Comparing Use-Cases of Tree-Fold vs Fold-Left

How to fold and color a map

Jim Newton
EPITA, School of Engineering and Computer Science
LRE, EPITA Research Lab
Le Kremlin-Bicêtre, FRANCE
jnewton@lrde.epita.fr

Andreas Klebinger
Well-Typed
London, AUSTRIA
klebinger.andreas@gmx.at

ABSTRACT

We examine some consequences of computation order for two different implementations of the `fold` function. We explore a set of performance and accuracy-based experiments on two implementations of this function. In particular, we contrast results for the traditional `fold-left` implementation with another approach we refer to as `tree-fold`—looking at operations with non-constant complexity or accuracy: rational arithmetic, floating-point arithmetic and Binary Decisions Diagram construction. These are binary operations for which the performance or accuracy of the result varies based on the order of computation. We show that these types of binary operations are good candidates for `tree-fold`.

CCS CONCEPTS

• **Theory of computation → Theory and algorithms for application domains**.

KEYWORDS

fold,scala,BDD,associativity,binary decision diagram,lisp,rational numbers

ACM Reference Format:
Jim Newton and Andreas Klebinger. 2022. Comparing Use-Cases of Tree-Fold vs Fold-Left: How to fold and color a map. In *Proceedings of The 34th Symposium on Implementation and Application of Functional Languages (IFL'22).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The higher-order function[1, Sec 1.3], `fold` [14], is present in many programming languages. Definition 1.1 shows the essence of the function specifying the features pertinent to our research.

Definition 1.1. Let $f : D \times D \longrightarrow D$ be an associative function and $x_{[1,n]} = (x_1, x_2, ..., x_n) \in D^n$ for $n > 0$ be a sequence of values from D. Then we define *fold* as follows.

$$fold(f, x_{[1,n]}) = \begin{cases} x_1 & \text{if n = 1} \quad (1a) \\ f(fold(f, x_{[1,n-1]}), x_n) & \text{n > 1} \quad (1b) \end{cases}$$

If D is a monoid [31], with unity, e, we may also define `fold` on the empty sequence $fold(f, []) = e$.

The `fold` function is useful for extending a binary function to multiple arity. It is generally not required that the binary function in question be commutative. In case the operation is commutative, then `fold` extends trivially to unordered collections.

The notation is cleaner if we denote such a binary function as an operator, $\circ$, rather than as a function application. Given that we can compute $x_1 \circ x_2$, we may use the `fold` function to compute: $x_1 \circ x_2 \circ ... \circ x_n$. Because $\circ$ is assumed to be associative (but not necessarily commutative) we are free to *group* the terms how ever we like, as long as we respect the order.

$$\begin{aligned} \texttt{fold}(\circ, (x_1, ..., x_n)) &= (((x_1 \circ x_2) \circ x_3) ... \circ x_n) & (2) \\ &= (x_1 \circ x_2) \circ (x_2 \circ x_3) \circ ... \circ (x_{n-1} \circ x_n) & (3) \\ &= (x_1 \circ ...(x_{n-2} \circ (x_{n-1} \circ x_n))...) \\ &= etc. \end{aligned}$$

Even though all these groupings compute the same result mathematically, we will show that some have different performance or accuracy characteristics. In this article we look at two such groupings which we call `fold-left` (Section 4.1) and `tree-fold` (Section 4.2). The first grouping, `fold-left`, implements the *standard* algorithm used in most programming language implementations, and implements Definition 1.1 by following the computation directly as described in Equations (1b) and (2). The second grouping, `tree-fold`, implements Definition 1.1 by taking advantage of associativity and by grouping pairs such as in Equation (3), evaluating those pairs to obtain another sequence of values, to which Equation (3) is applied recursively.

2 MOTIVATION

To better understand the connection between BDDs and the `fold` operation, in this section we present a short summary of our larger research project.

A BDD, is a data structure which represents a Boolean function. Figure 2.1 illustrates the function,

$$\neg ((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)).$$

BDDs have many nice features which we exploit.

- Syntactically different Boolean functions having the same truth table, have isomorphic BDD representations. If coded as Bryant [3] suggests, the BDDs are represented by pointers to the same memoized object.

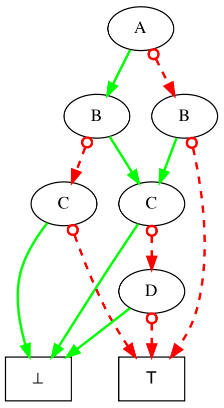


Figure 2.1: A BDD is a data structure which canonically represents a Boolean function.

- Symbolic manipulation of Boolean expressions such as intersection, union, and relative complement are graph (traversal and construction) operations.
- Vacuity checks are done in constant time.
- Satisfying a Boolean function is done in linear time, once the BDD is constructed.

For graphical display of a BDD, we use an arrow drawing convention.[1] In particular, `Boolean true` arrows are green solid lines, and `Boolean false` arrows are red dashed lines with an addition logical inversion bubble indicated. That being said, in Figure 2.1, the right-most path from A to $\top$ should be read as $\neg A \wedge \neg B$, with A and B being negated as the exit arrows are red-dashed. The set of all such top-to-$\top$ traversals, of which there are 4, forms a DNF expansion of the Boolean function.

$$\begin{aligned}\neg\,(AC \vee BC \vee BD) = &\, A\neg B\neg C\\ &\vee\ AB\neg C\neg D\\ &\vee\ \neg AB\neg C\neg D\\ &\vee\ \neg A\neg B\end{aligned}$$

In a BDD, variables are ordered. In this article we impose the top-to-bottom order of $A < B < C < D$ and $x_1 < x_2 < \cdots < x_n$. *E.g.*, A is above C, and x_2 is above x_4.

In [22], we presented a technique for reasoning about the types of heterogeneous sequences in Common Lisp [2]. We modeled sequences of types with deterministic symbolic finite automata [7], *i.e.* DFA over infinite alphabets—the infinite alphabet being the set of values supported by Common Lisp. Any subset of this set of values is called a *type*. Each transition in a DFA is labeled with a type designator, such that the set of transitions leaving any given state is a (pairwise disjoint) partition of the set of all Common Lisp values. In order to assure the property of determinism is maintained during finite automata operations (such as construction, combination [intersection, union, complement], and minimization) it is necessary to perform type computations similar to what is described in semantic type theory [9], except that we are dealing with a dynamically typed language rather than a statically typed one.

We use BDDs to represent types, and consequently represent type intersection, union, and complement operations as the corresponding BDD operations. When a type equivalence cannot be proven at compile time, the DFA may as a result contain redundant, useless, transitions which have a run-time performance penalty.

In [22, Sec 5.5] we reported that the particular implementation of `tree-fold` which was used in BDD construction, sometimes made an *unexpected* difference in construction time. The comment in that work was that `tree-fold` deserves further systematic study.

An admitted shortcoming of [22] was that we exclusively used Common Lisp as implementation language. The perspectives of that thesis indicated that we should attempt to generalize our results to apply to a wider audience and wider class of programming languages. Toward this end, for this article we have chosen Scala [6, 24] as the primary implementation language. Scala is a strictly typed language compiled to the Java Virtual Machine allowing the run-time environment to take advantage of garbage collection and the JVM run-time optimization.

In [22] we constructed BDDs from a Boolean formula, usually given in a DNF (sum of products) form such as: $x_1x_2\overline{x_3} + \overline{x_1}x_2x_3 + x_1\overline{x_2}x_3\overline{x_4}$. As Bryant *et al.* [4, 11] explain, such BDD construction can be exponential in complexity. To gather heuristics about time and space complexity of BDD construction in practice, in [23] we analyzed samples of BDDs of different numbers of variables to predict ranges of expected sizes.

In order to represent such a Boolean function programmatically, we suppose that Γ is a finite set of variables and their complements such as $\Gamma = \{x_1,\ \overline{x_1},\ x_2,\ \overline{x_2},\ \ldots,\ x_n,\ \overline{x_n}\}$.

Definition 2.1. A subset, γ, of Γ is called *contradictory* if $\{x_i, \overline{x_i}\} \subset \gamma$ for some $1 \le i \le n$. A subset of Γ which is not contradictory is called *consistent.*

Let $S = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ be a set of consistent subsets of Γ. Consider a Boolean formula in DNF (disjunctive normal form) such as:

$$DNF = \sum_{i=1}^{m} \prod \gamma_i = \sum_{i=1}^{m} \prod_{x \in \gamma_i} x\,. \tag{4}$$

This sum of products is computed as two concentric `fold` operations,[2] as shown in Figure 2.2. We assume the existence of a binary function `BddAnd` along with its neutral element `BddTrue` which performs the Boolean intersection operation between two objects of type `Bdd`, and as well, the existence of a binary function `BddOr` along with its neutral element `BddFalse` which performs the Boolean union operation between two `Bdd` objects.

