TPLP **23** (4): 900–917, 2023.
doi:10.1017/S1471068423000145 First published online 6 July 2023

Automatic Differentiation in Prolog[*]

TOM SCHRIJVERS and BIRTHE VAN DEN BERG
KU Leuven, Leuven, Belgium
(*e-mails:* tom.schrijvers@kuleuven.be, birthe.vandenberg@kuleuven.be)

FABRIZIO RIGUZZI
Università degli Studi di Ferrara, Ferrara, Italy
(*e-mail:* fabrizio.riguzzi@unife.it)

submitted 25 May 2023; accepted 12 June 2023

Abstract

Automatic differentiation (AD) is a range of algorithms to compute the numeric value of a function's (partial) derivative, where the function is typically given as a computer program or abstract syntax tree. AD has become immensely popular as part of many learning algorithms, notably for neural networks. This paper uses Prolog to systematically derive gradient-based forward- and reverse-mode AD variants from a simple executable specification: evaluation of the symbolic derivative. Along the way we demonstrate that several Prolog features (DCGs, co-routines) contribute to the succinct formulation of the algorithm. We also discuss two applications in probabilistic programming that are enabled by our Prolog algorithms. The first is parameter learning for the Sum-Product Loop Language and the second consists of both parameter learning and variational inference for probabilistic logic programming.

KEYWORDS: Prolog, automatic differentiation, probabilistic programming

1 Introduction

Kowalski's slogan "algorithm = logic + control" (Kowalski 1979) has been an inspiration to express and study algorithms in Prolog. Notable examples are the *Logic Programming Pearls* of *Theory and Practice of Logic Programming* (Vandecasteele and Janssens 2003; Bruynooghe 2004; Schrijvers and Frühwirth 2006), and more recently Prolog versions of SAT/SMT solving (Howe and King 2012) and of backjumping (Robbins *et al.* 2021). Perhaps one of the best known examples and a direct inspiration of this work, is the elegant symbolic differentiation approach of Clocksin and Mellish (2003) that originally appeared in 1981.

Following these footsteps, we present a Prolog version of *automatic differentiation* (AD), a range of algorithms to compute the numeric value of a function's (partial) derivative, where the function is typically given as a computer program or abstract syntax tree. AD differs from numeric differentiation approaches, which are approximative, and from symbolic differentiation, which yields a symbolic result: AD computes the derivate value exactly, evaluating it at a specific point. While originally conceived in 1964

[*] This project is partly funded by the Flemish Fund for Scientific Research (FWO).

(Wengert 1964), AD has become immensely popular in recent years under the name of *backpropagation*, the algorithm at the heart of neural networks. It also has applications in logic programming, for instance for learning parameters of probabilistic logic programs.

This work is inspired by the approach of van den Berg *et al.* (2022), which derives forward-mode- and reverse-mode AD from symbolic differentiation using algebraic abstractions in the purely functional setting of Haskell. We recast AD in a logic programming setting: we proceed step by step from symbolic differentiation and show how to incorporate different optimizations and obtain several forward-mode and reverse-mode variants (Sections 2–8). Along the way we demonstrate that several Prolog features such as definite clause grammars (DCGs) and co-routines contribute to the succinct formulation of the algorithm. Co-routines in particular make it easy to express reverse-mode variants in a single phase where conventional presentations require two phases. While we use a minimal expression language throughout the main developments to focus on the essence, Section 9 explains how additional primitives (e.g., sin, cos) can be incorporated. We also present two case studies of AD where our Prolog implementations[1] can be used. The first (Section 10) implements parameter learning for the recently proposed Sum-Product Loop Language (SPLL). The second (Section 11) concerns parameter learning *and* variational inference for probabilistic logic programming. Finally, Section 12 discusses related work and Section 13 concludes.

2 Symbolic expressions and their evaluation

Throughout most of the paper, we use a minimal expression grammar consisting of literals `lit(N)`, with `N` a number, variables `var(X)`, with `X` a variable identifier, addition `add(`E_1`,`E_2`)`, and multiplication `mul(`E_1`,`E_2`)`.

The `eval(E,Env,N)` predicate evaluates expressions `E` to a numeric result `N` given an environment `Env` that maps the variables to their value.

```
eval(lit(N),     _Env,N) .
eval(var(X),     Env ,N) :- lookup(X,Env,N).
eval(add(E1,E2),Env ,N) :- eval(E1,Env,N1), eval(E2,Env,N2), N is N1 + N2.
eval(mul(E1,E2),Env ,N) :- eval(E1,Env,N1), eval(E2,Env,N2), N is N1 * N2.
```

To allow a $\mathcal{O}(1)$ lookup in the environment, we use natural numbers in the range $[1, n]$ for the variable identifiers and represent the environment itself as a term `env(`N_1`,...,`N_n`)` whose arguments are the values of the corresponding variables. Then, `lookup/3` can be defined as follows.

```
lookup(X,Env,N) :- arg(X,Env,N).
```

3 Symbolic differentiation

As a warm-up we start with the symbolic differentiation of our expressions, following the example of Clocksin and Mellish (2003), Section 7.11 and the textbook differentiation

[1] https://github.com/birthevdb/ad-prolog-code.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

rules. The predicate `symb(E,X,DE)` captures the symbolic differentiation relation $\frac{\partial E}{\partial X} =$ `DE` and computes it in a naive way.

