A Succinct Multivariate Lazy Multivariate Tower AD for Weil Algebra Computation

Hiromi Ishii
DeepFlow, Inc.

Abstract

We propose a functional implementation of *Multivariate Tower Automatic Differentiation*. Our implementation is intended to be used in implementing C^∞-structure computation of an arbitrary Weil algebra, which we discussed in [5].

1 Introduction

Automatic Differentiation (AD) is known as a powerful technique to compute differential coefficients of a given (piecewise) smooth function efficiently and accurately. In the upcoming paper [5], the author proposed to use C^∞-rings and Weil algebras to provide a modular and exprresive framework for forward-mode automatic difrerentiation. There, compute the C^∞-structure of an arbitrary Weil algebra as a quotient of that of the formal power series ring $\mathbb{R}[\![\boldsymbol{X}]\!]$. The C^∞-structure of $\mathbb{R}[\![\boldsymbol{X}]\!]$ was then computed via *multivariate tower AD*. It can be implemented in various ways, such as Lazy Multivariate Tower AD [8], or nested Sparse Tower AD [7, module `Numeric.AD.Rank1.Sparse`].

Theoretically, such existing methods can be used to compute the C^∞-structure of $\mathbb{R}[\![\boldsymbol{X}]\!]$. However, these methods are somewhat complex and not optimised for our purpose. In this paper, we will propose another implementation of Lazy Multivariate Tower-Mode AD using tree representation and exploiting smoothness to save memory consumption. Our method can be seen as aforementioned existing implementations [8, 7].

2 Implementation

As an implementation language, we adopt a Haskell [3], a purely functional lazy programming language. It has several virtues useful for our purpose:

1. It supports higher-order functions natively.
2. It is a lazy language, enabling us to treat *infinite* structures.
3. The type-class mechanism in Haskell allows us to use function overloading in handy.
4. Its type system allows us to implement complex inductive types safely.

For a general discussion on the advantages of using Haskell in computer algebra, we refer readers to Ishii [4].

We follow the standard pattern in implementing ADs in Haskell: we use the function overloading to implement operations on types corresponding ADs. This strategy is taken, for example, in Karczmarczuk [6], Elliott [1] and implemented in `ad` package [7]. In Haskell, the `Floating` type-class gives an abstraction over floating-point numbers that admit elementary functions, as excerpted in Listing 1.

For example, a simple forward-mode AD implemented as a dual number can be defined as in Listing 2. The data-type `AD` encapsulates a value of some univariate function and its first-order differential

```
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sin :: a -> a
  asin :: a -> a
  ...
```

Listing 1: The `Floating` class

```
  data AD a = AD a a deriving (Show, Eq, Ord)
  instance Floating a => Floating (AD a) where
    exp (AD f f') = AD (exp f) (f' * exp f)
    sin (AD f f') = AD (sin f) (f' * cos f)
    log (AD f f') = AD (log f) (f' / f)
    ...
```

Listing 2: The definition of `AD`

coefficient and calculates the result using the Chain Rule, using `Floating`-operations on the coefficient `a`.

So our goal is to implement `STower n a` data-type conveying information of *all* the higher-order derivatives of an n-variate smooth function on `a`, which has `Floating (STower n a)` instances for all `Floating a` and n. In addition, we demand the implementation to be *succinct* and *efficient*, in a sense that it avoids equivalent calculation as much as possible. For example, if we want to calculate $f_{xy^2}(a, b)$ and $f_{x^2y}(a, b)$ for some smooth $f : \mathbb{R}^2 \to \mathbb{R}$, it should calculate the derivatives up to f_{xy} at most once and share their results in computing both f_{xy^2} and f_{x^2y}; in other words, results up to f_{xy} must be *memoised*.