This close connection between the `fold` operations and BDD operations motivated the investigation leading to this article. We noticed that changing the association (moving the parentheses) of Boolean functions *sometimes* had a noticeable effect of construction

[1]Our arrow drawing convention is both safe for black-white printing and also friendly for readers suffering from color blindness.

[2]https://users.scala-lang.org/t/expressing-a-sum-of-products-as-a-fold/5314 Thanks to Matthew Rooney, @javax-swing, for suggesting the concise implementation shown here.

```
def sumOfProducts[A](seq:Seq[Seq[A]])(plus:(A,A)=>A,
                                      zero:A,
                                      times:(A,A)=>A,
                                      one:A):A = {
  seq.foldLeft(zero) {
    (sum, gamma) => plus(sum, gamma.foldLeft(one)(times))
  }
}

// example usage, returns integer sum of products 6006006
sumOfProducts( Seq(Seq( 1, 2, 3), Seq(10, 20, 30), Seq(100, 200,
      300)))(
  plus = _ + _,  zero = 0,
  times = _ * _, one = 1)

// example usage, returns BDD which is an OR of ANDs
// of the given BDDs
sumOfProducts( Seq(seq1ofBdds, seq2ofBdds, sea3ofBdds))(
  plus = BddOr,   zero = BddFalse,
  times = BddAnd, one = BddTrue)
```

Figure 2.2: Scala sum-of-products and usage examples.

times. Unfortunately, we could not infer any practical characterization of this phenomenon. We noted in [22] that more research was needed. The BDD construction computations in Section 5.4 are steps in systematically investigating these effects.

3 HISTORICAL CONTEXT

Because of its higher-order nature, the fold function was originally conceived for functional-style languages. One might guess that the earliest appearance of fold would have been in Lisp. While Lisp 1.5 [19, 20] did have the functions MAP and MAPCAR [26], we found no reference to the fold function.

As far as we can determine, David Turner (author of SASL and Miranda), seems to be[3] the inventor of fold [29]. In 1986, Turner [28] shows how to implement foldr in Miranda. The earliest mention of fold that we have found, was from 1979 where Turner [30] mentions that "folding a list to the right" is a "commonly occurring pattern" and encapsulates the pattern by defining foldr in SASL.

The function is called REDUCE in Common Lisp [2]; in Scala [6, 24], foldLeft, foldRight and others; in Haskell [15, p. 115], foldl and foldr and others; and in OCaml [25, p. 63] fold_left.

In recent times, many tools of functional programming languages have made their way into many other languages which are traditionally thought of as imperative or object-oriented [10, 27]. Although it is not a definitive source of information, we note that the Wikipedia article on Fold[4] lists ≈ 44 programming languages which support this feature, sometimes with different names such as reduce, accumulate, aggregate, compress, or inject.

The tree-fold algorithm that we present in this paper is reminiscent of the map-reduce algorithm presented by Dean and Jeffrey [8]. Whereas, map-reduce, is generally intended to for parallelizing a fold computation for performance reasons, fold-left, is an easy way to reduce work independent of whether the work is done in parallel or sequentially—easy in terms of lines of code.

[3]https://www.quora.com/Where-did-the-common-functional-programming-functions-get-their-names Mark Harrison inlines an email form David Turner claiming to be the inventor of the foldr/foldl functions sometime between 1976 and 1983. We verified this claim in a face-to-face conversation with David Turner.

[4]https://en.wikipedia.org/wiki/Fold_(higher-order_function), last edited on 5 November 2019, at 05:48.

4 COMPUTING A FOLD OPERATION

In Section 5 we will investigate operations on BDDs as alluded to in Section 2. Rather than doing so straightaway, instead we have first devised experiments based on arithmetic of rational and floating point numbers. We have chosen this diversion to illustrate the principles of fold to the reader without being required to understand the subtle inner-workings of a BDD library. Rational number arithmetic is easy to illustrate and intuitive to understand. In particular, we explore the task of summing a sequence of fractions, each expressed as the ratio of two integers.

$$\frac{1}{23}+\frac{1}{29}+\frac{1}{31}+\frac{1}{37}+\frac{1}{41}+\frac{1}{43}+\frac{1}{47}+\frac{1}{53}+\frac{1}{57}+\frac{1}{67} = \frac{3,304,092,302,051,372}{12,831,131,327,329,923} \tag{5}$$

Computing the sum in (5) involves representing the numerators and denominators as bignum integer type. [16, Sec 4.5] Integers in Common Lisp are specified to have unlimited precision, and the built-in ratio type provides precise fractions whose numerators and denominators never *roll-over*. However in Scala, integers do not have this feature; thus we use an external library, spire.math.Rational which provides a type called Rational.

Regardless programming language and implementation of ratio, each rational addition must compute some variant of

$$\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1\cdot d_2+n_2\cdot d_1}{d_1\cdot d_2}\,, \tag{6}$$

including explicit or implicit cancellation of common factors in the numerator and denominator. There are several strategies to optimize such a computation. *E.g.*, if $g = gcd(d_1, d_2)$, the greatest common divisor, then the sum can be computed as in (7). Since $d_3 = \frac{d_1}{g}$ and $d_4 = \frac{d_2}{g}$ are integers, the quotient can be rewritten as

$$\frac{g\cdot n_1\cdot\frac{d_2}{g}\;+\;g\cdot n_2\cdot\frac{d_1}{g}}{\left(g\cdot\frac{d_1}{g}\right)\cdot\left(g\cdot\frac{d_2}{g}\right)} = \frac{g\cdot n_1\cdot d_4\;+\;g\cdot n_2\cdot d_3}{(g\cdot d_3)\cdot(g\cdot d_4)} = \frac{n_1\cdot d_4+n_2\cdot d_3}{g\cdot d_3\cdot d_4} \tag{7}$$

According to Theorem 4.1, (6) can be computed by an application of (7), but involving smaller numbers, in ≈ 40% of the cases.

Theorem 4.1 (G. Lejeune Dirchlet, 1849). *If d_1 and d_2 are chosen at random, then the probability that $gcd(d_1, d_2) = 1$ is $6/\pi^2 \approx$ 60.793%.*

Knuth [16, page 342] proides a proof of Dirchlet's theorem in Section 4.5.2 of *Art of Computer Programming*. If $gcd(d_1, d_2) \neq 1$, Knuth [16, page 330] suggests the following to calculate n_3 and d_3 such that $\frac{n_3}{d_3} = \frac{n_1}{d_1} + \frac{n_2}{d_2}$. Equations (8) and (9) are an improvement over (7) in the case numerator and denominator of (7) have a common factor.

$$\begin{aligned} g_1 &= gcd(d_1, d_2) \\ t &= n_1 \cdot (d_2/g_1) \ + \ n_2 \cdot (d_1/g_1) \\ g_2 &= gcd(t, g_1) \\ n_3 &= t/g_2 \qquad (8) \\ d_3 &= (d_1/g_1) \cdot (d_2/g_2) \qquad (9) \end{aligned}$$

Regardless of the implementation or optimizations a given rational number library uses, for sufficiently large denominators, adding fractions becomes more computationally intensive as the denominators grow. *E.g.*, it is easier to add $\frac{1}{2} + \frac{2}{3}$ than to add $\frac{105,000}{765,049} + \frac{385,544}{4,391,633}$.

Mollin [21] argues that $gcd(a, b)$ can be computed in $O(\log^3 max(a, b))$. Since $\log max(a, b)$ is roughly the number of digits in the larger of a and b, we see that if the larger is an n-digit number, then the complexity of computing $gcd(a, b)$ is $O(n^3)$. Since multiplication and division have $O(n^2)$ complexity, Knuth's proposed algorithm has cubic complexity in the number of digits.