$$\begin{aligned}
\frac{\partial n}{\partial x} &= 0 \\
\frac{\partial y}{\partial x} &= \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases} \\
\frac{\partial e_1 + e_2}{\partial x} &= \frac{\partial e_1}{\partial x} + \frac{\partial e_2}{\partial x} \\
\frac{\partial e_1 \times e_2}{\partial x} &= e_2 \times \frac{\partial e_1}{\partial x} + e_1 \times \frac{\partial e_2}{\partial x}
\end{aligned}$$

```
symb(lit(_N),    _X, lit(0)).
symb(var(Y),     X,  lit(DE)) :-
     ( X == Y -> DE = 1 ; DE = 0).
symb(add(E1,E2), X,  add(DE1,DE2)) :-
     symb(E1,X,DE1), symb(E2,X,DE2).
symb(mul(E1,E2), X,  add(mul(E2,DE1),mul(E1,DE2)) :-
     symb(E1,X,DE1), symb(E2,X,DE2).
```

In order to prepare for actual AD algorithms, we refactor the above code from `symb/3` to `symb/4`. The latter not only returns the symbolic derivative but also reconstructs the original expression; this structure becomes advantageous in Section 4's nonsymbolic setting. The combination of the original expression (*primal*) and the (symbolic) partial derivative (*tangent*) is also known as a (here: symbolic) *dual number* in the AD literature.

```
symb(lit(N),    _X,lit(N),   lit(0)).
symb(var(Y),    X, var(Y),   lit(DF)) :-
  ( X == Y -> DF = 1 ; DF = 0).
symb(add(E1,E2),X,add(F1,F2),add(DF1,DF2)) :-
  symb(E1,X,F1,DF1), symb(E2,X,F2,DF2).
symb(mul(E1,E2),X,mul(F1,F2),DF) :-
  symb(E1,X,F1,DF1), symb(E2,X,F2,DF2),
  DF = add(mul(F2,DF1),mul(F1,DF2)).
```

`symb(E,X,F,DF)`

$$\Leftrightarrow$$

$$\begin{cases} E = F \\ \frac{\partial E}{\partial X} = DF \end{cases}$$

4 Forward-mode automatic differentiation

Many practical applications do not require the symbolic derivative, but its numeric value at a point. Naively, this can be obtained by evaluation after symbolic derivation.

```
ad_spec(E,X,Env,N,DN) :-
  symb(E,X,F,DF),
  eval(F,Env,N),
  eval(DF,Env,DN).
```

The idea of AD is to perform the two steps, symbolic derivation and numeric evaluation, simultaneously, in order to avoid the intermediate symbolic result. Predicate `fwdad(E,X,Env,F,DF)` accomplishes this, where `F` and `DF` represent a dual number with the original expression and partial derivative, respectively, evaluated with respect to the

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

environment `Env`. This definition is essentially obtained from `ad_spec/4` by unfold/fold transformations (Debray 1988) (Appendix A).

```
fwdad(lit(N),_X,_Env,N,0).
fwdad(var(Y),X,Env,F,DF) :-
  lookup(Y,Env,F),
  ( X == Y -> DF = 1 ; DF = 0).
fwdad(add(E1,E2),X,Env,F,DF) :-
  fwdad(E1,X,Env,F1,DF1),
  fwdad(E2,X,Env,F2,DF2),
  F is F1 + F2,
  DF is DF1 + DF2.
fwdad(mul(E1,E2),X,Env,F,DF) :-
  fwdad(E1,X,Env,F1,DF1),
  fwdad(E2,X,Env,F2,DF2),
  F is F1 * F2,
  DF is F2 * DF1 + F1 * DF2.
```

Had we started from `symb/3` rather than `symb/4`, we'd have instead ended up with the following clause for `mul/2`.

```
fwdad(mul(E1,E2),X,Env,DF) :-
  fwdad(E1,X,Env,DF1),
  fwdad(E2,X,Env,DF2),
  eval(E1,Env,F1),
  eval(E2,Env,F2),
  DF is F2 * DF1 + F1 * DF2.
```

In the case of nested multiplications (like `mul(mul(E1,E2),mul(E3,E4))`), it would lead to naive re-evaluation (of `E1`–`E4`), with a quadratic runtime in the worst case.

Thanks to `symb/4`'s dual numbers, the expression is evaluated incrementally, reusing intermediate results. This way the runtime remains linear in the size of the expression.

5 Gradients in forward mode

Our previous definition of forward mode computes a single partial derivative $\frac{\partial E}{\partial X}$ at a time. However, often we are interested in computing the *gradient* ∇E, which is the vector of partial derivatives with respect to all variables. To compute the gradient, we can repeatedly invoke `fwdad/4`, but this is rather wasteful as it repeatedly computes the same primal for each partial derivative. A more efficient approach computes the whole gradient in one go. To accomplish that, we switch from the conventional dual numbers where both the primal and tangent are numeric values to a more heterogeneous structure where the tangent is the whole gradient.[2]

We represent the gradient by means of the common `assoc` library[3] (based on AVL trees) as a partial map from variables X to the partial derivative $\frac{\partial E}{\partial X}$. If a variable is not

[2] The generalized dual number structure is known as Nagata's *idealization of a module* (van den Berg *et al*. 2022; Nagata 1962).