To that end, we employ an infinite tree representation. The main idea is to use an n-ary infinite tree to express a (piecewise) smooth functions: the root node corresponds to the value $f(\boldsymbol{a})$, and its child in the n^{th} branch corresponds to $\partial x_n f(\boldsymbol{a})$. The intuition in the trivariate case is depected in Figure 1. This can be viewed as a nested trie (or prefix-tree) for memoising functions with n-many natural number arguments. Actually, this representation is isomorphic to the one we can obtain when `Sparse` tower from `ad` package [7] to the fixed-length vectors. However, as we assume f to be (piecewise) smooth, there are space for optimisation. That is, in the above representation, f_{xy} and f_{yx} must almost always coincide except on non-smooth points. In other words, we can assume partial differential operators to be almost always commutative on inputs. In many applications, the value on the non-smooth points is negligible and hence we must use more *succinct* representation making use of the commutativity.

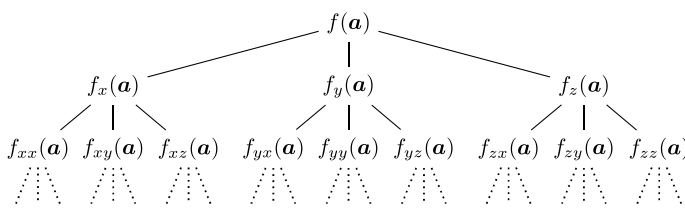


Figure 1: Trivariate case, first trial

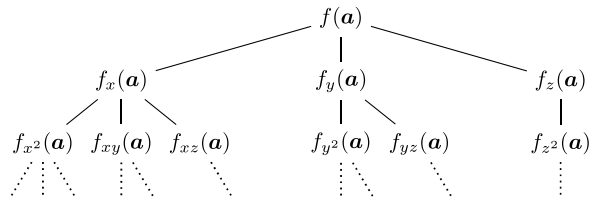


Figure 2: Trivariate case, succinct version

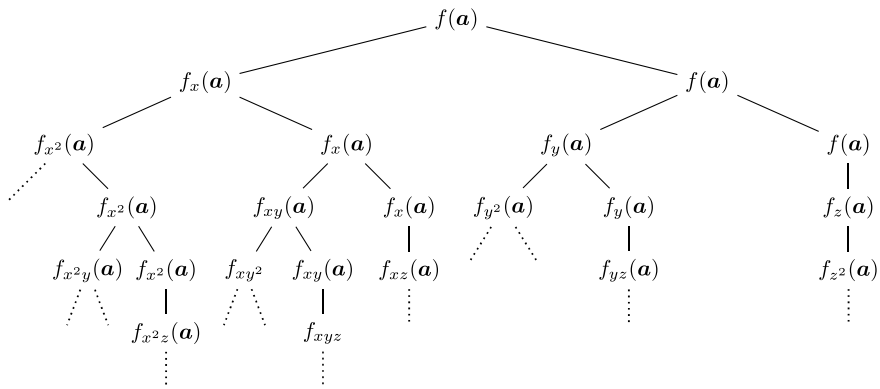


Figure 3: Trivariate case, tweaked succinct version

The idea is simple: if once one goes down i^{th} path, we can only choose j^{th} branches for $j \geq i$. This trick is illustrated in Figure 2 for trivariate case. This can be seen as a special kind of an infinite trie (or prefix-tree) of alphabets ∂_{x_i} with available letter eventually decreasing.

We further tweak this representation to make data-type definition and algorithm simple (Figure 3). This has two advantages:

1. We can directly express the tree structure purely as an algebraic data-type (ADT); we need to use fixed-length vectors here otherwise.

2. Derivatives can be computed simply by induction on the number of variables.

To see these advantages, we now turn to the Haskell implementation (Listing 3). `STower n a` is the type corresponding to the n-variate formal power series ring. Instance declarations implements functions and their derivatives on `STower n a`. It might seems circular at the first glance, but these definitions works several reasons. For example, the definitions of sin and cos works because:

1. The calls of sin and cos in the first arguments (`sin f` and `cos f`) are those on the value `f` in the coefficient field `a`, not `STower n a`, whose implementation is already given.