4.1 Strategy using `fold-left`

The `fold-left` function computes the result of $x_1 \circ x_2 \circ \ldots \circ x_{i-1}$ before combining that result with x_i, grouping these addition operations as follows:

$$\underbrace{((((((\underbrace{(\underbrace{(\underbrace{\frac{1}{23}+\frac{1}{29}}_{\#\,1})+\frac{1}{31}}_{\#\,2})+\frac{1}{37}}_{\#\,3\,\ldots})+\frac{1}{41})+\frac{1}{43})+\frac{1}{47})+\frac{1}{53})+\frac{1}{57})+\frac{1}{67}}_{\ldots\ computation\ \#\,9} \qquad (10)$$

Such a computation computes eight intermediate values before arriving at the final value. Figure 4.1 denotes these as computations # 1 through # 8, and final value as a result of computation # 9. The **Digits Computed** column records the total number of digits (numerator digits plus denominator digits) accumulated from computation # 1 until the row in question. These values are plotted in Figure 4.4 (top). We compare the cumulative number of digits also for the analogous experiments which follow in Section 4.2. The **Digits Retained** column records the number of digits (again numerator digits plus denominator digits) which must be held in memory pending a future computation.

We present these two columns (**Digits Computed** and **Digits Retained**) as it is conceivable that they effect the computation time. *I.e.*, we suppose the *gcd* computations which are calculated to perform the rational number additions are dependent on the number of digits (roughly dependent on the logarithms of the numbers), and also that computations which retain large amounts of heap-allocated objects might decrease performance of computation.

4.2 Strategy using `tree-fold`

The `tree-fold` algorithm, described here, attempts to retain as few intermediate values as possible, by consuming the values as soon as possible, yet respecting the grouping shown in Equation (11).

$$\underbrace{\Big(\underbrace{\big(\underbrace{(\underbrace{\frac{1}{23}+\frac{1}{29}}_{\#\,1})+(\underbrace{\frac{1}{31}+\frac{1}{37}}_{\#\,2})}_{\#\,3}\big)+\big(\underbrace{(\underbrace{\frac{1}{41}+\frac{1}{43}}_{\#\,4})+(\underbrace{\frac{1}{47}+\frac{1}{53}}_{\#\,5})}_{\#\,6}\big)}_{\#\,7}\Big)+(\underbrace{\frac{1}{57}+\frac{1}{67}}_{\#\,8})}_{computation\#\,9} \qquad (11)$$

As shown graphically in Figure 4.2 (top), the `tree-fold` approach consumes #1 and #2, immediately in computing #3. However, the value returned from computation #3 is held until #4 and #5 and combined in #6 at which point both results #3 and #6 are combined in computation #7. In Equation (5), `tree-fold` retains at most $\log_2 n$ intermediate values; n being the total length of the sequence being combined in Equation (5).

Figure 4.2 (bottom) shows the Scala code for `tree-fold`. The code has three parts, A, B, and C. Part A deals with *dwindling* the tree, the green nodes, into several components each of size 2^k for some k. Part B reads the input sequence, the pink nodes, and feeds these to the `dwindle-tree` function. Finally, part C handles the remaining nodes including the nodes computed in part A. There are maximally $\log_2 n$ such nodes so one might think (as we initially did) that it does not matter which order these are handled. For example, one might simply use `fold-left` to iterate the operator over these nodes. That would be a mistake for two reasons.

(1) The operator is not always commutative, so care must be taken to assure the rhs and lhs are never inverted.
(2) As we are supposing the size of the computed objects grows as `fold` iteration progresses, we should postpone computation involving values already computed, until smaller objects from the input sequence have been handled.

The easiest way to satisfy these two requirements is to reverse the stack of computed values, and recursively call the `tree-fold`.

This recursion will terminate with a stack of one single node. That this recursion terminates is easy to see, because if there is a stack of two or more, at least two will be combined into a single node, thus the stack size will decrease by at least one on each recursive call. In actuality, it will almost always decrease by more than one leaving at most $\log_2$ of the size of the input sequence — a fact which is not necessary to realize in order to prove termination.

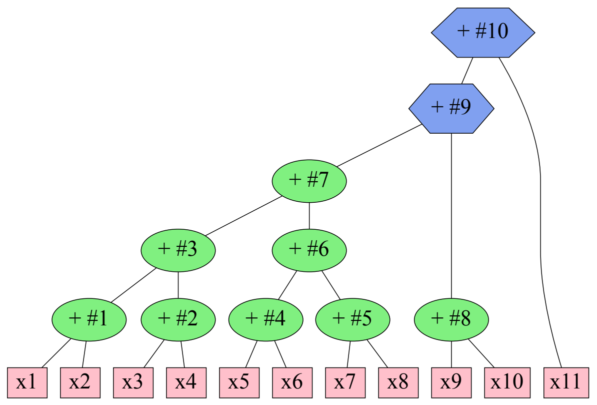
As in Section 4.1, once again we look at the number of digits computed and retained by `tree-fold`. When we observe the **Digits Computed** column of Figure 4.3 and the curve in Figure 4.4 (top) corresponding to `tree-fold`, we see at computation #9.

4.3 Summarizing the two `fold` strategies

The two sequences of computations from Sections 4.1 and 4.2 are recapped in Figure 4.4. From the **Digits Computed** column of Figures 4.1 and 4.3 and the curve in Figure 4.4 (top), we see that `fold-left` computes the more digits of the two strategies. In terms of number of digits computed, it is the worse of the two alternatives. However, from the **Digits Retained** column of Figures 4.1 and 4.3 and the corresponding curve in Figure 4.4 (bottom) we see that it `fold-left` retains less heap storage during the computation.

Compu. #	Ratio Addition	Result	Digits Computed	Digits Retained
# 1	$\frac{1}{23} + \frac{1}{29}$	$= \frac{52}{667}$	5	5
# 2	$\frac{52}{667} + \frac{1}{31}$	$= \frac{2279}{20,677}$	+9 = 14	9
# 3	$\frac{2279}{20,677} + \frac{1}{37}$	$= \frac{105,000}{765,049}$	+12 = 26	12
# 4	$\frac{105,000}{765,049} + \frac{1}{41}$	$= \frac{5,070,049}{31,367,009}$	+15 = 41	15
# 5	$\frac{5,070,049}{31,367,009} + \frac{1}{43}$	$= \frac{249,379,116}{1,348,781,387}$	+19 = 60	19
# 6	$\frac{249,379,116}{1,348,781,387} + \frac{1}{47}$	$= \frac{13,069,599,839}{63,392,725,189}$	+22 = 82	22
# 7	$\frac{13,069,599,839}{63,392,725,189} + \frac{1}{53}$	$= \frac{756,081,516,656}{3,359,814,435,017}$	+25 = 107	25
# 8	$\frac{756,081,516,656}{3,359,814,435,017} + \frac{1}{57}$	$= \frac{46,456,460,884,409}{191,509,422,795,969}$	+29 = 136	29
# 9	$\frac{46,456,460,884,409}{191,509,422,795,969} + \frac{1}{67}$	$= \frac{3,304,092,302,051,372}{12,831,131,327,329,923}$	+33 = 169	33

Figure 4.1: Intermediate and final values of adding 10 ratios, default `fold-left` algorithm, computed as shown in (10).



```scala
def treeFold[A](m: List[A])(z: A)(f: (A, A) => A): A = {

  // A: Handle the green oval nodes
  def dwindleTree(stack: List[(Int, A)]): List[(Int, A)] = {
    stack match {
      case (i, b1) :: (j, b2) :: tail if i == j =>
        dwindleTree((i + 1, f(b2, b1)) :: tail)
      case stack => stack
  }}

  // B: Handle the pink rectangle nodes
  m.foldLeft((1, z) :: Nil) { (stack: List[(Int, A)], ob: A) =>
    dwindleTree((1, ob) :: stack)

  } match { // C: Handle the blue hexagon nodes
    case Nil => z
    case (_,b)::Nil => b
    case stack => treeFold(stack.map(_._2).reverse)(z)(f)
 }
}
```

Figure 4.2: Top: Topological view of the `tree-fold` operation. Bottom: Scala implementation of the `tree-fold` algorithm.

The rational numbers in question, Equation (5), have been especially chosen to be a difficult case. The denominators are all prime numbers, assuring that the $gcd = 1$ in every case, and thus the sizes of the ratios, in terms of number of digits will be monotonically increasing. As was mentioned in Theorem 4.1, 40% of the time, the gcd will be different than 1. Cases for which the denominators are less often relatively prime may not see as drastic a difference in the performance between `tree-fold` and `fold-left`.

5 EXPERIMENTS AND RESULTS

First, in Section 5.1, we examine the computation time results of the two `fold` algorithms when applied to ratio additions as explained in Sections 4.1 and 4.2. Next, in Section 5.3, we examine the accuracy of certain floating-point computations which exploit the `fold` algorithm. Finally, in Section 5.4 we examine the results when we apply the same techniques to BDD construction.

