple $\vec{y}_i : \vec{\varphi}_i$. We define a transformation $\mathcal{R}_\sigma[\![\cdot]\!]$ that changes terms of type σ to type

$$\varphi = (\vec{\varphi}_1 \multimap \varphi_1)\ \&\ \cdots\ \&\ (\vec{\varphi}_n \multimap \varphi_n). \tag{19}$$

Again because our purpose is to eliminate σ, we require that φ not contain σ.

On types, the transformation $\mathcal{R}_\sigma[\![\cdot]\!]$ just changes occurrences of σ to φ, so it leaves $\vec{\varphi}_i$ and φ_i unchanged. On programs, it adds a global function f_σ (f stands for fold), of type $\bar{\sigma}\{\alpha := \varphi\} \multimap \varphi$:

$$\textbf{define}\ f_\sigma = \lambda x\colon \bar{\sigma}\{\alpha := \varphi\}.\ \langle \lambda \vec{y}_1.\ \mathcal{R}_\sigma[\![m_1]\!], \ldots, \lambda \vec{y}_n.\ \mathcal{R}_\sigma[\![m_n]\!] \rangle \tag{20}$$

On terms, it moves from $\textbf{unfold}_\sigma$ to $\textbf{fold}_\sigma$ the work done by f_σ:

$$\mathcal{R}_\sigma[\![\textbf{fold}_\sigma\ e]\!] = f_\sigma\ \mathcal{R}_\sigma[\![e]\!] \qquad \mathcal{R}_\sigma[\![\textbf{unfold}_\sigma\ x = e\ \textbf{in}\ m_i]\!] = (\mathcal{R}_\sigma[\![e]\!].i)\ \vec{y}_i \tag{21}$$

This is essentially the same as refunctionalization [Danvy and Millikin 2009; Danvy and Nielsen 2001]; in Danvy and Millikin's terminology, we have *disentangled* the $\textbf{unfold}_\sigma$ expressions into the

apply functions $\lambda\vec{y}_i.\, \mathcal{R}_\sigma[\![m_i]\!]$ and *merged* them using an additive tuple (which works even if they have different types). The original definition of refunctionalization assumed that $\bar{\sigma}$ was a union type and created a function for each case; we create a single function f_σ for the same purpose.

Example 5.3. In Example 5.1, there is just one **unfold** expression. It has type **Bool**, and its body has no free variables other than `n'`. So instead of defunctionalizing `Nat`, we can also refunctionalize it. Create new types `NatF` and `NatU`, which take the roles of φ and $\bar{\sigma}\{\alpha := \varphi\}$ above, respectively:

```
type NatF = Unit ⊸ Bool
data NatU = Zero | Succ NatF
```

(According to Equation (19), `NatF` should be a unary additive tuple type, which has no notation in this paper. To keep the presentation simple, we just omit unary tuple types here.)

The body of the **unfold** expression moves into a new function `f_Nat`, which plays the role of f_σ:

```
define f_Nat (n': NatU) : NatF =
  λ(): Unit. case n' of Zero → false | Succ m → not (odd m)
```

The rest of the program is transformed by Equation (21):

```
define sample : NatF = if flip then f_Nat (Succ sample) else f_Nat Zero
define odd (n: NatF) : Bool = n ()
odd sample
```

5.3 Correctness

It is easy to show that the transforms $\mathcal{D}$ and $\mathcal{R}$ preserve types; see Appendix A.2. In Appendix A.3, we show that they furthermore preserve the overall meaning of a program (a distribution over **Unit**, without loss of generality). Intuitively, this is because they merely change where the work done by u_σ or f_σ is expressed: $\mathcal{D}_\sigma$ moves the work done by u_σ from $\mathbf{fold}_\sigma$ to $\mathbf{unfold}_\sigma$, whereas $\mathcal{R}_\sigma$ moves the work done by f_σ from $\mathbf{unfold}_\sigma$ to $\mathbf{fold}_\sigma$.

5.4 Example: CFG Parsing

We next show how defunctionalization and refunctionalization together enable PERPL to handle some programs that neither transformation alone can. Specifically, we eliminate recursive types from the CFG parser in Example 2.4. This larger example demonstrates how the classic CYK algorithm is recovered, as well as some additional aspects of the transformations.

We repeat Example 2.4 below, with a few more details spelled out. First, we write **fold**s and **unfold**s explicitly. Second, as mentioned above, tag inference automatically distinguishes two uses of `String`, which we call $\texttt{String}^1$ and $\texttt{String}^2$. If it were not for tags, we would not be able to handle this program: it would not be possible to defunctionalize `String` because of the free variable `acc` in **fold** `(Cons A acc)`, and it would not be possible to refunctionalize `String` either because of the free variable `xs` inside the second **case unfold** `ys` expression.

Finally, some variables are used nonlinearly. For the nonrecursive type `Terminal`, this is harmless; Appendix B.1 discusses how to handle it. But for recursive types, we need to add a `discard` function to turn non-use into linear use; Appendix B.2 shows how this can be done automatically instead.

```
data String¹ = Nil¹ | Cons¹ Terminal String¹
data String² = Nil² | Cons² Terminal String²

define gen (lhs: Nonterminal) (acc: String¹) : String¹ =
  case lhs of S → if flip then gen S (gen S acc) else fold (Cons¹ A acc)

define equal (xs: String¹) (ys: String²) : Bool =
```

```
  case unfold xs of
    Nil¹ ⟶ case unfold ys of Nil² ⟶ true | Cons² y ys ⟶ discard² ys
    Cons¹ x xs ⟶ case unfold ys of Nil² ⟶ discard¹ xs
                                   Cons² y ys ⟶ x = y and equal xs ys
define discard¹ (xs: String¹) : Bool =
  case unfold xs of Nil¹ ⟶ false | Cons¹ x xs ⟶ discard¹ xs
define discard² (ys: String²) : Bool =
  case unfold ys of Nil² ⟶ false | Cons² y ys ⟶ discard² ys
equal (gen S (fold Nil¹))
      (fold (Cons² A (fold (Cons² A (fold (Cons² A (fold Nil²)))))))
```

There are four occurrences of $\mathbf{fold}_{\texttt{String}^2}$, all on the last line. Since none of them have any free variables, we can defunctionalize $\texttt{String}^2$. Create new types $\texttt{String}^2\texttt{F}$ and $\texttt{String}^2\texttt{U}$, which take the roles of φ and $\bar{\sigma}\{\alpha := \varphi\}$ in Section 5.2.1:

```
data String²F = Fold₁ | Fold₂ | Fold₃ | Fold₄
data String²U = Nil² | Cons² Terminal String²F
```

