Performance Comparison of Several Folding Strategies

Jim Newton[0000−0002−1595−8655]

EPITA Research Lab, 94270 Le Kremlin Becêtre, FRANCE `jnewton@lrde.epita.fr`

Abstract. In this article we examine the computation order and consequent performance of three different conceptual implementations of the `fold` function. We explore a set of performance based experiments on different implementations of this function. In particular, we contrast the `fold-left` implementation with two other implements we refer to as `pair-wise-fold` and `tree-like-fold`. We explore two application areas: ratio arithmetic and Binary Decisions Diagram construction. We demonstrate several cases where the performance of certain algorithms is very different depending on the approach taken. In particular iterative computations where the object size accumulates are good candidates for the `tree-like-fold`.

Keywords: fold · binary decision diagram· scala · lisp · rational numbers

1 Introduction

The higher-order function[1, Sec 1.3], `fold` [14], is present in many modern programming languages. Because of its higher-order nature, the `fold` function was originally conceived for functional-style languages. The earliest mention of `fold` that we have found, was in the SASL language [20].[1] However, 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 [19, 10]. As a few examples, in the Common Lisp language [3] the function is called `REDUCE`; the Scala language [18, 8] offers several variants `foldLeft`, `foldRight` and others. According to Bird [4, p 42], the Haskell language does not provide a general `fold` function, but he demonstrates how one could be defined in three lines.

Although it is not a definitive source of information, we note that the Wikipedia article on `Fold`[2] lists ≈ 44 programming languages which support this feature, sometimes with different names such as `reduce`, `accumulate`,

[1] https://www.quora.com/Where-did-the-common-functional-programming-functions-get-their-names Mark Harrison inlines an email form David Turner (the author of SASL and Miranda) claiming to be the inventor of the `foldr`/`foldl` functions sometime between 1976 and 1983.

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

`aggregate`, `compress`, or `inject`. The count in the Wikipedia article depends on whether you consider Python 2.x and Python 3.x to be different languages, and whether you consider JavaScript and ECMAScript the same or different.

Such a function is useful for extending a binary function to multiple arity, and applying the function to objects in a collection which may be a sequence or some sort of recursive data structure. It is generally supposed that the binary function in question be associative but not necessarily commutative. For example if we assume $f : D \times D \to D$ is a binary function; we suppose that $f(x_1, f(x_2, x_3)) = f(f(x_1, x_2), x_3)$, but we do not suppose that $f(x_1, x_2) = f(x_2, x_1)$.

The notation becomes somewhat cleaner if we denote such a binary function as an operator, $\circ$, written as infix, 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 however we like, as long as we respect the order.

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

While all these groupings result in the same result mathematically, they may have different performance characteristics. In this article we look at three such groupings which we call `fold-left` (Section 3.1), `pair-wise-fold` (Section 3.2), and `tree-fold` (Section 3.3).

2 Motivation

In [17], we (Newton *et al.*) discuss approaches to construct Binary Decision Diagrams (BDDs), in particular construction of the BDD representing a randomly selected Boolean function of n variables. We further noted, in [16], some *unexpected* performance results and suggested they be further investigated.

The principle problem is to construct a BDD given a Boolean formula. In our case the Boolean formula is given in a sum of products form such as:

$$x_1 x_2 \overline{x_3} \ + \ \overline{x_1} x_2 x_3 \ + \ x_1 \overline{x_2} x_3 \overline{x_4} \, .$$

In order to represent such a Boolean function programmatically, we suppose that Γ is a finite set of variables and their complements such as

$$\Gamma = \{x_1,\ \overline{x_1},\ x_2,\ \overline{x_2},\ ...,\ x_n,\ \overline{x_n}\} \, .$$

Definition 1. *A subset, γ, of Γ is called* Γ-contradictory *(or simply* contradictory*) if $\{x_i, \overline{x_i}\} \subset \gamma$ for some $1 \le i \le n$. On the contrary, a subset of Γ which is not Γ-contradictory is called* Γ-consistent *(or simply* consistent*).*

Now suppose for $S = \{\gamma_1, \gamma_2, ..., \gamma_m\}$ is a set of consistent subsets of Γ. With this notation we wish to consider a Boolean formula such as:

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

This sum of products form is sometimes referred to as DNF (disjunctive normal form).

Programmatically, this sum of products is computed as two concentric `fold` operations. In Figure 1, we show two implementations of the `sum-of-products` function, first using Scala[3] and second using Common Lisp. In each case we assume the existence of a binary function `BddAnd` along with its neutral element `BddTrue` which performs the Boolean intersection operation between to 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.