5.1 Ratio Addition

The first experiment we performed entailed summing sequences of rational numbers of incrementally increasing length. The plots in Figure 5.1 show the computation time of computing sums of different length sequences, using two different folding algorithms. The plots in Figure 5.1 differ in that the top plot is the result of summing the sequence in sorted order, and the bottom, shuffled into random order. The x-axis indicates the value of n and the y-axis indicates the time needed to compute the sum

$$\sum_{-n \leq i \leq -1} \frac{1}{i} + \sum_{1 \leq i \leq n} \frac{1}{i} = 0 \qquad (12)$$

whose sum is expected to be zero. *I.e.*, we sum the negative and positive fractions of the form $1/i$, for $-n \leq i \leq n$, excluding $1/0$.

For each value of n, the sum was performed in two different ways as outlined in Sections 4.1 and 4.2. It is clear from Figure 5.1, that `tree-fold` performs better especially as the value of n grows, particularly for values of $n > 100$. The benefit gained from `tree-fold`

Compu. #	Ratio Addition	Result	Digits Computed	Digits Retained
# 1	$\frac{1}{23} + \frac{1}{29}$	$= \frac{52}{667}$	5	5
# 2	$\frac{1}{31} + \frac{1}{37}$	$= \frac{68}{1147}$	+6 = 11	11
# 3	$\frac{52}{667} + \frac{68}{1147}$	$= \frac{105{,}000}{765{,}049}$	+12 = 23	12
# 4	$\frac{1}{41} + \frac{1}{43}$	$= \frac{84}{1763}$	+6 = 29	18
# 5	$\frac{1}{47} + \frac{1}{53}$	$= \frac{100}{2491}$	+7 = 36	25
# 6	$\frac{84}{1763} + \frac{100}{2491}$	$= \frac{385{,}544}{4{,}391{,}633}$	+13 = 49	25
# 7	$\frac{105{,}000}{765{,}049} + \frac{385{,}544}{4{,}391{,}633}$	$= \frac{756{,}081{,}516{,}656}{3{,}359{,}814{,}435{,}017}$	+25 = 74	25
# 8	$\frac{1}{57} + \frac{1}{67}$	$= \frac{124}{3819}$	+7 = 81	32
# 9	$\frac{756{,}081{,}516{,}656}{3{,}359{,}814{,}435{,}017} + \frac{124}{3819}$	$= \frac{3{,}304{,}092{,}302{,}051{,}372}{12{,}831{,}131{,}327{,}329{,}923}$	+33 = 114	33

Figure 4.3: Intermediate and final values of added 10 ratios using default `tree-fold` algorithm, computed as shown in Equation (11).

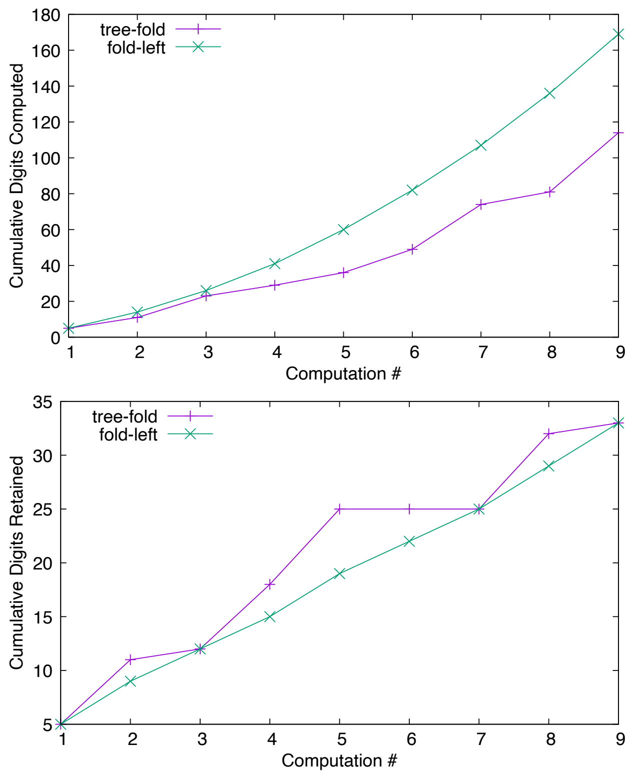


Figure 4.4: Top: Cumulative digits computed for each fold strategy. Bottom: Quantity of digits retained at each computation state.

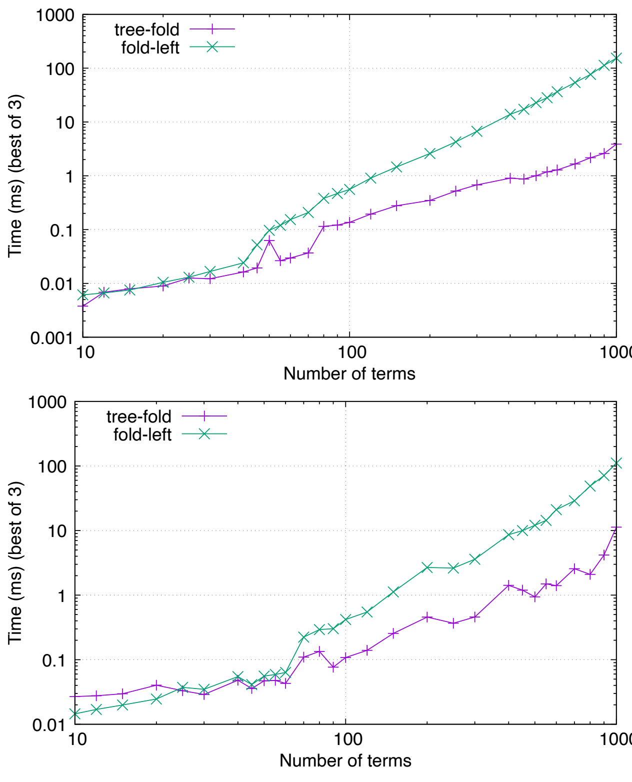


Figure 5.1: Computation time of `fold` strategy on rational addition. Fewer milliseconds is better than more. Top: Addition in sorted order. Bottom: Addition in randomized/shuffled order.

increases as measured by the gap between the `fold-left` and the other curve. This gap widens as *n* increases.

These results are promising and lend some credence to our hope that such techniques might also benefit BDD construction times.

```haskell
-- Input list is required to be non-empty
foldTree1 :: (a -> a -> a) -> [a] -> a
foldTree1 f xs = go f xs
  where
    -- Combine one level of the tree until we reduced the
    -- input to a single value.
    go f xs = case combine f xs of
        [x] -> x
        xs' -> go f xs'
    -- Combine one level of the tree
    combine :: (a->a->a) -> [a] -> [a]
    combine _f [] = []
    combine _f [x] = [x]
    combine f (x:y:xs) =
        let
            xs' = combine f xs
            x' = f x y
        in x' : xs'
```

Figure 5.2: A Haskell implementation of lazy `tree-fold`.

However, it does not appear that the amount of heap usage has any effect on computation time. Despite our analysis of the retained digits, we do not conclude any causal connection. This may be do to the memory management capabilities of the JVM.

Figure 5.4 shows the timing results of computing the sum in (12) vs the number of terms, n. Each data point indicates the minimum time of thrice evaluating the sum by the indicated `fold` strategy. We evaluate three times, because the JVM applies run-time optimizations as the program run, each time obtaining faster results.

5.2 Tree-Folds in Lazy Languages

Tree-folds extend to lazy languages like Haskell [15, p. 115] without issue. The main difference arises from being able to define strict and lazy variants of tree folds. Figure 5.2 shows an implementation of a lazy `tree-fold` in Haskell using lists for simplicity. It can be turned into a strict tree fold by adding strictness annotations to `x´` `xs´` evaluating them eagerly.

Using the code from Figure 5.2, the ghc[5] Haskell compiler (version 9.2.4), takes around 4.5 milliseconds[6] to compute $\sum_{k=1}^{5000} \frac{1}{k}$ using rational arithmetic. This compares to 0.17 seconds to perform the analogous computation using the `foldl` (i.e., `fold-lef`) approach.

Choosing between the use of strict and lazy `tree-fold` is similar to choosing between using the lazy `foldl` and the strict `foldl'` in Haskell. The lazy variant has the potential to avoid unneeded computation for operations if the result does not depend on both inputs. Meanwhile the strict version will compute all intermediate results but avoids some of the overhead associated with laziness. This is important as for less intensive operations the overhead of laziness can dominate.