The four values of type $\texttt{String}^2\texttt{F}$ can be thought of as positions between the symbols of the input string *aaa* (including the beginning and end of the string). The new function $\texttt{u\_String}^2$ can then be thought of as taking a string position and returning the input symbol at that position, together with its successor position.

```
define u_String² (ys: String²F) : String²U =
  case ys of Fold₁ ⟶ Cons² A Fold₂ | Fold₂ ⟶ Cons² A Fold₃
             Fold₃ ⟶ Cons² A Fold₄ | Fold₄ ⟶ Nil²
```

Functions gen and $\texttt{discard}^1$ remain the same, but equal, $\texttt{discard}^2$, and the last line become

```
define equal (xs: String¹) (ys: String²F) : Bool =
  case unfold xs of
    Nil¹ ⟶ case u_String² ys of Nil² ⟶ true | Cons² y ys ⟶ discard² ys
    Cons¹ x xs ⟶ case u_String² ys of Nil² ⟶ discard¹ xs
                                      Cons² y ys ⟶ x = y and equal xs ys
define discard² (ys: String²F) : Bool =
  case u_String² ys of Nil² ⟶ false | Cons² y ys ⟶ discard² ys
equal (gen S (fold Nil¹)) Fold₁
```

As for $\texttt{String}^1$, it cannot be defunctionalized, because $\mathbf{fold}$ $(\texttt{Cons}^1\ \texttt{A}\ \texttt{acc})$ has a free variable acc of type $\texttt{String}^1$. Can it be refunctionalized? There are two $\mathbf{case\ unfold}_{\texttt{String}^1}$ expressions, one in equal and one in $\texttt{discard}^1$. Both bodies have type $\mathbf{Bool}$; the first has a free variable ys with type $\texttt{String}^2\texttt{F}$, and the second has no free variables. So yes, we can refunctionalize $\texttt{String}^1$ by creating new types $\texttt{String}^1\texttt{F}$ and $\texttt{String}^1\texttt{U}$, which take the roles of φ and $\bar{\sigma}\{\alpha := \varphi\}$ in Section 5.2.2:

```
type String¹F = (String²F ⊸ Bool) & (Unit ⊸ Bool)
data String¹U = Nil¹ | Cons¹ Terminal String¹F
```

A $\texttt{String}^1\texttt{F}$ can be thought of as a string accessed through two "methods" [cf. Rendel et al. 2015]: the first (with type $\texttt{String}^2\texttt{F} \multimap \texttt{Bool}$) compares it with a suffix of the input string, and the second (with type $\texttt{Unit} \multimap \texttt{Bool}$) always returns $\mathbf{false}$. These methods are implemented in the new function $\texttt{f\_String}^1$:

```
define f_String¹ (xs: String¹U) : String¹F =
  ⟨λys: String²F. case xs of
    Nil¹ ⟶ case u_String² ys of Nil² ⟶ true | Cons² y ys ⟶ discard² ys
    Cons¹ x xs ⟶ case u_String² ys of Nil² ⟶ discard¹ xs
                                      Cons² y ys ⟶ x = y and equal xs ys,
   λ(): Unit. case xs of Nil¹ ⟶ false | Cons¹ x xs ⟶ discard¹ xs⟩
define equal (xs: String¹F) (ys: String²F) : Bool = xs.1 ys
define discard¹ (xs: String¹F) : Bool = xs.2 ()
```

And occurrences of $\mathbf{fold}_{\texttt{String}^1}$ are changed to calls to `f_String`^{1}.

```
define gen (lhs: Nonterminal) (acc: String¹F) : String¹F =
  case lhs of S ⟶ if flip then gen S (gen S acc) else f_String¹ (Cons¹ A acc)
equal (gen S (f_String¹ Nil¹)) Fold₁
```

No recursive types remain, so this program converts to an MSPE with a finite number of weight variables. For an input string $w = w_1 \cdots w_n$, there are $O(n^2)$ equations in $O(n^2)$ weight variables, and each equation has $O(n)$ terms. They can be ordered so that each weight variable depends only on earlier variables, and solving them is equivalent to the CYK algorithm.

To show this equivalence with less clutter, we define

$$\overline{\ell} \;=\; 1 : (\texttt{Fold}_\ell\,() \mapsto \texttt{True}\,()) \;\in\; [\![\texttt{String}^1\texttt{F}]\!] \qquad 1 \le \ell \le n+1. \tag{22}$$

The set $[\![\texttt{String}^1\texttt{F}]\!] = [\![(\texttt{String}^2\texttt{F} \multimap \texttt{Bool}) \;\&\; (\texttt{Unit} \multimap \texttt{Bool})]\!]$ has $2n+4$ possible values (not $O(2^n)$, thanks to our treatment of functions as nondeterministic pairs). Of these, the $\overline{\ell}$ are the interesting ones; each corresponds to a generated suffix that is equal to the input suffix starting at position ℓ. Thus, it will be easiest to think of them as string positions.

Then the equations simplify to

$$\eta(\texttt{gen})(\texttt{S}\,() \mapsto \overline{k} \mapsto \overline{i}) = \sum_j p \cdot \eta(\texttt{gen})(\texttt{S}\,() \mapsto \overline{j} \mapsto \overline{i}) \cdot \eta(\texttt{gen})(\texttt{S}\,() \mapsto \overline{k} \mapsto \overline{j}) + q \cdot \eta(\texttt{f\_String}^1)(\texttt{Cons}\,(\texttt{A}\,(), \overline{k}) \mapsto \overline{i}) \tag{23}$$

$$\eta(\texttt{f\_String}^1)(\texttt{Cons}\,(\texttt{A}\,(), \overline{k}) \mapsto \overline{i}) = \mathbb{I}[w_i = a \wedge i = k-1] \tag{24}$$

and the probability of the whole program being true is $\eta(\texttt{gen})(\texttt{S}\,() \mapsto \overline{n+1} \mapsto \overline{1})$. Equation (24) corresponds to the initialization in CYK; it gives weight 1 to occurrences of terminal symbols. Equation (23) corresponds to the main triple loop in CYK; it computes the weight of all derivations $S \Rightarrow^* w_i \cdots w_{k-1}$.