[3] We have extended the library with `union_with_assoc/4` and `insert_with_assoc/5`: established AVL operations of respectively $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ time complexity.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

present in the map, we assume its corresponding derivative is zero. This is convenient as many intermediate results tend to be zero and thus need not be represented explicitly.

```
fwdadgrad(lit(N),_Env,N,DF) :-
  empty_assoc(DF).
fwdadgrad(var(Y),Env,F,DF) :-
  lookup(Y,Env,F),
  singleton_assoc(Y,1,DF).
fwdadgrad(add(E1,E2),Env,F,DF) :-
  fwdadgrad(E1,Env,F1,DF1),
  fwdadgrad(E2,Env,F2,DF2),
  F is F1 + F2,
  union_with_assoc(plus,DF1,DF2,DF).
fwdadgrad(mul(E1,E2),Env,F,SDF) :-
  fwdadgrad(E1,Env,F1,DF1),
  fwdadgrad(E2,Env,F2,DF2),
  F is F1 * F2,
  map_assoc(times(F2),DF1,SDF1),
  map_assoc(times(F1),DF2,SDF2),
  union_with_assoc(plus,SDF1,SDF2,SDF).

  plus(A,B,C) :- C is A + B.
  times(A,B,C) :- C is A * B.
```

If the expression tree has N nodes and contains V variables, then `fwdad/4` has an $\mathcal{O}(NV)$ time complexity: each node is visited once and the most costly operations `map_assoc/3` and `union_with_assoc/4` are both linear in V.

For a large number of variables V, the reverse-mode variant of AD appears more efficient, but also more sophisticated. In what follows we show how to obtain this reverse-mode AD by means of successively optimizing scalar multiplication and vector addition. This way, we become a declarative version of reverse-mode AD, with an $\mathcal{O}(N \log V)$ time complexity, and its imperative counterpart, at $\mathcal{O}(N + V)$.

6 Scalar multipliers

The first step toward efficient reverse-mode AD is to replace the costly scalar multiplication of the gradient vector in the `mul/1` operation with a constant-time multiplication.

Assume a scalar factor M that should be multiplied with a gradient. Instead of applying the scalar factor to the gradient vector in the `mul/1` case, it is propagated down into the subexpressions. At the `var/1` and `lit/0` leaves it becomes trivial to take the factor M into account: in the former case we create a single $M \times 1 = M$ vector entry; in the latter case we ignore it as $M \times 0 = 0$ for all variables and 0 need not be explicitly represented.

In the case of nested multiplications, we pass multiple factors $M_1, \ldots M_n$ down a path of the tree, which can be combined into a single factor $M = M_1 \times \ldots \times M_n$. In the absence of a factor, we take $M = 1$ (the neutral element of multiplication).

```
revad1(E,Env,F,DF) :-
  revad1(E,Env,1,F,DF).

revad1(lit(N),_Env,_M,N,DF) :-
  empty_assoc(DF).
revad1(var(Y),Env,M,F,DF) :-
  lookup(Y,Env,F),
  singleton_assoc(Y,M,DF).
revad1(add(E1,E2),Env,M,F,DF) :-
  revad1(E1,Env,M,F1,DF1),
  revad1(E2,Env,M,F2,DF2),
  F is F1 + F2,
  union_with_assoc(plus,DF1,DF2,DF).
revad1(mul(E1,E2),Env,M,F,DF) :-
  times(M,F2,M1),
  times(M,F1,M2),
  revad1(E1,Env,M1,F1,DF1),
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

```
  empty_assoc(DF).
revad1(var(Y),Env,M,F,DF) :-
  lookup(Y,Env,F),
  singleton_assoc(Y,M,DF).
revad1(add(E1,E2),Env,M,F,DF) :-
  revad1(E1,Env,M,F1,DF1),
  revad1(E2,Env,M,F2,DF2),
  F is F1 + F2,
  union_with_assoc(plus,DF1,DF2,DF).
revad1(E2,Env,M2,F2,DF2),
F is F1 * F2,
union_with_assoc(plus,DF1,DF2,DF).
```

There is one major snag in the above formulation. The modes in the `mul/2` case do not work out: we need `F1` (to compute `M2`), which is needed to compute `F2` and vice versa. Fortunately, the mutual dependency is only an artefact as the algorithm is set up as a single traversal of the expression tree. In fact, `F1` does not depend on `F2`, only `DF1` does. Hence, traditional algorithms work in two phases: they compute the factors in a first phase and the derivatives in a second phase. This requires additional book-keeping to store the appropriate information at different nodes in the tree during the first phase for later use in the second phase.

With Prolog's co-routine mechanisms, this problem is much easier to solve; it does not require any restructuring of the code. We only need to revise the definitions of `plus/3` and `times/3` to delay the arithmetic operations until their inputs are known. We use `freeze(N,Goal)` for this purpose, which defers the execution of `Goal` until `N` is bound.

```
plus(A,B,C) :- freeze(A,freeze(B,C is A + B)).
times(A,B,C) :- freeze(A,freeze(B,C is A * B)).
```

7 Gradient threading

After eliminating the costly scalar multiplication of the gradient vector, we now address the remaining costly operation: vector addition.

Vector additions combine the vectors that are created at the leaves of the expression in a bottom-up fashion. We change this dataflow to one that threads the gradient vector through the computation and inserts elements where they are created (in the `var/1` case). To accomplish the threading, we make use of Prolog's DCG notation (with association trees instead of lists), which allows us to implicitly pass the gradient. We initialize this vector with the neutral element of vector addition, that is, the empty vector.

```
revad2(E,Env,F,DF) :-
  empty_assoc(DF0),
  revad2(E,Env,1,F,DF0,DF).

revad2(lit(N),_Env,_M,F) -->
  { F = N }.
revad2(var(Y),Env,M,F) -->
  { lookup(Y,Env,F) },
  insert_with_assoc(plus,Y,M).
revad2(add(E1,E2),Env,M,F) -->
  revad2(E1,Env,M,F1),
  revad2(E2,Env,M,F2),
  { F is F1 + F2 }.
revad2(mul(E1,E2),Env,M,F) -->
  { times(M,F2,M1) },
  { times(M,F1,M2) },
  revad2(E1,Env,M1,F1),
  revad2(E2,Env,M2,F2),
  { F is F1 * F2 }.
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

The resulting algorithm has $\mathcal{O}(N \log V)$ time complexity, where the logarithmic factor is due to `insert_with/3`. We bring this down to $\mathcal{O}(1)$ by switching to destructive updates.