The Clojure [12] language is not a lazy language per se, but mapping functions produce lazy sequences. Figure 5.3 shows a Clojure implementation of `tree-fold` inspired by the Haskell version in Figure 5.2. The function, `map`, applies the given function lazily, but in order, to each element of the given sequence. Furthermore, the function `partition` lazily converts the given sequence, `coll`, into a sequence of pairs, *e.g.* `[1 2 3 4 5 6]` is converted to `[(1 2) (3 4) (5 6)]`. If the sequence does not have even length then the final pair is padded with the given zero element, `z`.

[5]https://www.haskell.org/ghc/
[6]Figure 5.14 describes the hardware used.

```clojure
(defn tree-fold [f z coll]
  (cond
    (empty? coll)
    z

    (empty? (rest coll))
    (first coll)

    :else
    (tree-fold f z
               (map (fn [[a b]] (f a b))
                    (partition 2 2 [z] coll)))))
```

Figure 5.3: A Clojure implementation of lazy `tree-fold`.

Using the code in Figure 5.3, Clojure (version 1.11.1.1200) takes around 25 to 30 milliseconds to compute $\sum_{k=1}^{5000} \frac{1}{k}$ using rational around, while the same computation using `reduce`, the Clojure version of `fold-left`, needs around 2.5 seconds.[7]

In both the Haskell code and the Clojure code (Figures 5.2 and 5.3) we notice that the code itself is shorter. The laziness of the language obviates the need to manage the tree level. The Scala `tree-fold` implementation manages pairs indicating the tree-level and the computed value, in order to prevent accidentally combining values from different tree levels; `(1, ob)` on line 13 and `(i+1, f(b2, b1))` on line 7 of Figure 4.2. The Haskell and Clojure implementations have no need for such bookkeeping.

5.3 Floating Point Addition

Not only is computation time of concern, but also of concern is the fact that floating point arithmetic is not truly associative. Thus we get different cumulative round-off error depending on which version of `fold` is used. In this section, we demonstrate this problem of round-off error by two experiments.

In the first experiment, we sum the floating point numbers in (13).

$$\sum_{i=1}^{n} (i + \frac{1}{10}) \qquad (13)$$

For each value of n we can compute the exact (expected) value of this sum (using rational arithmetic), and compare it to the value computed in floating-point arithmetic. We compute the value using `fold-left` and also using `tree-fold`, and in each case we measure the total error $|computed - expected|$.

[7]Figure 5.14 describes the hardware used.

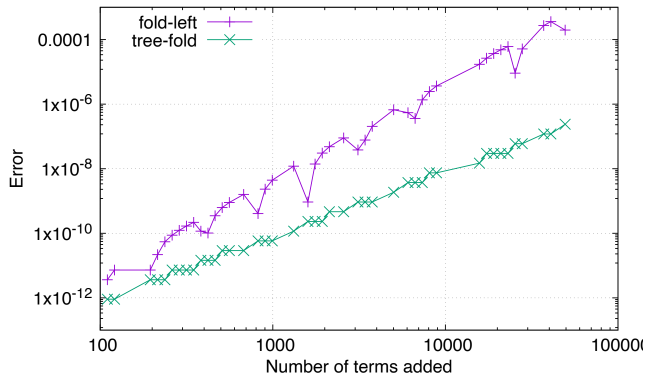


Figure 5.4: Error when adding progressively more floating point numbers. Less error is better than more.

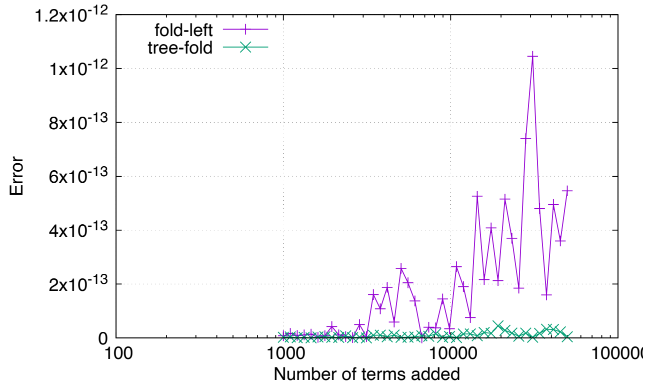


Figure 5.5: Error when adding progressively more randomly chosen numbers and their negatives. Less error is better than more.

For example

$$\begin{aligned}\sum_{i=1}^{1000}(i+\frac{1}{10}) &= \sum_{i=1}^{1000} i + \sum_{i=1}^{1000}\frac{1}{10}\\ &= \frac{1000\times 1001}{2}+\frac{1000}{10}\\ &= 500600\end{aligned}$$

We see in Figure 5.4 that the error increases linearly (log vs log scale) using both fold strategies, but tree-fold seems to exacerbate the round-off error less than fold-left. For example, when $n = 100$, rather than computing 500600, fold-left computes 500599.9999999954 (for a Δ of 4.60×10^{-9}) whereas tree-fold computes 500600.00000000006 (for a Δ of 5.82×10^{-11}).

Figure 5.5 shows the results of a second experiment with floating point numbers. In this experiment we collect between 1000 and 50,000 double precision floating point numbers, and their negatives, so that the total theoretical sum should exactly 0. We then shuffle the sequence and add them up using the two fold implementations. The plot shows the absolute value of the sum for each population size. Since the expected sum is zero, the absolute value of the sum is actually the error. We see again that tree-fold gives better results (less error), as the error is never beyond $\Delta = 1 \times 10^{-13}$ while the error for fold-left exceeds $\Delta = 1 \times 10^{-12}$.

It is easy to understand a reason for these results. With each floating-point operation, an inaccuracy is liable to be introduced. If that result is used in a subsequent computation, then the error is multiplied. The tree-fold algorithm reduces the number of times inaccurate results are reused in subsequent computations.

5.4 Computing with BDDs

As we mentioned in Section 2, the catalyst for this research into the fold operation, was BDD generation. Our original experiments involved generating random BDDs, *i.e.* a BDD from a randomly selected Boolean function, equivalently a randomly selected 2^n sized truth table. Each row of the truth table denoting a true value of the function, induces a γ-consistent subset as in (Definition 2.1) and constructing the BDD involves evaluating an expression such as Equation (4)— the bdd-and operation replacing the product, and bdd-or replacing the sum. As iteration through the outer sum progresses, the BDD generally becomes progressively larger.

The construction of BDDs of increasingly many Boolean variables is known to have exponential complexity [4, 11].

On average, a randomly selected DNF contains $\frac{2^n}{2} = 2^{n-1}$ minterms, and each such minterm contains 2^{n-1} true plus 2^{n-1} false Boolean variables.

Although randomly generated BDDs are useful in property-based testing for BDD libraries, such a BDD tends not to resemble BDDs coming from typical applications.

In a random sample, a minterm is likely to contain $\frac{n}{2}$ true literals and $\frac{n}{2}$ false literals. However, in many meaningful applications the Boolean variables are correlated in a way that each term contains a small number of literals. Langberg *et al.* [18] refer to such Boolean formulas as *simple*. Knuth [17, Sec 7.1.4] demonstrates a playful example: 4-coloring map problem. The constraint that two neighboring regions on the map be colored differently results in a conjunction of Boolean terms each involving exactly 4 literals.

Figure 5.6 is a colorized map of the countries in Europe. As Knuth [17, Sec 7.1.4] explains, this map can be colorized by finding any satisfying assignment of a Boolean function which is the product (Boolean And) of a set of constraints, one constraint for each common border. Each constraint specifies that a given pair of neighboring countries must not share the same color. For example, France and Spain share a border (Figure 5.7). Since we know the map can be colored with 4 colors (or possibly fewer), we may assign two variables to each country, and allow each of the 4 combinations of values to represent the 4 colors.

Denote the color of France as $(A_{🇫🇷}, B_{🇫🇷})$, Spain as $(A_{🇪🇸}, B_{🇪🇸})$, and Germany as $(A_{🇩🇪}, B_{🇩🇪})$. The constraints are as follows.