5.5 Inferring the Sequence of Transformations

In the previous section, we saw that not all recursive types are amenable to both defunctionalization and refunctionalization. Furthermore, some transformations can preclude others. For example, even though `String`^{2} could be refunctionalized, it would preclude refunctionalizing `String`^{1} because, in function `equal`, it would change the type of variable `ys` from `String`^{2} to `(Unit ⊸ Bool) & ((Terminal ⊗ String`^{1}`) ⊸ Bool) & (Unit ⊸ Bool)`. This type in turn contains `String`^{1}, and `ys` occurs free within the **case unfold**$_{\texttt{String}^1}$ `xs` expression. Although it may seem that such dependencies between transformations would make it difficult to find a successful sequence of transformations that eliminates all recursive types, the good news is that there is a simple greedy algorithm for deciding whether a successful sequence of transformations exists, and if so, to find it.

While there are any recursive types remaining:

- If there is a recursive type σ such that $\mathcal{D}_\sigma[\![\sigma]\!]$ contains no recursive type, defunctionalize σ.
- If there is a recursive type σ such that $\mathcal{R}_\sigma[\![\sigma]\!]$ contains no recursive type, refunctionalize σ.
- Else, there is no successful sequence of transformations.

We give the formal statement of correctness here and defer its proof to the appendix.

Definition 5.4. A *DR-sequence* S for a program p is a sequence $\mathcal{F}^{(1)}, \ldots, \mathcal{F}^{(n)}$, where each $\mathcal{F}^{(i)}$ is either $\mathcal{D}_\sigma$ or $\mathcal{R}_\sigma$ for some recursive type σ. We say that S is *successful* if it is empty and p has no recursive types, or if $S = \mathcal{F} \cdot S'$, where $\mathcal{F}[\![p]\!]$ is well-defined (that is, $\mathcal{F}[\![\sigma]\!]$ does not contain σ) and S' is a successful DR-sequence for $\mathcal{F}[\![p]\!]$.

Proposition 5.5. *Let p be a program with at least one recursive type and a successful DR-sequence.*

(1) *There is a recursive type σ and a transformation $\mathcal{F} \in \{\mathcal{D}, \mathcal{R}\}$ such that $\mathcal{F}_\sigma[\![\sigma]\!]$ contains no recursive type.*
(2) *For any such σ and $\mathcal{F}$, the program $\mathcal{F}_\sigma[\![p]\!]$ is well-defined and has a successful DR-sequence.*

Proof. See Appendix A.4. □

A note on complexity: The transformations $\mathcal{D}$ and $\mathcal{R}$ only move code around, so any DR-sequence will not make the program much bigger. However, some successful DR-sequences may lead to more efficient inference than others. Moreover, there exist programs whose size is blown up exponentially by the monomorphization that needs to take place before the DR-sequence.

6 FURTHER EXAMPLES: PUSHDOWN AUTOMATA

PERPL automates the derivation of nonobvious inference algorithms from natural probabilistic models expressed using unbounded recursion. We've seen already how Section 5.4 derived the CYK algorithm from code that generates strings from a PCFG and compares them with an input string. In this section, we present three further examples, all related to pushdown automata (PDAs), to illustrate what kinds of recursive data structures PERPL can and cannot handle.

6.1 One Stack

The parser of Section 5.4 could also be implemented by converting the weighted CFG to a weighted nondeterministic PDA and running the PDA. Define a type for stacks:

```
data Stack = StkNil | StkCons Nonterminal Stack
```

(Our implementation of PERPL supports polymorphic datatypes such as `List` and monomorphizes their uses such as `List Nonterminal`.) The stack is initially just one `S`. The `run_pda` function below runs a nondeterministic PDA on a string. This PDA keeps no state, and accepts by empty stack.

```
define run_pda (zs: Stack) (ws: String) : Bool =
  case unfold zs of
    StkNil → case unfold ws of Nil → true                          -- accept
                               Cons w ws → discard ws              -- reject
    StkCons z zs → z = S and                                       -- pop S
      if flip then
        run_pda (fold (StkCons S (fold (StkCons S zs)))) ws        -- push SS
      else
        case unfold ws of Nil → discard_stk zs                     -- reject
                          Cons w ws → w = A and run_pda zs ws       -- scan a
```

```
define discard (ws: String) =
  case unfold ws of Nil ⟶ false | Cons w ws ⟶ discard ws
define discard_stk (zs: Stack) =
  case unfold zs of StkNil ⟶ false | StkCons z zs ⟶ discard_stk zs

run_pda (fold (StkCons S (fold StkNil)))
        (fold (Cons A (fold (Cons A (fold (Cons A (fold Nil)))))))
```

For simplicity, we have hard-coded the PDA into this program; it would be easy to add more symbols and transitions.

Unlike our previous examples using recursive data, a program that uses a stack like this is not structured as a producer–consumer pipeline. Nevertheless, `String` can be defunctionalized as in Section 5.4, yielding the following, where `StringF`, `StringU`, `u_String`, and `discard` are as before:

```
define run_pda (zs: Stack) (ws: StringF) : Bool =
  case unfold zs of
    StkNil ⟶ case u_String ws of Nil ⟶ true                          -- accept
                                 Cons w ws ⟶ discard ws              -- reject
    StkCons z zs ⟶ z = S and                                         -- pop S
      if flip then
        run_pda (fold (StkCons S (fold (StkCons S zs)))) ws          -- push SS
      else
        case u_String ws of Nil ⟶ discard_stk zs                     -- reject
                            Cons w ws ⟶ w = A and run_pda zs ws       -- scan a

run_pda (fold (StkCons S (fold StkNil))) Fold1
```

Although `Stack` cannot be defunctionalized because there are two occurrences of $\mathbf{fold}_{\texttt{Stack}}$ with a free variable `zs` of type `Stack`, the two occurrences of $\mathbf{unfold}_{\texttt{Stack}}$ meet the criterion for refunctionalization. So we refunctionalize `Stack` by creating new types `StackF` and `StackU`:

```
type StackF = (StringF ⊸ Bool) & (Unit ⊸ Bool)
data StackU = StkNil | StkCons Nonterminal StackF

define f_Stack (zs: StackU) : StackF =
   ⟨λws: StringF. case zs of
       StkNil ⟶ case u_String ws of Nil ⟶ true                          -- accept
                                    Cons w ws ⟶ discard ws              -- reject
       StkCons z zs ⟶ z = S and                                         -- pop S
         if flip then
           run_pda (f_Stack (StkCons S (f_Stack (StkCons S zs)))) ws    -- push SS
         else
           case u_String ws of Nil ⟶ discard_stk zs                     -- reject
                               Cons w ws ⟶ w = A and run_pda zs ws,      -- scan a
    λ(): Unit. case zs of StkNil ⟶ false | StkCons z zs ⟶ discard_stk zs⟩

define run_pda (zs: StackF) (ws: StringF) : Bool = zs.1 ws
define discard_stk (zs: StackF) : Bool = zs.2 ()

run_pda (f_Stack (StkCons S (f_Stack StkNil))) Fold1
```

Solving the corresponding MSPE is similar to Lang's algorithm for PDA recognition [Lang 1974]; as a matter of fact, refunctionalization has rederived a faster version of the algorithm found only recently [Butoi et al. 2022]. As observed by Danvy and Millikin [2009], this final program resembles the final program in Section 5.4, but written in continuation-passing style: the `Stack` data structure has been replaced by `StackF`, the type of continuations expecting a `StringF` or a `Unit`.