2. The second calls of sin and cos (`df * cos f` and `-df * sin x`) is those on the `STower n a` itself and results in an infinite loop. However, we are constructing an infinite trees that carries *all* the partial derivative coefficients, and those coefficients are stored as the first argument of `SS`, which has definite values by the discussion in (1).

```haskell
data STower n a where
  ZS :: !a -> STower 0 a
  SS :: !a -> STower (n + 1) a -> STower n a -> STower (n + 1) a

instance Num a => Num (STower n a) where
  ZS a + ZS b = SZ (a + b)
  SS f df dus + SS g dg dvs = SS (f + g) (df + dg) (dus + dvs)

  ZS a * ZS b = ZS (a * b)
  SS f df dus * SS g dg dvs = (f * g) (f * dg + df * g) (dus * dvs)
  ...

instance Fractional a => Fractional (STower n a) where
  ZS a / ZS b = ZS (a / b)
  SS f df dus / SS g dg dvs = SS (f / g) (df / g - f * dg / g^2) (dus / dvs)

instance Floating a => Floating (STower n a) where
  log (ZS a) = ZS (log a)
  log (SS f df dus) = SS (log f) (df / f) (log dus)
  sin (ZS a) = ZS (sin a)
  sin (SS f df dus) = SS (sin f) (df * cos f) (sin dus)
  cos (ZS a) = ZS (cos a)
  cos (SS f df dus) = SS (cos f) (- df * sin f) (cos dus)
  exp (ZS a) = ZS (exp a)
  exp (SS f df dus) = SS (exp f) (df * exp f) (exp dus)
  ...
```

Listing 3: Definitions of operations of `STower`

3. The direct calls in final arguments (`sin dus` and `cos dus`) seems looping, but they are indeed on power series with strictly less variable,that is, on `STower (n - 1) a`, instead of `STower n a`. As these arity strictly decreases, this stops after finite steps.

Note that (2) works also because both `sin` and `cos` are members of the `Floating` class. In other words, member functions of the `Floating` class is closed under derivatives. More generally, if the class `c a` provides a family of numerical functions on `c` which is closed under derivatives (together with functions from their superclasses), we can likewise derive the instance `c (STower n a)` as above. This idea is embodied in `liftSTower` function in Listing 4, and the usage is illustrated by the alternative definitions of instances that follows.

One might note that (3) and the last argument in the `SS`-clause in `liftSmooth` is really simple recursion. We adopted the tweaked representation as depicted in Figure 3, instead of Figure 2, for this simplicity. If we make branching n-ary as in Figure 2, the implementation gets more complicated as the number of variable decreases. One might worry that the same coefficient gets duplicated among multiple branches, but if we choose a language with a sharing (like Haskell), these value are shared across the siblings[1)].

[1)] We could achieved this also in more low-level langauges, such as C/C++, by representing each node as a pointer instead of a direct value.

```haskell
liftSTower
  :: forall c n a. (KnownNat n, c a, forall x k. c x => c (STower k x) )
  => (forall x. c x => x -> x)
      -- ^ Function
  -> (forall x. c x => x -> x)
      -- ^ its first-order derivative
  -> STower n a
  -> STower n a
liftSTower f df (ZS a) = ZS (f a)
liftSTower f df x@(SS a da dus) = SS (f a) (da * df x) (f df dus)

instance Floating a => Floating (STower n a) where
  log = liftSTower @Floating log recip
  sin = liftSTower @Floating sin cos
  cos = liftSTower @Floating cos (negate . sin)
  exp = liftSTower @Floating exp exp
```

Listing 4: Helper function for implementing derivatives

3 Benchmarks

We report several benchmarks on the proposed method with the existing implementation in the `ad` [7] package.

The code used in the benchmarks is implemented as a Haskell library and hosted on GitHub[2)]. All the benchmark code was compiled with GHC 8.10.4 with the flag `-threaded -O2`. We ran the benchmark suites on a virtual Linux environment available on GitHub Actions (Standard_DS2_v2 Azure instance) with two Intel Xeon Platinum 8171M virtual CPUs (2.60GHz) and 7 GiB of RAM. The Gauge framework [2] was used to report the run-time speed.