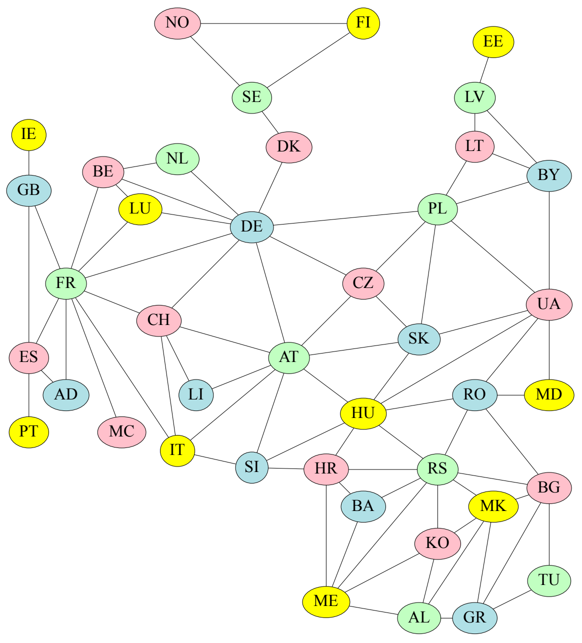


Figure 5.6: Coloring a map of some European countries

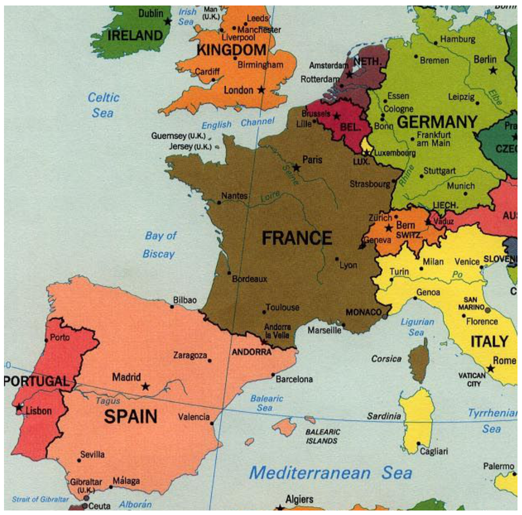


Figure 5.7: Neighbors of France. Map image from https://www.unomaha.edu, University of Nebraska Omaha.

$$\text{🇫🇷} \to \text{🇪🇸} = \left(\text{color}(\text{🇫🇷}) \neq \text{color}(\text{🇪🇸})\right)$$
$$= (A_{\text{🇫🇷}} \oplus A_{\text{🇪🇸}}) \vee (B_{\text{🇫🇷}} \oplus B_{\text{🇪🇸}})$$
$$= (A_{\text{🇫🇷}} \wedge \neg A_{\text{🇪🇸}}) \vee (\neg A_{\text{🇫🇷}} \wedge A_{\text{🇪🇸}})$$
$$\vee (B_{\text{🇫🇷}} \wedge \neg B_{\text{🇪🇸}}) \vee (\neg B_{\text{🇫🇷}} \wedge B_{\text{🇪🇸}}) \qquad (14)$$

$$\text{🇫🇷} \to \text{🇩🇪} = (A_{\text{🇫🇷}} \wedge \neg A_{\text{🇩🇪}}) \vee (\neg A_{\text{🇫🇷}} \wedge A_{\text{🇩🇪}})$$
$$\vee (B_{\text{🇫🇷}} \wedge \neg B_{\text{🇩🇪}}) \vee (\neg B_{\text{🇫🇷}} \wedge B_{\text{🇩🇪}}) \qquad (15)$$

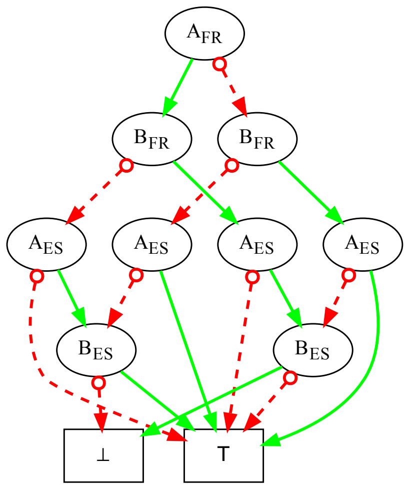


Figure 5.8: 🇫🇷 → 🇪🇸 BDD representing France/Spain border. 🇫🇷 → 🇩🇪 BDD of France/Germany border is isomorphic, but with different labels. (A_{FR}, B_{FR}),(A_{ES}, B_{ES}), and (A_{DE}, B_{DE}) are the colors of France, Spain, and Germany respectively.

We can conjoin all the constraints into one Boolean function:

$$(\text{ES} \to \text{FR}) \wedge (\text{FR} \to \text{DE}) \wedge (\text{PT} \to \text{ES}) \wedge (\text{FR} \to \text{BE}) \ldots$$

We can thus color the map by computing the function:

$$\bigwedge_{(\alpha,\beta)\in borders} \left(A_\alpha \oplus A_\beta\right) \vee \left(B_\alpha \oplus B_\beta\right). \qquad (16)$$

Constructing the BDD takes exponential time in average case, but satisfying the corresponding function, given the BDD can be done in linear time.

To compute the BDD corresponding to Equation (16), we iterate over the constraints, construct a 4-variable BDD for each constraint, and combine all such BDDs, compute the `bdd-and`, using `fold`.

For example, when we construct the 4-variable BDD for the constraint 🇫🇷 → 🇪🇸 in Equation (14), we construct the BDD shown in Figure 5.8. Similarly the BDD for 🇫🇷 → 🇩🇪 in Equation (15) is shown in Figure 5.8.

As the `fold` iterates, we must combine the constraints. Suppose that we combine (logical-and) the constraints in Equations (14) and (15). In this case the `bdd-and` operation is called to produce the BDD corresponding to the intersection of the two Boolean functions to arrive at the BDD shown in Figure 5.9, which you should notice, is *slightly less than* twice the size as either of the input BDDs. *I.e.*, these two input BDDs of 11 nodes each, combine to form a BDD of 19 nodes. This exponential growth is discussed in Section 5.5. The `fold` then continues to iterate in like manner, culminating into a final BDD, which represents all the border constraints. This final BDD has around 200K nodes (right-most point in Figure 5.11 (top)).

5.5 Analysis of Map Coloring Results

Figure 5.10 (top) shows the wall-time to colorizing connected sub-graphs of the map of Europe, for successively larger connected sub-graphs (number of countries indicated on the x-axis). We see in the plot that for this particular Boolean function, BDD construction using `tree-fold` is faster than `fold-left`, although the difference

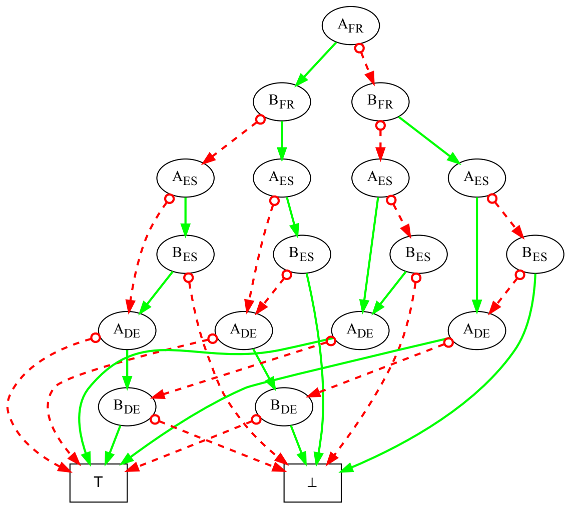


Figure 5.9: Two constraints corresponding to France/Spain and France/Germany

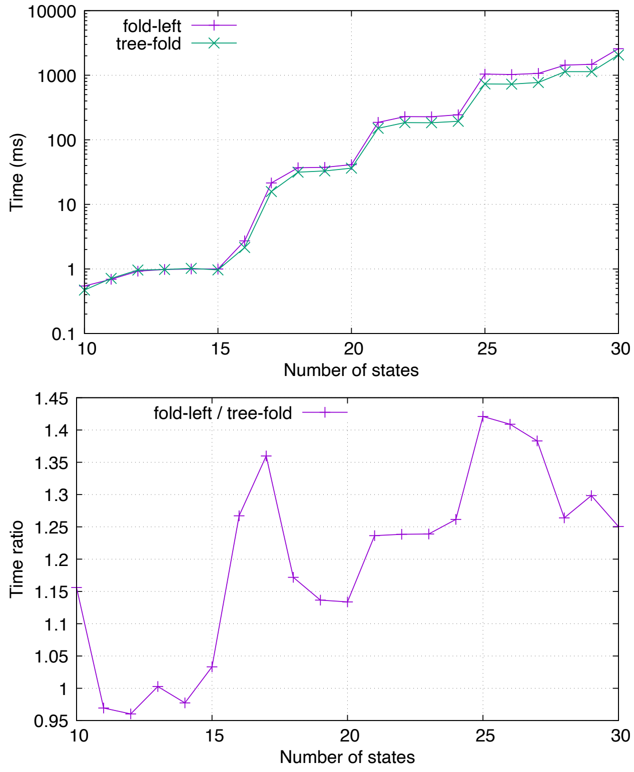


Figure 5.10: Top: Time to colorize successively larger sub-maps of Europe. Bottom: Ratio of `tree-fold` to `fold-left`; > 1 means `tree-fold` is faster; < 1 means `fold-left` is faster.