6.2 Two Stacks

An example of a program where defunctionalization and refunctionalization fail would be one that processes strings while maintaining two stacks instead of one. Since a two-stack PDA is equivalent to a Turing machine, such a program cannot be tractable, and it is desirable for PERPL to reject it at compile time. We omit a full program listing, but it would have a form like:

```
define run_2pda (left: Stack¹) (right: Stack²) (ws: String) : Bool =
  case unfold left of
    ··· case unfold right of
          ··· case unfold ws of
                ··· fold (StkCons¹ z left) ···  -- push onto left stack
                ··· fold (StkCons² z right) ··· -- push onto right stack
```

As in Section 6.1, the push operations render both `Stack` types non-defunctionalizable. Moreover, here **`case unfold`** `left` has free variable `right`, so refunctionalizing `Stack`^{1} would change it into a type containing `Stack`^{2}, while **`case unfold`** `right` has free variable `left`, so refunctionalizing `Stack`^{2} would change it into a type containing `Stack`^{1}. Thus, by Proposition 5.5, neither type is refunctionalizable.

6.3 A Stack of Stacks

In contrast to the two-stack PDA in Section 6.2, if multiple stacks are nested, refunctionalization does succeed, and it turns out to convert nested stacks into nested continuations. *Embedded PDAs* (EPDAs) are a generalization of PDAs where memory is structured not as a stack of symbols but as a stack of *stacks* of symbols [Vijayashanker 1994]. They are equivalent to tree-adjoining grammars (TAGs), which generalize CFGs. Moreover, TAGs can be parsed using a CYK-style algorithm [Vijay-Shankar and Joshi 1985]. Both of these results were major achievements in 1980s computational linguistics. Yet we show below that PERPL automatically takes a program that runs an EPDA and converts it into equations for a CYK-style TAG parser [cf. Alonso et al. 2000].

In the following code, we append a `*` to names relating to the stack-of-stacks.

```
data Stack  = StkNil  | StkCons  Nonterminal Stack
data Stack* = StkNil* | StkCons* Stack       Stack*
```

Whenever the top stack is empty, it is automatically popped. A move of the EPDA consists of popping a symbol `z` from the stack and then doing one of the following:

```
data Action =
  Pop                                -- do nothing further
  Scan Terminal                      -- scan terminal from input string
  Push Nonterminal Nonterminal       -- push y then x onto top stack
  PushAbove Nonterminal Nonterminal  -- push y then push stack [x] above top stack
  PushBelow Nonterminal Nonterminal  -- push x then push stack [y] below top stack
```

We assume a function called `transition` with type `Nonterminal ⊸ Action`. The following function decides whether the EPDA accepts a string. To sidestep the complication of explicit `discard` functions, it simply returns () if the EPDA accepts and **fail**s otherwise.

```
define run_epda (zs*: Stack*) (ws: String) : Unit = case zs* of
  StkNil* ⟶ case ws of Nil ⟶ () | Cons w ws ⟶ fail
  StkCons* z* zs* ⟶ case z* of
    StkNil ⟶ run_epda zs* ws
    StkCons z zs ⟶ case transition z of
      Pop ⟶ run_epda (StkCons* zs zs*) ws
      Scan a ⟶ case ws of Nil ⟶ fail
                          Cons w ws ⟶ if w = a then run_epda (StkCons* zs zs*) ws
                                                else fail
      Push x y ⟶ run_epda (StkCons* (StkCons x (StkCons y zs)) zs*) ws
      PushAbove x y ⟶
        run_epda (StkCons* (StkCons x StkNil) (StkCons* (StkCons y zs) zs*)) ws
      PushBelow x y ⟶
        run_epda (StkCons* (StkCons x zs) (StkCons* (StkCons y StkNil) zs*)) ws
run_epda (StkCons* (StkCons Z StkNil) StkNil*) (Cons A (Cons A (Cons A Nil)))
```

Like in Section 6.1, we defunctionalize `String` and refunctionalize `Stack` and `Stack*`, then simplify:

```
type StackF = Stack*F ⊸ StringF ⊸ Unit
type Stack*F = StringF ⊸ Unit

define f_StkNil* : Stack*F = λws: StringF. case u_String ws of
  Nil ⟶ () | Cons w ws ⟶ fail

define f_StkNil : StackF = λzs*: Stack*F. λws: StringF. zs* ws
define f_StkCons (z: Nonterminal) (zs: StackF) : StackF =
  λzs*: Stack*F. λws: StringF. case transition z of
    Pop ⟶ zs zs* ws
    Scan a ⟶ case u_String ws of Nil ⟶ fail
                                 Cons w ws ⟶ if w = a then zs zs* ws else fail
    Push x y ⟶ f_StkCons x (f_StkCons y zs) zs* ws
    PushAbove x y ⟶ f_StkCons x f_StkNil (f_StkCons y zs zs*) ws
    PushBelow x y ⟶ f_StkCons x zs (f_StkCons y f_StkNil zs*) ws

f_StkCons Z f_StkNil f_StkNil* Fold1
```

As before, `StringF` can be thought of as a position in the input string, as can `Stack*F`. Then `StackF` can be thought of as a pair of string positions. The most computationally expensive expression is `f_StkCons x (f_StkCons y zs)`, which represents $O(n^4)$ weight variables because it and its free variable `zs` both have type `StackF`. Each of these weight variables is a sum over $O(n^2)$ possible values of the subexpression `f_StkCons y zs`. Thus the program denotes a system of equations of size $O(n^6)$. Readers familiar with TAG will recognize $O(n^6)$ as the time complexity of CYK-style TAG parsing. Indeed, as we detail in Appendix D, when this program is converted to an FGG, the FGG is essentially a TAG, and PERPL's inference amounts to CYK-style TAG parsing.