3.1 Tower Automatic Differentiation

In this subsection, we compare the run-time speed and memory consumption of the existing multivariate tower AD and the proposed method. In univariate case, we compare our proposed method with two existing implementations: `Sparse` and `diffs` both from `ad` package. The former is the generic implementation of multivariate tower AD, and the former is speciailised implementation for the univariate case.

Figures 4 and 5 show the run-time speed and heap allocation of the existing implementations (`Sparse` and `diffs` only for univariate case), and the proposed method. In univaricate case, the existing implementation for univariate tower AD provided, i.e. `diffs` from `ad` package, peforms the best both in terms of run-time speed and memory allocation. Notably, our implementation outperforms `Sparse` both in time and space. In particular, although `Sparse` performs linearly both in time and space when applied to the identity, our method performs eventually constantly. This is because our actual implementation employs some heuristics to detect a function whose higher derivatives are eventually zero.

Compared to `Sparse`, our proposed method performs always better both in univariate and multivariate cases. In particular, `Sparse` presents a steep quadratic growth both in time and space, our proposed method peroforms almost linearly.

In summary, our method performs really well both in terms of time and space in the multivariate case, although we need more optimisation in univariate cases.

[2)] https://github.com/konn/smooth/tree/d3386a15c97f23d4071b09f97b7bc87f2c4b1da4

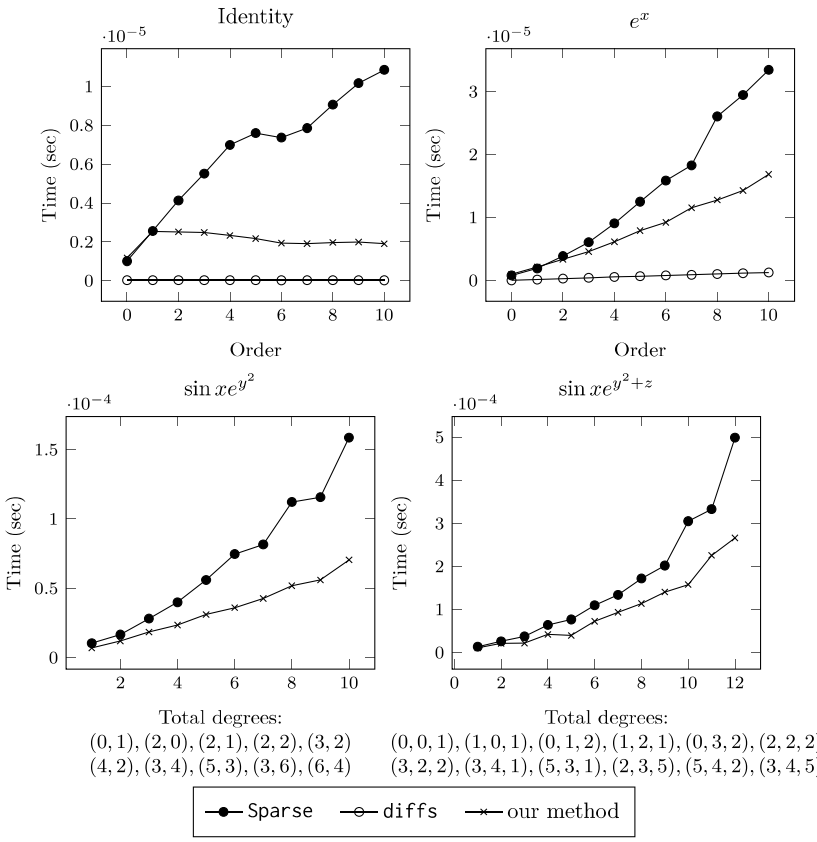


Figure 4: Speed benchmarks. `Sparse` and `diffs` are existing implementation in `ad` package.