8 Reverse-mode AD, destructively

This section replaces the AVL tree representation of the gradient vector with an array to achieve an overall $\mathcal{O}(N+V)$ time complexity. The two operations we need to replace are the initialization of the gradient and the insertion of a new value. Initialization goes from $\mathcal{O}(1)$ for an empty AVL tree to $\mathcal{O}(V)$ to create a new array with an explicit 0 entry for each variable. Luckily, this operation is only performed once, at the start of the algorithm. Insertion becomes a constant time operation thanks to the `setarg/3` destructive update.

```
empty_array(N,Arr) :-
  findall(0,between(1,N,_),List),
  Arr =.. [grad|List].
```

```
insert_with_array(Pred,Var,Value,Arr) :-
  arg(Var,Arr,OldValue),
  call(Pred,OldValue,Value,NewValue),
  setarg(Var,Arr,NewValue).
```

The impact of this representation change on the algorithm's code itself is minimal.

```
revad(E,Env,F,DF) :-
  functor(Env,_,N),
  empty_array(N,DF),
  revad(E,Env,1,F,DF).

revad(lit(N),_Env,_M,F,_DF) :-
  F = N.
revad(var(Y),Env,M,F,DF) :-
  lookup(Y,Env,F),
  insert_with_array(plus,Y,M,DF).
```

```
revad(add(E1,E2),Env,M,F,DF) :-
  revad(E1,Env,M,F1,DF),
  revad(E2,Env,M,F2,DF),
  F is F1 + F2.
revad(mul(E1,E2),Env,M,F,DF) :-
  times(M,F2,M1),
  times(M,F1,M2),
  revad(E1,Env,M1,F1,DF),
  revad(E2,Env,M2,F2,DF),
  F is F1 * F2.
```

9 Extensions

So far, we have only considered a minimal grammar for expressions. Many applications of AD involve additional mathematical functions (e.g., trigonometric, exponential, logarithmic), which can be easily incorporated into our approach. For example, we can extend our expression language with negation `neg(E)`, sine `sin(E)` and cosine `cos(E)`, exponentials `exp(E)`, ... The chain rule explains how to support extensions:

$$\frac{\partial f(e)}{\partial x} = \frac{\partial f(e)}{\partial e} \times \frac{\partial e}{\partial x}$$

where $\frac{\partial f(e)}{\partial e}$ is the derivative of f with respect to its argument and evaluated at e. The following clauses incorporate these additional functions in the last version we have presented (left) based on the symbolic derivatives of the functions (right).

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

```
revad(neg(E1),Env,M,F,DF) :-
  freeze(M,freeze(F1,M1 is M * -F1)),
  revad(E1,Env,M1,F1,DF),
  F is -F1.
revad(sin(E1),Env,M,F,DF) :-
  freeze(M,freeze(F1,M1 is M * cos(F1))),
  revad(E1,Env,M1,F1,DF),
  F is sin(F1).
revad(cos(E1),Env,M,F,DF) :-
  freeze(M,freeze(F1,M1 is M * -sin(F1))),
  revad(E1,Env,M1,F1,DF),
  F is cos(F1).
revad(exp(E1),Env,M,F,DF) :-
  freeze(M,freeze(F1,M1 is M * exp(F1))),
  revad(E1,Env,M1,F1,DF),
  F is exp(F1).
```

$$\frac{\partial -x}{\partial x} = -x$$

$$\frac{\partial \sin x}{\partial x} = \cos x$$

$$\frac{\partial \cos x}{\partial x} = -\sin x$$

$$\frac{\partial \exp x}{\partial x} = \exp x$$

The chain rule generalizes to multiargument functions as follows:

$$\frac{\partial f(e_1,\ldots,e_n)}{\partial x} = \sum_{i=1}^{n} \frac{\partial f(e_1,\ldots,e_n)}{\partial e_i} \times \frac{\partial e_i}{\partial x}$$

where f_i' is the partial derivative of f with respect to its ith argument.

For example, we use the notation `pow(E1,E2)` to denote $\mathtt{E}_1$ raised to the power $\mathtt{E}_2$.

```
revad(pow(E1,E2),Env,M,F,DF) :-
  freeze(M,freeze(F1,freeze(F2,
      M1 is M * F2 * F1**(F2 - 1)))),
  freeze(M,freeze(F1,freeze(F2,
      M2 is M * (F1**F2) * log(F1)))),
  revad(E1,Env,M1,F1,DF),
  revad(E2,Env,M2,F2,DF),
  F is F1**F2.
```

$$\frac{\partial x^b}{\partial x} = bx^{b-1}$$

$$\frac{\partial a^x}{\partial x} = a^x \ln a$$

10 Case study: Sum-product loop programming

Pfanschilling *et al.* (2022) have recently proposed the Sum-Product Loop Language (SPLL), but (as far as we are aware) no implementation has been made publicly available. As a case study, we have implemented this probabilistic language in Prolog, and notably its parameter estimation functionality, with our AD algorithms.