Above, we noted that the result of refunctionalizing Section 6.1 was written in continuation-passing style. The twice-refunctionalized program above is written in extended continuation-passing

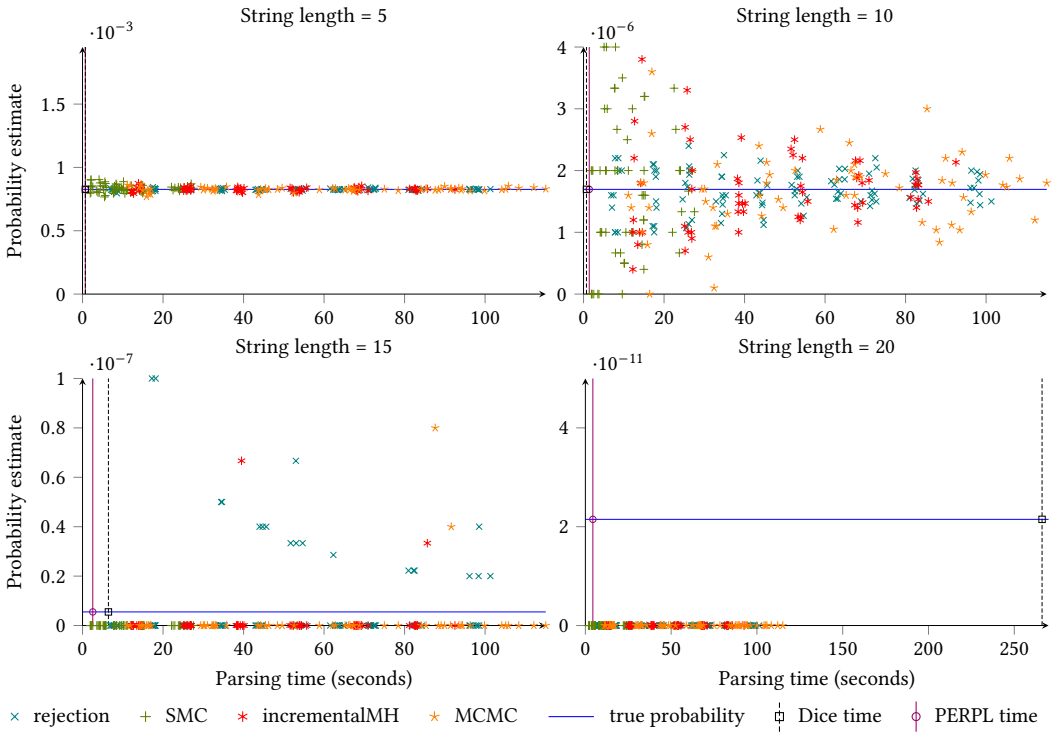

Fig. 7. The WebPPL version of Example 2.4 (PCFG parsing), using a variety of inference methods, displays high variance for all but the shortest strings. Dice and PERPL obtain exact probabilities much faster. For string length 15, one data point with probability estimate $3 \cdot 10^{-7}$ has been omitted. We ran rejection sampling up to 50 M samples, SMC up to 3 M particles, and incrementalMH and MCMC up to 30 M samples.

style [Danvy and Filinski 1990], with `StackF` being the type of continuations and `Stack*F` being the type of metacontinuations. Moreover, generalizations of EPDAs to automata with k-nested stacks (so-called k-EPDAs) can be expressed by similar PERPL programs and successfully compiled. We hence conjecture more generally that PERPL's program transformations relate the *control hierarchy* of context-free grammars [Weir 1992] to the (serendipitously same-named) *control hierarchy* of continuations [Danvy and Filinski 1990; Sitaram and Felleisen 1990].

7 BENCHMARK

We empirically demonstrate the improved speed and accuracy enabled by PERPL, by comparing against two existing probabilistic languages: WebPPL [Goodman and Stuhlmüller 2014], which supports unbounded recursive calls and data, but does not perform exact inference on them; and Dice [Holtzen et al. 2020], which performs exact inference, but only supports bounded loops. Because so much of PERPL's expressive power lies in its novel support for unbounded recursive data, it is tricky to find a benchmark that allows a quantitative comparison. We took the PCFG parsing problem in Example 2.4 and expressed it as idiomatically as we could (Appendix E): in WebPPL using unbounded recursion, and in Dice by unrolling a loop that builds up long parses from shorter ones. All experiments were performed on a 2.8 GHz CPU with 16 GB of RAM.

Comparison with approximate inference. Figure 7 shows that general-purpose approximate inference algorithms in WebPPL are inadequate for PCFG parsing, especially as the string parsed gets longer. Each scatterplot depicts the result of trying to parse a different string length, and each point represents one inference run. The horizontal axis shows the time taken; variation is caused primarily by different inference methods and different sample sizes. The vertical axis shows the probability estimated; variation is caused by random sampling, which has trouble explaining low-probability events: when the string length is 15, all probability estimates are either 0 or a wild overestimate, and when the string length exceeds 19, all estimates are 0. In contrast, Dice and PERPL produce answers within 0.000001% of the truth (shown by horizontal lines), in a fraction of the time taken by WebPPL (shown by vertical lines; also see below).

Comparison with exact inference. Figure 8 shows that PERPL scales better than Dice to parsing longer strings. The vertical axis shows time in log scale. Along the horizontal axis, we varied the string length between 1 and 100, but Dice ran out of memory at length 22. The plot shows that the running time of Dice (not to mention WebPPL) grows much more quickly than that of PERPL.

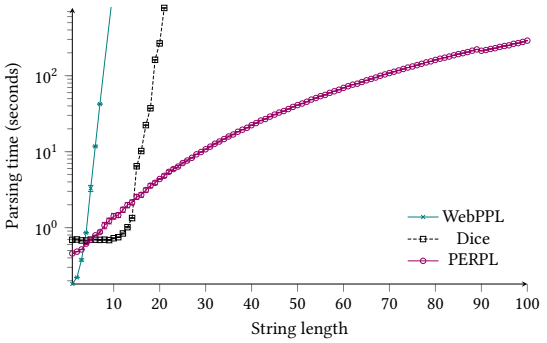

Fig. 8. The PCFG parser of Section 5.4 scales to longer strings much better than its Dice equivalent. Parsing time (y-axis) is on a logarithmic scale. The WebPPL curve shows how long it takes rejection sampling just to get estimates within 5% of the truth with probability 95%.

8 RELATED WORK

Exact inference and recursion. PERPL appears to be the first PPL that performs exact inference while allowing both unbounded recursive calls and *unbounded recursive data*. Here we consider other PPLs that support exact inference.