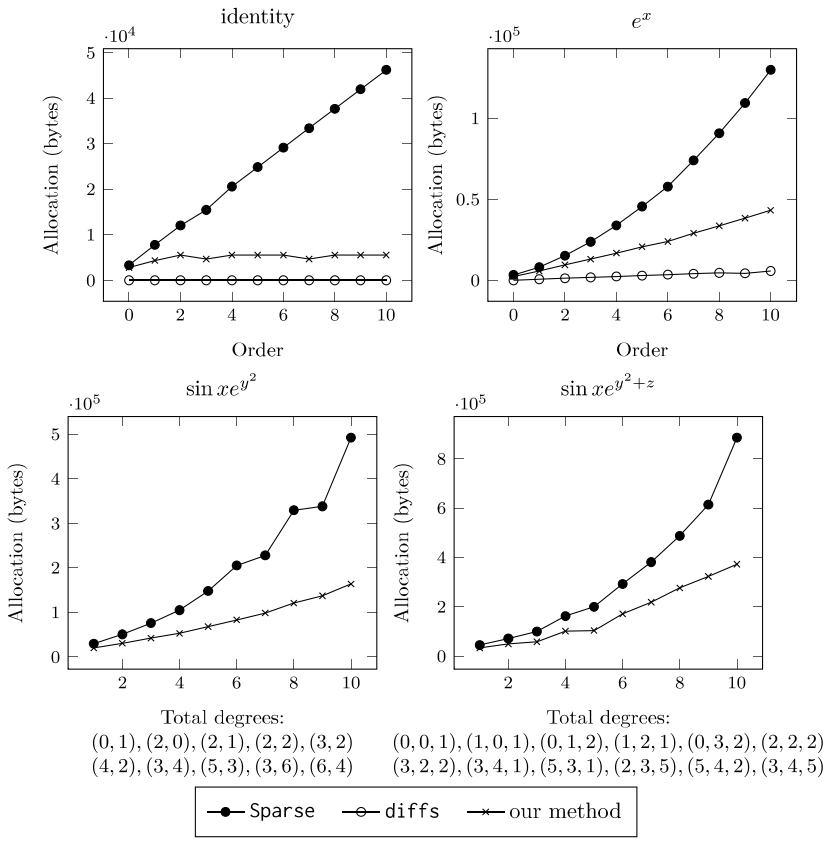
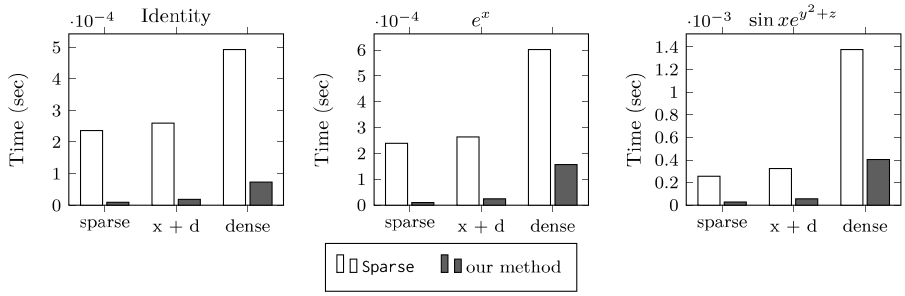
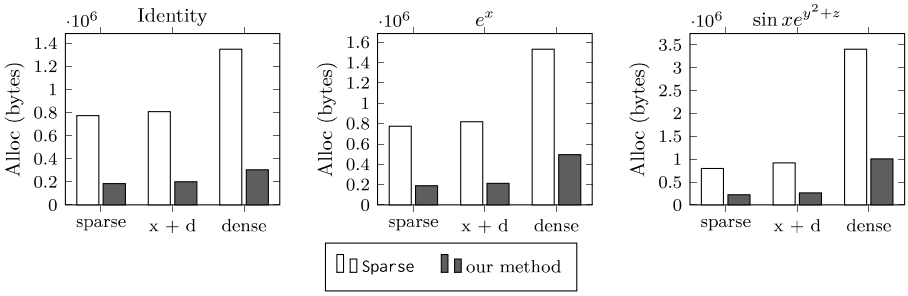


Figure 5: Heap benchmarks. `Sparse` and `diffs` are existing implementation in `ad` package.