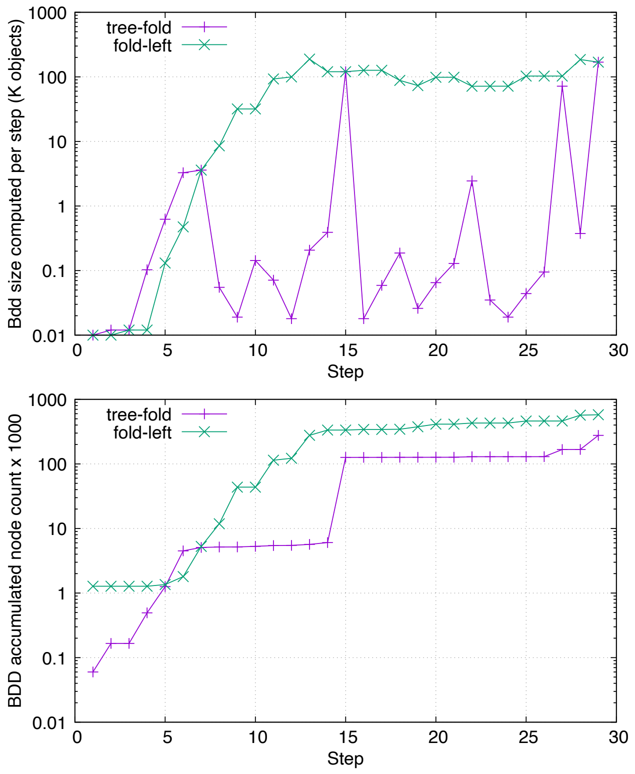


Figure 5.11: BDD node count. Top: Nodes allocated at each step of the iteration. Bottom: Total node accumulated thus far. Top is effectively the derivative of the bottom. Fewer nodes allocated is better than more.

is not as pronounced as in our previous examples, and admittedly not as much a performance improvement as we had hoped for. Because the curves in Figure 5.10 (top) may be difficult to distinguish with the eye, we also provide Figure 5.10 (bottom) which shows the ratio of $\frac{\texttt{tree-fold}}{\texttt{fold-left}}$. When this quantity is greater than 1, `tree-fold` is faster by the factor indicated. *E.g.*, when the ration is 1.4, then `fold-left` took 1.4× the computation time of `tree-fold`.

The speed improvement for BDD computations is a positive advance, but the improvement is not as much as we would like. In Section 5.6 we attempt to account for the timing results.

5.6 Run-time Probing the Running JVM

In Figures 5.11 5.12, and 5.13 we analyze the timing results from Figure 5.10. Each figure is the result of a particular sequence of probes as `fold` iterates over the constraints of a 30-nation map.

The time performance of BDD related computation can be difficult to predict due to the caching strategy inherent to how BDDs work. When building a large BDD, which can be naively thought of as an exponentially sized binary search tree, many of the branch copies can be circumvented if they already appear in the cache. The algorithm is nevertheless exponential in worst case, but as is discussed in [23], the caching has the effect of lowering the exponent. A reason we hoped `tree-fold` would increase performance is that

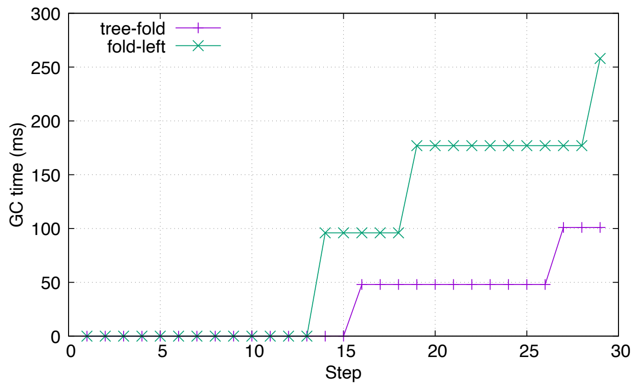


Figure 5.12: Time (milliseconds) spent in GC while coloring a map of Europe. Fewer milliseconds is better than more.

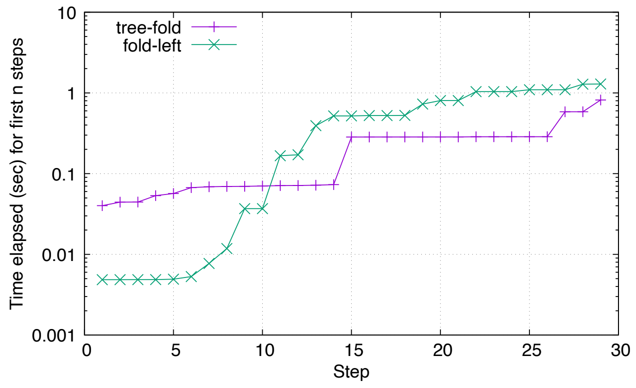


Figure 5.13: Time (seconds) to compute the BDD representing the border constraints of the first n-many countries. Fewer seconds is better than more.

it may decrease the total number of allocations needed for the full computation. Figure 5.11 (top) shows the number of allocations at each step of the iteration, and Figure 5.11 (bottom) shows the accumulated number of BDD nodes allocated during the `fold` iteration. In the experiment illustrated, we see that in general `tree-fold` allocates fewer nodes, $\approx 300K$, compared to the $> 500K$ nodes allocated by `fold-left`. Additionally, this means that most of the `tree-fold` iteration executes with less memory footprint—another reason to hope for better performance.

A second reason we suspected that `tree-fold` would perform better is that the algorithm spends less time in garbage collection (GC), because (as we theorized) creating fewer objects, should decrease the amount of time the GC needs to find stranded objects.[8] The plot in Figure 5.12 confirms this suspicion. The figure shows the total accumulated GC time spent at each iteration of `fold`. We see that in this particular case `tree-fold` spends about 100 milliseconds in GC while `fold-left` spends in excess of 250 ms. A difference of 150 ms is significant when considering a total computation time of 1 second as shown in Figure 5.13.

[8]Thanks to Jasper M, https://users.scala-lang.org/u/jasper-m, for helping us incorporate the `ManagementFactor` class of the `java.lang.management` library into our Scala code. We used this library to probe GC related information from the JVM while our program was running.

Model Name:	MacBook Pro
Processor Name:	Quad-Core Intel Core i7
Processor Speed:	2.7 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	8 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB

Figure 5.14: Hardware used for experiments.

Figure No.	Scala Function
5.1	`rationalSums`
5.4	`floatSums`
5.5	`floatSumsRandom`
5.10	`timeColorGraph`
5.11	`fourColor`
5.12	`fourColor`
5.13	`fourColor`

Figure 5.15: Scala functions to reproduce plots in this article. Functions are found in package `ifl` in Object `Ifl2022`.

5.7 Reproducing our Results

All timing tests mentioned in this article were performed using Scala version 2.13.8, on the hardware described in Figure 5.14.

The code used in the experiments in Section 5 are freely and publicly available on the GitLab server of EPITA: gitlab.lrde.epita.fr. The code is governed by an MIT-style license. To download the code, clone the git repository.[9] The project `regular-type-expression` is a research project whose scope is much larger than what is discussed in this article. The relevant part can be found at the relative path cl-robdd/src/cl-robdd-scala, which is a Scala/sbt project.

The plots in this article may be reproduced using the Scala functions indicated in Figure 5.15.

Each of the functions produces a file with a `.gnu` extension. This file is intended as input to the `gnuplot` program. To produce a graphical plot in PNG format, for example, execute

```
gnuplot -e "set terminal png" file.gnu > file.png
```

6 CONCLUSION

In this article we have looked at two implementations of the `fold` function which agree on their semantics but differ in how they group expressions and which order evaluation occurs. We have looked at several experiments which measure execution time and round-off error of the various approaches. We found that in some cases the approach makes a significant difference and in other cases the difference is less poignant.

[9]git clone https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git-bifl-2022. Commit SHA id 7d2f308e1c4 marks time this article was submitted.