The "stochastic Lisp" of Koller et al. [1997] computes exact distributions by a generalization of variable elimination. Many PPLs that followed enable exact inference via graph representations. IBAL [Pfeffer 2001] compiles to a factor graph; Fun [Borgström et al. 2011] compiles to factor graphs with gates [Minka and Winn 2008]. Dice [Holtzen et al. 2020] and BernoulliProb [Claret et al. 2013] compile to binary/algebraic decision diagrams [Bahar et al. 1997; Bryant 1986], which are also used by the probabilistic model checker PRISM [Kwiatkowska et al. 2011]. SPPL [Saad et al. 2021] compiles to sum-product networks [Poon and Domingos 2011], and FSPN [Stuhlmüller and Goodman 2012] generalizes sum-product networks to represent recursive dependencies. PSI [Gehr et al. 2016, 2020] and ProbZelus [Atkinson et al. 2022] perform exact inference symbolically. Finally, the expectations—and more generally *moments*—of distributions can be computed automatically by representing them with polynomials, interval arithmetic, and linear recurrences [Bartocci et al. 2019; Bouissou et al. 2016; Moosbrugger et al. 2022; Sankaranarayanan et al. 2020].

Most of these PPLs, to our knowledge, do not support unbounded recursive calls or loops, in the sense of allowing programs where for any N, there is a branch of computation with recursion/iteration depth greater than N and nonzero weight. There are four exceptions. BernoulliProb handles loops using fixed-point iteration (whereas PERPL always compiles loops to linear equations then solves them directly). FSPN handles recursive calls as in PERPL, by solving a system of equations. ProbZelus handles a *stream* of observations by maintaining a symbolic state, which can represent and update a distribution exactly. The methods for computing moments just mentioned are focused on unbounded loops, but do not handle other recursive calls such as those in a PCFG.

Most of these PPLs, to our knowledge, do not allow unbounded recursive data such as strings, trees, or stacks. Stochastic Lisp does, evaluating them lazily so they can be of unbounded or infinite size without necessarily causing nontermination. However, Pfeffer [2005] later noted that the interaction between lazy evaluation and memoization was problematic. This line of research continued in IBAL with a new evaluation strategy for lazy memoization, but later languages in this line, such as Figaro [Pfeffer 2016], supported exact inference only for programs with bounded recursive calls and data. Our use of defunctionalization can also be seen as lazy evaluation and covers similar cases to those that stochastic Lisp and IBAL intended to. But because it requires delayed computations to depend only on finite types, memoization is straightforward. Also, lazy evaluation would not turn programs using stacks into polynomial-time algorithms; refunctionalization does.

Probabilistic function semantics. The semantics of first-class probabilistic functions has long been known to require more than endowing a domain of functions with measurable-space structure [Aumann 1961]. Compared to recent semantics of probabilistic functions and recursion using quasi-Borel spaces [Heunen et al. 2017] and logical relations [Wand et al. 2018], our elementary treatment is limited to models where inference without sampling is possible, by solving MSPEs.

Linear logic. We build on linear logic [Girard 1987; Walker 2005] in several ways. First, the contrast between "eager" evaluation of $\otimes$ and "lazy" evaluation of $\&$ draws on the typical computational interpretation of linear logic [Abramsky 1993]. Second, polarity [Girard 1991], which we use to allow local nonlinear bindings (Appendix B.1), has been used to structure the type system of a PPL [Ehrhard and Tasson 2019]. Finally, our treatment of linearly used functions as nondeterministic pairs, which dates back to encoding λ-calculus variables as Prolog variables [Pereira and Shieber 1987], is reminiscent of the symmetric monoidal closed category $\mathcal{R}^{\Pi}$ of Laird et al. [2013], whose tensor product $\otimes$ as well as exponential $\multimap$ are just the Cartesian product of sets. Whereas the PPLs in this literature [Ehrhard et al. 2018; Ehrhard and Tasson 2019; Laird 2020; Laird et al. 2013] allow functions to be used any number of times and interpret them by infinitary means, we use linearity to *limit* expressivity to finitely many weight variables, reducing inference to solving an MSPE.

De- and refunctionalization. De- and refunctionalization [Danvy and Millikin 2009; Danvy and Nielsen 2001; Rendel et al. 2015; Reynolds 1972] are not new, but using defunctionalization to eliminate non-functions appears novel. Defunctionalization has been proven correct before [Banerjee et al. 2001; Nielsen 2000; Pottier and Gauthier 2006], but not for a language with effects other than nontermination. Refunctionalization is defined as the (left) inverse of defunctionalization, so the correctness of one follows from that of the other. However, our $\mathcal{R}$ transform incorporates additional steps of *disentangling* and *merging apply functions* presented by Danvy and Millikin [2009].

9 CONCLUSION

PERPL allows programmers to write probabilistic code using unbounded recursive calls and data and still maintain exact inference. We have seen that the compilation to MSPEs can automatically derive inference algorithms that originally took significant intellectual effort to discover.

We have implemented a PERPL compiler in Haskell, including nonlinear bindings for positive types (Appendix B.1), affine bindings for all types (Appendix B.2), and defunctionalization and refunctionalization (Sections 5.2 and 5.5). Rather than compile directly to an MSPE, it compiles to an FGG (Appendix C.2). Our implementation of FGGs, which employs the inference methods of Section 4.2 and makes them automatically differentiable using PyTorch [Paszke et al. 2019], will be the subject of a future paper. Both implementations are released as open-source software.[1]

[1] PERPL: https://github.com/diprism/perpl FGGs: https://github.com/diprism/fggs

REFERENCES

Samson Abramsky. 1993. Computational Interpretations of Linear Logic. *Theoretical Computer Science* 111, 1–2 (1993), 3–57. https://doi.org/10.1016/0304-3975(93)90181-R

Miguel A. Alonso, Éric Villemonte de la Clergerie, and Manuel Vilares. 2000. A redefinition of Embedded Push-Down Automata. In *Proc. International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+5)*. 19–26. https://aclanthology.org/W00-2002

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. *PACM on Programming Languages* 6, OOPSLA2, Article 184 (2022), 29 pages. https://doi.org/10.1145/3563347

Robert J. Aumann. 1961. Borel Structures for Function Spaces. *Illinois Journal of Mathematics* 5, 4 (1961), 614–630. https://doi.org/10.1215/ijm/1255631584

R. I[ris] Bahar, E[rica] A. Frohm, C[harles] M. Gaona, G[ary] D. Hachtel, E[nrico] Macii, A[belardo] Pardo, and F[abio] Somenzi. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design* 10 (1997), 171–206. https://doi.org/10.1023/A:1008699807402