Parameter Estimation An SPLL program defines a probability distribution over its possible outcomes. For instance, `main = Uniform >= Theta[1]` transforms the `Uniform` distribution on the interval $[0,1]$ into a Bernoulli distribution: $p(\texttt{false}) = \theta_1$ and $p(\texttt{true}) = 1 - \theta_1$. Formally, SPLL comes with both a generative and a probabilistic semantics. The former samples the program (i.e., generates a result x) in accordance with its probability distribution, and the latter computes the probability $p(x)$ of a given sample x. The probabilistic semantics can be used to estimate the parameters θ of a program in terms of a set of samples X. Specifically, given a set of samples X and a consistent SPLL program, find the parameterization θ of the program that minimizes the negative log-likelihood $\mathscr{L}$ of all samples: $\mathscr{L} = \sum_{x \in X} -\log p(x|\theta)$.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

This problem can be tackled with gradient-based optimization, where the gradient $\frac{\partial \mathscr{L}}{\partial \theta}$ is computed with automatic differentiation. Indeed, given a learning rate λ and an initial guess for the parameters θ, we can iteratively improve the parameters with $\theta := \theta - \lambda \frac{\partial L}{\partial \theta}$. We have implemented a symbolic version of the probabilistic semantics, which yields for our small example program

$$p(\texttt{true}) = \int_{\theta_1}^{\infty} \varphi_U(x)\,dx \qquad p(\texttt{false}) = 1 - \int_{\theta_1}^{\infty} \varphi_U(x)\,dx$$

Given an initial guess $\theta_1 = 0.5$, a learning rate of 0.02, and a set X of 3 `false` samples and 7 `true` samples, the parameter converges to 0.3000000000000001 in 13 steps.

Benchmarking our AD Versions To compare the runtime of the different AD versions, we have used the following SPLL program, which features six parameters $\theta_1,\ldots,\theta_6$.

```
main = if Uniform >= Theta[1]
       then if Uniform >= Theta[2]
            then if Uniform >= Theta[3] then null    else [true]
            else if Uniform >= Theta[4] then [false] else [true,true]
       else if Uniform >= Theta[5]
            then if Uniform >= Theta[6] then [true,false] else [false,true]
            else [false,false]
```

We have used the same gradient-based optimization of the negative log-likelihood to simultaneously learn the six parameters. Given an initial guess of 0.5 for θ_1 and 0.25 for the others, a learning rate of 0.02 and sets of 3 examples for each outcome, the parameters converge to the following in 100 iterations:

Parameter	Learned value	Optimal value
θ_1	0.4285714285714287	3/7
θ_2	0.5	1/2
θ_3	0.49999999999999994	1/2
θ_4	0.49999999999999994	1/2
θ_5	0.3333333333333333	1/3
θ_6	0.49999999999999994	1/2

We have also measured the runtime of this experiment (repeated 100 times) using our four different AD algorithms. These measurements were performed on a Macbook Pro with M1 chip, 16GB RAM, Ventura 13.1, with SWI-Prolog version 8.4.1 for arm64-darwin.

AD version	`fwdadgrad`	`revad1`	`revad1`	`revad`
Runtime (s)	9.346	9.098	5.768	4.909

These results demonstrate the improvements achieved by the successive algorithms.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

11 Case study: Probabilistic logic programming

Automatic differentiation has various applications in probabilistic logic programming (Riguzzi 2018). For instance, it can be used to perform both parameter learning and variational inference on *hybrid* programs, which are programs that include both continuous and discrete random variables.

11.1 Learning hybrid programs

Islam *et al.* (2012) proposed Extended PRISM, a version of PRISM (Sato and Kameya 1997) that allows continuous random variables with a Gaussian or gamma distribution.

PRISM introduces random atoms via the predicate `msw/2` where the first argument is a *random switch name*, a term representing a discrete random variable, and the second argument represents a value for that variable. An example atom is `msw(m,a)`.

The set of possible values for a switch is defined by a fact for the predicate `values/2` where the first argument is the name of the switch and the second argument is a list of terms representing its possible values. For example, `values(m,[a,b])`.

The probability distribution over the values of the random variable associated with a switch name is defined by a directive for the predicate `set_sw/2` where the first argument is the name of the switch and the second argument is a list of probabilities. For example:

```
:- set_sw(m, [0.3, 0.7]).
```

The semantics of the language can be defined by specifying a way to sample values for the variables of a query atom from the program: the query atom is answered by resolution for deterministic programs with the only difference that, each time a `msw(name,value)` atom is encountered, a value is sampled from the distribution for `name` and unified with `value`. Extended PRISM adds to PRISM continuous random variables with a Gaussian or gamma distribution. In this case, the `values/2` predicate has `real` instead of the list of values and the directive `set_sw/2` specifies the probability density such as `norm(0.5,0.1)` for a Gaussian distribution with mean 0.5 and variance 0.1.