We have investigated computations based on fold—computations whose binary operation degrades in performance due to growth in intermediate values. The tree-based fold implementation outperforms the traditional `fold-left` implementation in some cases. We do not, however, claim that tree-based fold is better in all cases. Rather we suggest that in some cases the programmer needs finer control over the computation order depending on the nature of the computation being performed.

We have not talked about another common `fold` implementation, `fold-right`, which effectively (if not explicitly) reverses the input sequence before folding. There are cases where `fold-right` is known to be efficient, such as end-appending lists.

In an email conversation with Fritz Henglein[10], he suggested that floating point addition of sequences which are themselves growing exponentially are not good candidates for `tree-fold`. Certainly, the applications we tested with used data bounded and uniform in order of magnitude. Henglein reiterated our claim that the version of `fold` which gives the best results heavily depends on the input data. However, we keep in mind that intermediate values, produced by the binary operation in question, themselves become input for the `fold` operation in progress. If this feedback loop produces data which cause degradation in performance/accuracy, the `tree-fold` algorithm tends to avoid so-called feedback interference, by not mingling generated values with input values if it can be avoided.

We believe that researchers should be honest about their results and avoid the temptation to show only positive results. In keeping with this belief we emphasize that some of the results in Section 5.4 are not as convincing that we had hoped. This can perhaps be attributed to the fact that BDDs tend to grow exponentially, while the examples in Section 4 are effected by polynomial growth.

This fact is disappointing as one of the primary motivating factors for starting this research was to improve BDD construction times or at least to characterize which cases can be improved.

7 PERSPECTIVES

7.1 Dynamic folding strategies

In this article we have investigated computations whose *performance* degrades over time due to the progressive growth of intermediate results. Tree folds perform very well for inputs where elements are reasonable uniform, but they can still be a worse choice where inputs are, for example, increasing exponentially.

This could potentially be avoided by selecting a folding strategy based on input data. However the tradeoffs of such an approach are not obvious and warrant further research. Further they bring with them additional complexity which might make it unappealing compared to the simplicity of tree folds.

7.2 Further BDD construction optimizations

We have addressed constructing BDDs only as a sum of products (4), *i.e.*, as DNF. BDDs used in model checking [5] and SAT solving [13] are most often constructed based on a product of sums, referred to as CNF (conjunctive normal form) (17).

[10]Fritz Henglein is a professor of Programming Languages and Systems at University of Copenhagen, and Head of Research at Deon Digital AG

```
// Example usage, returns integer product of sums 216000
sumOfProducts( Seq(Seq(1, 2, 3),
                   Seq(10, 20, 30),
                   Seq(100, 200, 300)))(
  plus = _ * _,  zero = 1,
  times = _ + _, one = 0)

// Example usage, return BDD = AND of ORs of the given BDDs
sumOfProducts( Seq(seq1ofBdds, seq2ofBdds, sea3ofBdds))(
  plus = BddAnd, zero = BddTrue,
  times = BddOr, one = BddFalse)
```

Figure 7.1: Scala example of using the `sum-of-products` function to compute the product of sums, simply by swapping the arguments at the call site.

$$CNF = \prod_{i=1}^{m} \sum \gamma_i = \prod_{i=1}^{m} \sum_{x \in \gamma_i} x \,. \tag{17}$$

The computation necessary to construct a BDD from a CNF form can be done using the code in Figure 2.2, simply by swapping the keyed arguments, as in Figure 7.1. Because of duality, we would expect to get the same performance characteristics using CNF rather than DNF, but admittedly we have not tested this hypothesis.

In this article we have investigated how our BDD construction computations interact with runtime, the heap and garbage collection. Previous work [22] discussed the incorporation of weak-hash-tables into the BDD computation which showed very positive results in our Common Lisp BDD implementation. Unfortunately, similar enhancements to our Scala library do not show analogous performance boosts. On the contrary, we have noticed the using `WeakValueHashMap` from `org.jboss.util` as the weak-hash-table implementation may have a net negative effect as it seems to significantly increases memory usage over all. Our conclusions are not definitive; more research is needed.

Until now, we have observed similar results (not published here) using the Common Lisp language. Both Scala and Common Lisp have strict evaluation orders. It would be interesting to know which if any of our observations depend on this order, and whether normal evaluation order obviates any of our suggested need to user intervention in the associativity of the `fold` operation.

REFERENCES

[1] Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edn. (1996)

[2] Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)

[3] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**, 677–691 (August 1986)

[4] Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. ACM Comput. Surv. **24**(3), 293–318 (Sep 1992). https://doi.org/10.1145/136035.136043, http://doi.acm.org/10.1145/136035.136043

[5] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 1020 States and Beyond. Inf. Comput. **98**(2), 142–170 (Jun 1992). https://doi.org/10.1016/0890-5401(92)90017-A

[6] Chiusano, P., Bjarnason, R.: Functional Programming in Scala. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2014)

[7] D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Computer Aided Verification, 29th International Conference (CAV'17). Springer (July 2017), https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/

[8] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (Jan 2008). https://doi.org/10.1145/1327452.1327492, http://doi.acm.org/10.1145/1327452.1327492

[9] Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 19:1–19:64 (Sep 2008). https://doi.org/10.1145/1391289.1391293, http://doi.acm.org/10.1145/1391289.1391293

[10] Haveraaen, M., Morris, K., Rouson, D., Radhakrishnan, H., Carson, C.: High-Performance Design Patterns for Modern Fortran. Scientific Programming p. 14 (2015)

[11] Heap, M.A., Mercer, M.R.: Least Upper Bounds on OBDD Sizes. IEEE Transactions on Computers **43**, 764–767 (June 1994)

[12] Hickey, R.: The clojure programming language. In: Proceedings of the 2008 symposium on Dynamic languages. p. 1. ACM (2008)

[13] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

[14] Hutton, G.: A tutorial on the universality and expressiveness of fold. J. Funct. Program. **9**(4), 355–372 (Jul 1999). https://doi.org/10.1017/s0956796899003500

[15] Jones, S.P. (ed.): Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (September 2002), http://haskell.org/definition/haskell98-report.pdf

[16] Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)

[17] Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 12th edn. (2009)

[18] Langberg, M., Pnueli, A., Rodeh, Y.: The ROBDD Size of Simple CNF Formulas. In: Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. pp. 363–377 (2003). https://doi.org/10.1007/978-3-540-39724-3_32, https://doi.org/10.1007/978-3-540-39724-3_32

[19] McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press (1962)

[20] McCarthy, J.: History of LISP, p. 173–185. Association for Computing Machinery, New York, NY, USA (1978), https://doi.org/10.1145/800025.1198360

[21] Mollin, R.A.: Fundamental Number Theory with Applications, Second Edition. Chapman and Hall/CRC, 2nd edn. (2008)

[22] Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)

[23] Newton, J., Verna, D.: A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. ACM Transactions on Computational Logic **20**(1), 6:1–6:36 (Jan 2019). https://doi.org/10.1145/3274279

[24] Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The scala language specification (2004)

[25] Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)

[26] Steele, Jr., G.L., Gabriel, R.P.: The Evolution of Lisp. In: The Second ACM SIGPLAN Conference on History of Programming Languages. pp. 231–270. HOPL-II, ACM, New York, NY, USA (1993). https://doi.org/10.1145/154766.155373, http://doi.acm.org/10.1145/154766.155373

[27] Swaine, M.: Functional Programming: a PragPub Anthology: Exploring Clojure, Elixir, Haskell, Scala, and Swift. Pragmatic programmers, Pragmatic Bookshelf (2017), https://books.google.fr/books?id=AMoXMQAACAAJ

[28] Turner, D.: An overview of miranda. SIGPLAN Not. **21**(12), 158–166 (Dec 1986). https://doi.org/10.1145/15042.15053, https://doi.org/10.1145/15042.15053

[29] Turner, D.A.: Some history of functional programming languages. In: Proceedings of the 2012 Conference on Trends in Functional Programming. TFP 2012, vol. 7829, pp. 1–20. Springer-Verlag New York, Inc., New York, NY, USA (2013). https://doi.org/10.1007/978-3-642-40447-4_1

[30] Turner, D.: A new implementation technique for applicative languages. Software Practice and Experience **9**(1), 31–49 (Jan 1979), https://doi.org/10.1002/spe.4380090105

[31] Yorgey, B.A.: Monoids: Theme and variations (functional pearl). SIGPLAN Not. **47**(12), 105–116 (Sep 2012). https://doi.org/10.1145/2430532.2364520, http://doi.acm.org/10.1145/2430532.2364520