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 2001. Design and Correctness of Program Transformations Based on Control-Flow Analysis. In *Theoretical Aspects of Computer Software: TACS 2001 (LNCS, 2215)*. Springer, 420–447. https://doi.org/10.1007/3-540-45500-0_21

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic*. Technical Report ECS-LFCS-96-347. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh. https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/

Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *Automated Technology for Verification and Analysis: ATVA 2019 (LNCS, Vol. 11781)*. Springer, 255–276. https://doi.org/10.1007/978-3-030-31784-3_15

Michel Bauderon and Bruno Courcelle. 1987. Graph Expressions and Graph Rewritings. *Mathematical Systems Theory* 20 (1987), 83–127. https://doi.org/10.1007/BF01692060

Taylor L. Booth and Richard A. Thompson. 1973. Applying Probability Measures to Abstract Languages. *IEEE Trans. Comput.* C-22, 5 (1973), 442–450. https://doi.org/10.1109/T-C.1973.223746

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure Transformer Semantics for Bayesian Machine Learning. In *Programming Languages and Systems: ESOP 2011 (LNCS, Vol. 6602)*. Springer, 77–96. https://doi.org/10.1007/978-3-642-19718-5_5

Olivier Bouissou, Eric Goubault, Sylvie Putot, Aleksandar Chakarov, and Sriram Sankaranarayanan. 2016. Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities. In *Tools and Algorithms for the Construction and Analysis of Systems: TACAS 2016 (LNCS, Vol. 9636)*. Springer, 225–243. https://doi.org/10.1007/978-3-662-49674-9_13

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

Wray L. Buntine. 1994. Operations for Learning with Graphical Models. *J. Artificial Intelligence Research* 2 (1994), 159–225.

Alexandra Butoi, Brian DuSell, Tim Vieira, Ryan Cotterell, and David Chiang. 2022. Algorithms for Weighted Pushdown Automata. In *Proc. EMNLP*. https://aclanthology.org/2022.emnlp-main.656

Shu Cai, David Chiang, and Yoav Goldberg. 2011. Language-Independent Parsing with Empty Elements. In *Proc. NAACL HLT*. 212–216. https://aclanthology.org/P11-2037

Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proc. STOC*. 77–90. https://doi.org/10.1145/800105.803397

Stanley F. Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394. https://doi.org/10.1006/csla.1999.0128

David Chiang and Darcey Riley. 2020. Factor Graph Grammars. In *Proc. NeurIPS*. 6648–6658. https://proceedings.neurips.cc/paper/2020/hash/49ca03822497d26a3943d5084ed59130-Abstract.html

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Proc. ESEC/FSE 2013*. 92–102. https://doi.org/10.1145/2491411.2491423

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph. D. Dissertation. University of Pennsylvania.

Gregory F. Cooper. 1990. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence* 42, 2 (1990), 393–405. https://doi.org/10.1016/0004-3702(90)90060-D

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proc. ACM Conference on Lisp and Functional Programming (LFP)*. 151–160. https://doi.org/10.1145/91556.91622

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at Work. *Science of Computer Programming* 74, 8 (2009), 534–549. https://doi.org/10.1016/j.scico.2007.10.007

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proc. 3rd International Conference on Principles and Practice of Declarative Programming*. 162–174. https://doi.org/10.1145/773184.773202

Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge Replacement Graph Grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, Grzegorz Rozenberg (Ed.). World Scientific, 95–162. https://doi.org/10.1142/9789812384720_0002

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2018. Full Abstraction for Probabilistic PCF. *J. ACM* 65, 4, Article 23 (2018), 44 pages. https://doi.org/10.1145/3164540

Thomas Ehrhard and Christine Tasson. 2019. Probabilistic Call by Push Value. *Logical Methods in Computer Science* 15, 1, Article 3 (2019), 46 pages. https://doi.org/10.23638/LMCS-15(1:3)2019

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2007. On Fixed Point Equations over Commutative Semirings. In *Proc. STACS*. 296–307. https://doi.org/10.1007/978-3-540-70918-3_26

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2008. Solving Monotone Polynomial Equations. In *Proc. IFIP International Conference on Theoretical Computer Science (TCS)*. 285–298. https://doi.org/10.1007/978-0-387-09680-3_20 Invited paper.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian Program Analysis. *J. ACM* 57, 6, Article 33 (2010), 47 pages. https://doi.org/10.1145/1857914.1857917

Kousha Etessami and Mihalis Yannakakis. 2009. Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. *J. ACM* 56, 1, Article 1 (Feb. 2009), 66 pages. https://doi.org/10.1145/1462153.1462154

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. 2008. Efficient, Feature-based, Conditional Random Field Parsing. In *Proc. ACL: HLT*. 959–967. https://aclanthology.org/P08-1109

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification: CAV 2016 (LNCS, Vol. 9779)*. 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Proc. PLDI*. 883–897. https://doi.org/10.1145/3385412.3386006

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard. 1991. A New Constructive Logic: Classic Logic. *Mathematical Structures in Computer Science* 1, 3 (1991), 255–296. https://doi.org/10.1017/S0960129500001328

Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org

Alex Graves, Santiago Fernández, and Faustino Gomez. 2006. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. In *Proc. ICML*. 369–376.

Annegret Habel and Hans-Jörg Kreowski. 1987. May we introduce to you: Hyperedge replacement. In *Graph Grammars and Their Application to Computer Science: Graph Grammars 1986 (LNCS, Vol. 291)*. Springer, 15–26. https://doi.org/10.1007/3-540-18771-5_41

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. In *Proc. LICS*. 1–12. https://dl.acm.org/doi/10.5555/3329995.3330072

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *PACM on Programming Languages* 4, OOPSLA, Article 140 (2020), 31 pages. https://doi.org/10.1145/3428208

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. https://arxiv.org/abs/1508.01991 arXiv:1508.01991v1.

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Daphne Koller, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian Inference for Stochastic Programs. In *Proc. AAAI*. 740–747. https://www.aaai.org/Library/AAAI/1997/aaai97-115.php

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. 2001. Factor Graphs and the Sum-Product Algorithm. *IEEE Trans. Information Theory* 47, 2 (2001), 498–519. https://doi.org/10.1109/18.910572

Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification (CAV) (LNCS, Vol. 6806)*. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proc. ICML*. 282–289.