`Sparse` is the existing implementation in `ad` package. As an input, the "sparse" gives 1 , "x + d" as it is, and "dense" $x +$ (sum of all nonzero basis).

Figure 6: Speed benchmarks for Weil algebra $W = \mathbb{R}[x, y]/(x^3 - y^2, y^3)$.



`Sparse` is the existing implementation in `ad` package. As an input, the "sparse" gives 1 , "x + d" as it is, and "dense" $x +$ (sum of all nonzero basis).

Figure 7: Heap benchmarks for Weil algebra $W = \mathbb{R}[x, y]/(x^3 - y^2, y^3)$.

3.2 Weil Algebra Computation

In this subsection, we compare the performances of Tower AD applied to Weil algebra computation as described in [5].

Figures 6 and 7 present the time and space performance of Weil algebra computation based on the existing implementation (`Sparse`) and the proposed method. In particular, we evaluated three functions (the identity, e^x, and $\sin x \cdot e^{y^2+z}$) on a Weil algebra $W = \mathbb{R}[x, y]/(x^3 - y^2, y^3)$. We feed three types of inputs: a sparse input (1), $1 + d$, and dense input $1 + d_1 + d_2 + d_1^2 + d_1 d_2 + d_2^2$. In any case, our proposed method largely outperforms the existing implementation.

4 Conclusion

We presented a succinct and efficient implementation of a multivariate lazy forward-mode tower automatic differentiation. This can be viewed as a mixture of existing methods but optimised exploiting the commutativity of partial differential operators. The basic idea is to store all the partial derivatives in some kind of a prefix-tree, in which the number of branching will eventually decrease as the right branch is chosen. We applied laziness and the advanced type-system in Haskell.

Our implementation performs particularly well in multivariate cases and gives pleasing improvements

both in time and space when applied to Weil algebra computation as described in [5]. For univariate case, however, there is much room for improvements. It might be a good future work to explore the special treatment in the univariate case to remove overheads.

Acknowledgements

The author would like to thank Prof. Akira Terui for encouraging me to participate in the workshop.

References

[1] Conal Elliott. "Beautiful differentiation". In: *International Conference on Functional Programming (ICFP)*. 2009. URL: http://conal.net/papers/beautiful-differentiation.

[2] Vincent Hanquez. gauge*: small framework for performance measurement and analysis*. 2019. URL: https://hackage.haskell.org/package/gauge (visited on 03/21/2021).

[3] haskell.org. *Haskell Language*. 2021. URL: https://www.haskell.org (visited on 03/11/2021).

[4] Hiromi Ishii. "A Purely Functional Computer Algebra System Embedded in Haskell". In: *Computer Algebra in Scientific Computing* (Lille, France). Ed. by Vladimir P. Gerdt, Wolfram Koepf, and Werner M. Seiler. Vol. 11077. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 288–303. ISBN: 978-3-319-99638-7. DOI: 10.1007/978-3-319-99639-4_20. arXiv: 1807.01456.

[5] Hiromi Ishii. "Automatic Differentiation With Higher Infinitesimals, or Computational Smooth Infinitesimal Analysis in Weil Algebra". submitted to ISSAC 2021. 2021.

[6] Jerzy Karczmarczuk. "Functional Differentiation of Computer Programs". In: *Higher-Order and Symbolic Computation* 14.1 (2001), pp. 35–57. DOI: 10.1023/A:1011501232197. URL: https://doi.org/10.1023/A:1011501232197.

[7] Edward A. Kmett. ad*: Automatic Differentiation*. 2010. URL: https://hackage.haskell.org/package/ad (visited on 2020).

[8] Barak Pearlmutter and Jeffrey Siskind. "Lazy multivariate higher-order forward-mode AD". In: vol. 42. Jan. 2007, pp. 155–160. DOI: 10.1145/1190216.1190242.

DeepFlow, Inc.
3-16-40 Tsuruse nishi, Fujimi-shi, Saitama prefecture, Japan.
E-mail address: h-ishii@math.tsukuba.ac.jp

DeepFlow 株式会社　石井大海