Let us illustrate the language with an example. Suppose a factory has two machines a and b. Each machine produces a widget with a continuous feature. A widget is produced by machine a with probability 0.3 and by machine b with probability 0.7. If the widget is produced by machine a, the continuous feature is distributed as a Gaussian with mean 2.0 and variance 1.0. If the widget is produced by machine b, the continuous feature is distributed as a Gaussian with mean 3.0 and variance 1.0. A third machine then adds a random quantity to the feature. The quantity is distributed as a Gaussian with mean 0.5 and variance 0.1. This is encoded by the program:

```
widget(X) :- msw(m, M), msw(st(M), Z), msw(pt, Y), X = Y + Z.
values(m, [a,b]).
values(st(_), real).
values(pt, real).
:- set_sw(m, [0.3, 0.7]).
:- set_sw(st(a), norm(2.0, 1.0)).
:- set_sw(st(b), norm(3.0, 1.0)).
:- set_sw(pt, norm(0.5, 0.1)).
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

Islam *et al.* (2012) presented an inference algorithm that solves the DISTR task (Riguzzi 2018): computing the probability distribution or density of the nonground arguments of a conjunction of literals, for example, computing the probability density of `X` in goal `widget(X)` of the example above. The algorithm collects symbolic derivations for the query and then builds a representation of the probability density associated with the variables of each goal, bottom-up, starting from the leaves.

For the widget example, the probability density of `X` in goal `widget(X)` is (Islam *et al.* 2012):

$$p(x) = 0.3 \cdot \varphi_N(x; 2.5, 1.1) + 0.7 \cdot \varphi_N(x; 3.5, 1.1)$$

with $\varphi_N(x; \mu, \sigma^2)$ the density of a Gaussian distribution with mean μ and variance σ^2:

$$\varphi_N(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

For extended PRISM, the problem of parameter learning consists of being given a set of ground atoms X for a query predicate and a program with some unknown parameters and finding the parameters that maximize the log-likelihood of the atoms in X.

This problem can be solved by using inference to find the density of the arguments of the query predicate, using automatic differentiation for finding the derivatives of the density with respect to the various parameters and then using gradient ascent for optimizing the log-likelihood, similarly to SPLL. For the widget example, the parameters could be learned with this approach from a set of ground atoms for the predicate `widget/1`.

We performed an experiment that learns back the parameters for the `pt` switch from data. To do so, we generated 50,000 samples of the query `widget(X)` from the program above, resulting in 50,000 values for `X`. Then we replaced