James Laird. 2020. Weighted Models for Higher-Order Computation. *Information and Computation* 275, Article 104645 (2020), 34 pages. https://doi.org/10.1016/j.ic.2020.104645

Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. 2013. Weighted Relational Models of Typed Lambda-Calculi. In *Proc. LICS*. 301–310. https://doi.org/10.1109/LICS.2013.36

Bernard Lang. 1974. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Proc. Colloquium on Automata, Languages, and Programming*. 255–269. https://doi.org/10.1007/978-3-662-21545-6_18

Bernard Lang. 1994. Recognition Can Be Harder Than Parsing. *Computational Intelligence* 10, 4 (1994), 486–494. https://doi.org/10.1111/j.1467-8640.1994.tb00011.x

Daniel J. Lehmann. 1977. Algebraic Structures for Transitive Closure. *Theoretical Computer Science* 4, 1 (1977), 59–76. https://doi.org/10.1016/0304-3975(77)90056-1

Jonathan May and Kevin Knight. 2006. Tiburon: A Weighted Tree Automata Toolkit. In *Proc. Conference on Implementation and Application of Automata (CIAA)*. 102–113. https://doi.org/10.1007/11812128_11

David McAllester, Michael Collins, and Fernando Pereira. 2008. Case-Factor Diagrams for Structured Probabilistic Modeling. *J. Comput. System Sci.* 74, 1 (2008), 84–96. https://doi.org/10.1016/j.jcss.2007.04.015

Tom Minka and John Winn. 2008. Gates. In *Proc. NeurIPS*. 1073–1080. https://papers.nips.cc/paper/3379-gates

Mehryar Mohri. 1997. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics* 23, 2 (1997), 269–311. https://aclanthology.org/J97-2003

Marcel Moosbrugger, Miroslav Stanković, Ezio Bartocci, and Laura Kovács. 2022. This is the Moment for Probabilistic Loops. *PACM on Programming Languages* 6, OOPSLA2, Article 178 (2022), 29 pages. https://doi.org/10.1145/3563341

Lasse R. Nielsen. 2000. *A Denotational Investigation of Defunctionalization.* Report RS-00-47. BRICS. https://www.brics.dk/RS/00/47/

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. NeurIPS*. 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

Fernando C. N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural-Language Analysis.* Center for the Study of Language and Information.

Avi Pfeffer. 2001. IBAL: A Probabilistic Rational Programming Language. In *Proc. IJCAI*. 733–740.

Avi Pfeffer. 2005. *The Design and Implementation of IBAL: A General-Purpose Probabilistic Language.* Technical Report TR-12-05. Harvard Computer Science Group. http://nrs.harvard.edu/urn-3:HUL.InstRepos:25105000

Avi Pfeffer. 2016. *Practical Probabilistic Programming.* Manning.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

Hoifung Poon and Pedro Domingos. 2011. Sum-Product Networks: A New Deep Architecture. In *Proc. UAI*. 337–346. https://arxiv.org/abs/1202.3732

François Pottier and Nadji Gauthier. 2006. Polymorphic Typed Defunctionalization and Concretization. *Higher-Order and Symbolic Computation* 19, 1 (March 2006), 125–162. https://doi.org/10.1007/s10990-006-8611-7

Lawrence R. Rabiner. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE* 77, 2 (Feb. 1989), 257–286. https://doi.org/10.1109/5.18626

Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem. In *Proc. ICFP*. 269–279. https://doi.org/10.1145/2784731.2784763

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proc. ACM*. 717–740. https://doi.org/10.1145/800194.805852

Michael Riley, Cyril Allauzen, and Martin Jansche. 2009. OpenFst: An Open-Source, Weighted Finite-State Transducer Library and its Applications to Speech and Language. In *Proc. NAACL HLT, Companion Volume: Tutorial Abstracts*. 9–10. https://aclanthology.org/N09-4005

Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Proc. PLDI*. 804–819. https://doi.org/10.1145/3453483.3454078

Sriram Sankaranarayanan, Yi Chou, Eric Goubault, and Sylvie Putot. 2020. Reasoning about Uncertainties in Discrete-Time Dynamical Systems using Polynomial Forms. In *Proc. NeurIPS*, Vol. 33. 17502–17513. https://proceedings.neurips.cc/paper/2020/hash/ca886eb9edb61a42256192745c72cd79-Abstract.html

Dorai Sitaram and Matthias Felleisen. 1990. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation* 3, 1 (Jan. 1990), 67–99. https://doi.org/10.1007/BF01806126

Mitchell Stern, Jacob Andreas, and Dan Klein. 2017. A Minimal Span-Based Neural Constituency Parser. In *Proc. ACL (Volume 1: Long Papers)*. 818–827. https://doi.org/10.18653/v1/P17-1076

Alistair Stewart, Kousha Etessami, and Mihalis Yannakakis. 2015. Upper Bounds for Newton's Method on Monotone Polynomial Systems, and P-Time Model Checking of Probabilistic One-Counter Automata. *J. ACM* 62, 4, Article 30 (2015), 33 pages. https://doi.org/10.1145/2789208

Andreas Stuhlmüller and Noah Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. In *Proc. Workshop on Statistical Relational AI (StaRAI)*. https://arxiv.org/abs/1206.3555

Ben Taskar, Dan Klein, Mike Collins, Daphne Koller, and Christopher Manning. 2004. Max-Margin Parsing. In *Proc. EMNLP*. 1–8. https://www.aclweb.org/anthology/W04-3201

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. https://arxiv.org/abs/1809.10756 arXiv:1809.10756.

K. Vijay-Shankar and Aravind K. Joshi. 1985. Some Computational Properties of Tree Adjoining Grammars. In *Proc. ACL*. 82–93. https://doi.org/10.3115/981210.981221

K. Vijayashanker. 1994. *A Study of Tree Adjoining Grammars*. Ph. D. Dissertation. Univ. of Pennsylvania.

John von Neumann. 1951. Various Techniques Used in Connection With Random Digits. *National Bureau of Standards Applied Mathematics Series* 12 (1951), 36–38.

Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From High-Level Inference Algorithms to Efficient Code. *PACM on Programming Languages* 3, ICFP, Article 98 (2019), 30 pages. https://doi.org/10.1145/3352468

David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 1, 3–43.

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. *PACM on Programming Languages* 2, ICFP, Article 87 (2018), 38 pages. https://doi.org/10.1145/3236782

David J. Weir. 1992. A Geometric Hierarchy Beyond Context-Free Languages. *Theoretical Computer Science* 104, 2 (1992), 235–261. https://doi.org/10.1016/0304-3975(92)90124-X

Received 2022-10-28; accepted 2023-02-25