```
:- set_sw(pt, norm(0.5, 0.1)).
```

with

```
:- set_sw(pt, norm(mu, sigma2)).
```

and we generated with inference the probability density for `X` in `widget(X)`, obtaining

$$p(x) = 0.3 \cdot \varphi_N(x; 2 + \mu, 1 + \sigma^2) + 0.7 \cdot \varphi_N(x; 3 + \mu, 1 + \sigma^2)$$

where μ stands for `mu` and σ^2 for `sigma2`. We applied gradient-based optimization as for the SPLL case. To make sure σ^2 remains positive, we reparametrized it using a weight W between $-\infty$ and $+\infty$ and expressing σ^2 as e^W. We replace σ^2 in the formula with e^W and we computed the derivatives of $p(x)$ with respect to μ and W using `revad/5`. Once the procedure terminates and returns the values for μ and W, we obtain σ^2 as e^W.

With a learning rate of 0.00005, the procedure converged in 29 iterations giving the values 0.502130 for μ and 0.002331 for σ^2 that are quite close to the true values. The procedure took 153.27 s on a 2.8 GHz Intel Core i7 using SWI-Prolog.

11.2 Variational inference

In variational inference, we want to approximate a difficult-to-compute conditional probability density $p(x|y)$ with a simpler distribution $p_\theta(x)$. In the widget example, we may want to compute $p(Y|X = 0.2)$ where X and Y are the variables appearing in the program

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

as `X` and `Y`. The simpler distribution $p_\theta(x)$ is parameterized by a set of parameters θ, and we want to optimize them in order to make $p_\theta(x)$ as similar as possible to $p(x|y)$. This is typically done by minimizing the Kullback–Leibler divergence:

$$\begin{aligned} KL(p_\theta(x), p(x|y)) &= \int_x p_\theta(x) \log\left(\frac{p_\theta(x)}{p(x|y)}\right) \\ &= \int_x p_\theta(x) \log\left(\frac{p_\theta(x)}{p(y|x)p(x)}\right) + \log p(y) \\ &= -L(\theta) + \log p(y) \end{aligned} \tag{1}$$

where

$$L(\theta) \triangleq \int_x p_\theta(x) \log\left(\frac{p(y|x)p(x)}{p_\theta(x)}\right) \tag{2}$$

Since $\log p(y)$ does not depend on θ, the divergence is minimized by maximizing $L(\theta)$. The maximization can be performed using gradient ascent if we can compute the derivatives of $L(\theta)$ in terms of θ. Wingate and Weber (2013) proposed an approach to apply variational inference to probabilistic programming in general. The gradient can be estimated according to the following formula (Wingate and Weber 2013)

$$-\nabla_\theta L(\theta) = \approx \frac{1}{N} \sum_{x^j} \nabla_\theta \log p_\theta(x^j) \left(\log\left(\frac{p_\theta(x^j)}{p(y|x^j)p(x^j)}\right) + K\right) \tag{3}$$

which is a Monte Carlo approximation of an integral where $x^j \sim p_\theta(x)$, $j = 1 \ldots N$ and K is an arbitrary constant.

Suppose $p(x)$ is given by a probabilistic (logic) program. Our aim is to find a *target program* encoding the conditional distribution $p(x|y)$. We do so by considering another program encoding $p_\theta(x)$ that we call the *variational program*: we optimize the parameters θ to make $p_\theta(x)$ as similar as possible to $p(x|y)$. We use inference to compute $p_\theta(x)$ for the variational program and we use automatic differentiation to compute $\nabla_\theta \log p_\theta(x)$. From equation (3), we see that what is left to do is to compute $p(y|x^j)$, which is easy to execute with forward inference: in the widget example, it means computing $p(X|Y)$ where forward inference can be applied after setting Y to a fixed value.

12 Related work

There has been a lot of work on automatic differentiation in the general programming languages research community in recent years (Krawiec *et al.* 2022; Wang *et al.* 2019; Szirmay-Kalos 2021; Smeding and Vákár 2022; Pearlmutter and Siskind 2008). Much of this work has been focused on extending the expressivity of AD (e.g., to higher order functions) and on ways of showing the correctness of different AD flavors. Several works also focus on the compositionality of models (Nguyen *et al.* 2022; Dash *et al.* 2023).

This paper is most closely related to that of van den Berg *et al.* (2022), which provides a (Haskell-based) account of AD in terms of various forms of generalized dual numbers. A key difference with these existing works is that they use "Wengert lists," "tapes," or function abstractions to defer computations of tangents that depend on primals, while our Prolog approach can simply use coroutines.

In contrast, as far as we know, there is little work on AD in the logic programming community. We have found only two (unpublished) manuscripts. The first is a paper

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

by Abdallah (2017) on implementing AD with Constraint Handling Rules (CHR), which involves a different programming style (e.g., using constraints to represent the expression AST and using delay mechanisms much more extensively) and deviates more from the traditional algorithms. Moreover, it presents two algorithms as given, rather than deriving them systematically like we do. The second is an unfinished blogpost by Gabel (2020), which provides a Prolog implementation of forward-mode AD that takes a compilation-based approach and produces a sequence of assignments.

Algebraic ProbLog (Kimmig *et al.* 2011) uses semirings to support automatic differentiation in the domain of probabilistic logic programming, which is used by DeepProbLog (Manhaeve *et al.* 2018) for gradient-descent optimization in their backpropagation.

Many machine learning libraries come with highly optimized automatic differentiation implementations (e.g., with tailored memory management techniques and GPU leverage), such as Tensorflow's (Abadi *et al.* 2015) and PyTorch's (Paszke *et al.* 2019) Autograd, which targets Python code. Google's JAX (Bradbury *et al.* 2018) extends Autograd with just-in-time compilation, inspired by Tensorflow's XLA (accelerated linear algebra). The probabilistic programming language Stan Carpenter *et al.* (2015; 2017) leverages C++ to implement automatic differentiation. All of these easily outperform our Prolog implementations, whose aim is not to provide competitive performance. Instead, we aim to provide an account of AD for the logic programming community that is instructive, accessible, and lowers the threshold for experimentation.

13 Conclusion

We have shown that forward-mode and reverse-mode automatic differentiation in Prolog can be systematically derived from their specification in terms of symbolic differentiation and evaluation. Their definitions are elegant and concise and achieve the textbook theoretical time complexities.

In future work, we plan to explore Prolog's meta-programming facilities (e.g., term expansion) to implement partial evaluation of `revad/5` calls on known expressions. We also wish to develop further applications on top of our AD approach, such as Prolog-based neural networks and integration with existing probabilistic logic programming languages.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abdallah, S. 2017. Automatic differentiation using constraint handling rules in Prolog. URL: https://arxiv.org/abs/1706.00231.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S. and Zhang, Q. 2018. JAX: Composable transformations of Python+NumPy programs. URL: https://github.com/google/jax.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

Bruynooghe, M. 2004. Enhancing a search algorithm to perform intelligent backtracking. *Theory and Practice of Logic Programming 4,* 3, 371–380.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P. and Riddell, A. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software 76,* 1, 1–32.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P. and Betancourt, M. 2015. The stan math library: Reverse-mode automatic differentiation in c++. arXiv preprint arXiv:1509.07164.

Clocksin, W. F. and Mellish, C. S. 2003. *Programming in Prolog*, 5th ed. Springer, Berlin.

Dash, S., Kaddar, Y., Paquet, H. and Staton, S. 2023. Affine monads and lazy structures for bayesian programming. *Proceedings of the ACM on Programming Languages 7,* POPL, 1338–1368.

Debray, S. K. 1988. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI'88. New York, NY, USA, 297–307.

Gabel, P. L. 2020. Automatic differentiation. URL: http://petergabel.info/blog/AutoDiff/#Backward-Mode-Automatic-Differentiation-1.

Howe, J. M. and King, A. 2012. A pearl on SAT and SMT solving in prolog. *Theoretical Computer Science 435*, 43–55.

Islam, M. A., Ramakrishnan, C. and Ramakrishnan, I. 2012. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming 12*, 505–523.

Kimmig, A., Van den Broeck, G. and De Raedt, L. 2011. An algebraic prolog for reasoning about possible worlds. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Kowalski, R. A. 1979. Algorithm = logic + control. *Communications of the ACM 22,* 7, 424–436.

Krawiec, F., Jones, S. P., Krishnaswami, N., Ellis, T., Eisenberg, R. A. and Fitzgibbon, A. W. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages 6,* POPL, 1–30.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T. and De Raedt, L. 2018. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31*.

Nagata, M. 1962. *Local Rings*. Wiley Interscience.

Nguyen, M., Perera, R., Wang, M. and Wu, N. 2022. Modular probabilistic models via algebraic effects. *Proceedings of the ACM on Programming Languages 6,* ICFP, 381–410.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035.

Pearlmutter, B. A. and Siskind, J. M. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems 30,* 2, 7:1–7:36.

Pfanschilling, V., Shindo, H., Dhami, D. S. and Kersting, K. 2022. Sum-product loop programming: From probabilistic circuits to loop programming. In *Proceedings of KR*.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning*. River Publishers, Gistrup, Denmark.

Robbins, E., King, A. and Howe, J. M. 2021. Backjumping is exception handling. *Theory and Practice of Logic Programming 21,* 2, 125–144.

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

Sato, T. and Kameya, Y. 1997. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, Vol. 97, 1330–1339.

Schrijvers, T. and Frühwirth, T. W. 2006. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming 6,* 1–2, 213–224.

Smeding, T. and Vákár, M. 2022. Efficient dual-numbers reverse ad via well-known program transformations. URL: https://arxiv.org/abs/2207.03418.

Szirmay-Kalos, L. 2021. Higher order automatic differentiation with dual numbers. *Periodica Polytechnica Electrical Engineering and Computer Science 65,* 1, 1–10.

van den Berg, B., Schrijvers, T., McKinna, J. and Vandenbroucke, A. 2022. Forward- or reverse-mode automatic differentiation: What's the difference? CoRR abs/2212.11088.

Vandecasteele, H. and Janssens, G. 2003. An open ended tree. *Theory and Practice of Logic Programming 3,* 3, 377–385.

Wang, F., Zheng, D., Decker, J. M., Wu, X., Essertel, G. M. and Rompf, T. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages 3,* ICFP, 96:1–96:31.

Wengert, R. E. 1964. A simple automatic derivative evaluation program. *Communications of the ACM 7,* 8, 463–464.

Wingate, D. and Weber, T. 2013. Automated variational inference in probabilistic programming. URL: https://doi.org/10.48550/arXiv.1301.1299.

Appendix A Derivation of `fwdad/4` from `ad_spec/4`

```
ad_spec(E,X,Env,N,DN) :-
  symb(E,X,F,DF),
  eval(F,Env,N),
  eval(DF,Env,DN).

<=> (unfold symb)

  * Case lit/1

      ad_spec(lit(M),X,Env,N,DN) :-
        lit(M),X,lit(N),lit(0)
        eval(lit(M),Env,N),
        eval(lit(0),Env,DN).

    <=> (unfold of eval)

      ad_spec(lit(M),X,Env,M,0).

  * Case var/1

      ad_spec(var(Y),X,Env,N,DN) :-
        ( X == Y -> DF = 1 ; DF = 0),
        eval(var(Y),Env,N),
        eval(lit(DF),Env,DN).
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

```
  <=> (commutativity of ,/2 and distributivity of ,/2 wrt (->;))

    ad_spec(var(Y),X,Env,N,DN) :-
      eval(var(Y),Env,N),
      ( X == Y -> eval(lit(1),Env,DN); eval(lit(0),Env,DN)).

  <=> (unfold eval)

    ad_spec(var(Y),X,Env,N,DN) :-
      lookup(Y,Env,N),
      ( X == Y -> DN = 1; DN = 0).

* Case add/2

    ad_spec(add(E1,E2),X,Env,N,DN) :-
      symb(E1,X,F1,DF1),
      symb(E2,X,F2,DF2),
      eval(add(F1,F2),Env,N),
      eval(add(DF1,DF2),Env,DN).

  <=> (unfold eval)

    ad_spec(add(E1,E2),X,Env,N,DN) :-
      symb(E1,X,F1,DF1),
      symb(E2,X,F2,DF2),
      eval(F1,Env,N1),
      eval(F2,Env,N2),
      N is N1 + N2,
      eval(DF1,Env,DN1),
      eval(DF2,Env,DN2),
      DN is DN1 + DN2.

  <=> (commutativity of ,/2)

    ad_spec(add(E1,E2),X,Env,N,DN) :-
      symb(E1,X,F1,DF1),
      eval(F1,Env,N1),
      eval(DF1,Env,DN1),
      symb(E2,X,F2,DF2),
      eval(DF2,Env,DN2),
      eval(F2,Env,N2),
      N is N1 + N2,
      DN is DN1 + DN2.
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

```
    <=> (fold of ad_spec)

      ad_spec(add(E1,E2),X,Env,N,DN) :-
        ad_spec(E1,X,N1,DN1),
        symb(E2,X,N2,DN2),
        N is N1 + N2,
        DN is DN1 + DN2.

  * Case mul/2

      ad_spec(mul(E1,E2),X,Env,N,DN) :-
        symb(E1,X,F1,DF1),
        symb(E2,X,F2,DF2),
        eval(mul(F1,F2),Env,N),
        eval(add(mul(F1,DF2),mul(F2,DF1)),Env,DN).

    <=> (unfold eval)

      ad_spec(mul(E1,E2),X,Env,N,DN) :-
        symb(E1,X,F1,DF1),
        symb(E2,X,F2,DF2),
        eval(F1,Env,N1),
        eval(F2,Env,N2),
        N is N1 * N2,
        eval(F1,Env,M1),
        eval(DF2,Env,DN2),
        eval(F2,Env,M2),
        eval(DF1,Env,DN1),
        DN is M1 * DN2 + M2 * DN1.

    <=> (functional dependency of eval and idempotence)

      ad_spec(mul(E1,E2),X,Env,N,DN) :-
        symb(E1,X,F1,DF1),
        symb(E2,X,F2,DF2),
        eval(F1,Env,N1),
        eval(F2,Env,N2),
        N is N1 * N2,
        eval(DF2,Env,DN2),
        eval(DF1,Env,DN1),
        DN is N1 * DN2 + N2 * DN1.

    <=> (commutativity of ,/2)

      ad_spec(mul(E1,E2),X,Env,N,DN) :-
        symb(E1,X,F1,DF1),
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press

```
    eval(F1,Env,N1),
    eval(DF1,Env,DN1),
    symb(E2,X,F2,DF2).
    eval(F2,Env,N2),
    eval(DF2,Env,DN2),
    N is N1 * N2,
    DN is N1 * DN2 + N2 * DN1.

<=> (fold of ad_spec)

  ad_spec(mul(E1,E2),X,Env,N,DN) :-
    ad_spec(E1,X,N1,DN1),
    ad_spec(E2,X,N2,DN2),
    N is N1 * N2,
    DN is N1 * DN2 + N2 * DN1.
```

https://doi.org/10.1017/S1471068423000145 Published online by Cambridge University Press