3 Strategies for Computing a Fold Operation

In Section 4 we will investigate operations on BDDs as alluded to in Section 2. But rather doing so directly, instead we have first devised experiments based on rational number arithmetic. 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 adding a large sequence of fractions, where each fraction is expressed as the ratio of two integers as ratios.

$$\frac{1}{23}+\frac{1}{29}+\frac{1}{31}+\frac{1}{37}+\frac{1}{41}+\frac{1}{43}+\frac{1}{47}+\frac{1}{53}+\frac{1}{57}+\frac{1}{67}=\frac{3,304,092,302,051,372}{12,831,131,327,329,923} \tag{2}$$

Computing the sum in Equation (2) involves representing the numerator and denominators as a `bignum` integer type. [15, section 4.5] Integers in Common Lisp are specified to have unlimited precision, and the built-in `ratio` type provides precise fractions whose numerator and denominator never *roll-over*. However in Scala, Integers do not have this feature; thus the programmer must use a non-native type, such as the `Rational` type provided by `import spire.math.Rational`.

[3] 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.

```scala
// Scala-like coding style
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)
```

```lisp
;; Lisp-like coding style
(defun sum-of-products (seq &key + 0+ * 1*)
  (reduce (lambda (acc gamma)
            (funcall + acc
                     (reduce * gamma :initial-value 1*)))
          seq
          :initial-value 0+))

;; example usage, returns integer sum of products 6006006
(sum-of-products (list (list 1 2 3)
                       (list 10 20 30)
                       (list 100 200, 300))
   :+ #'+   :0+ 0
   :* #'*   :1* 1)

;; example usage,
;;   returns BDD which is an OR of ANDS of the given BDDs
(sum-of-products (list list-1-of-bdds
                       list-2-of-bdds
                       list-3-of-bdds)
   :+ #'BddOr    :0+ BddFalse
   :* #'BddAnd   :1* BddTrue)
```

Fig. 1. Scala and Lisp implementations of `sum-of-products` and usage examples.

Regardless of which programming language and which implementation of `ratio` is used, each such addition operation must compute some variant of

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \cdot d_2 + n_2 \cdot d_1}{d_1 \cdot d_2} . \tag{3}$$

$$= \frac{1}{gcd(d_1, d_2)} \cdot \left(\frac{n_1}{\frac{d_1}{gcd(d_1,d_2)}} + \frac{n_2}{\frac{d_2}{gcd(d_1,d_2)}} \right) . \tag{4}$$

Each of these operations is a `bignum` computation, followed either immediately or eventually by a fraction simplification—dividing the numerator and denominator by their common factors. Of course there are several strategies to optimize such a computation. For example if the greatest common divisor, $gcd(d_1, d_2)$ is known to be different than 1, then the sum can be computed as in Equation (4). According to Theorem 1, Equation (4) can be computed by an application of Equation (3), but involving smaller numbers, in $\approx 40\%$ of the cases.

Theorem 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 $6/\pi^2 \approx 60.793\%$.*

A statement and proof of Dirchlet's theorem are provided as Theorem D in Section 4.5.2 of Knuth's *Art of Computer Programming* [15, page 342].

If $gcd(d_1, d_2) \neq 1$, Knuth [15, page 330] suggests the following to calculate n_3 and d_3 such that $\frac{n_3}{d_3} = \frac{n_1}{d_1} + \frac{n_2}{d_2}$.

$$\begin{aligned} g_1 &= gcd(d_1, d_2) \\ t &= n_1 \cdot (d_2/g_1) + n_2 \cdot (d_1/g_1) \\ g_1 &= gcd(t, g_1) \\ n_3 &= t/g_3 \\ d_3 &= (d_1/g_1) \cdot (d_2/g_2) \end{aligned}$$

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

3.1 Default `fold-left`

The default version of `fold-left` always first computes the result of $x_1 \circ x_2 \circ ... \circ x_{i-1}$ before combining that result with x_i.

The `fold-left` operation groups these addition operations as follows:

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

Table 1. Intermediate find final values of added 10 ratios using default `fold-left` algorithm, computed as shown in Equation (5).

$$\underbrace{(((((\underbrace{(\underbrace{(\underbrace{(\frac{1}{23}+\frac{1}{29})}_{\#\ 1}+\frac{1}{31})}_{\#\ 2}+\frac{1}{37})}_{\#\ 3\ \ldots}+\frac{1}{41})+\frac{1}{43})+\frac{1}{47})+\frac{1}{53})+\frac{1}{57})+\frac{1}{67}}_{\ldots\ computation\ \#\ 9} \qquad (5)$$

Such a computation must compute eight intermediate values before arriving at the final value. Table 3.1 indicates these eight intermediate values as computations # 1 through # 8 before arriving at the final value as a result of computation # 9. The table also indicates two columns, labeled **Digits Computed** and **Digits Retained**. The number of digits computed is 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 5 (top). We compare the cumulative number of digits also for the analogous experiments which follow in Sections 3.2 and 3.3. The number of digits retained (the bottom-most column in the table) indicates 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**) because it is conceivable that they might have an effect on the computation time. I.e., we suppose that the *gcd* computations which are calculated to perform the rational number additions is 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.

When we observe the **Digits Computed** column of Table 3.1 and the curve in Figure 5 (top) corresponding to `fold-left`, we see that this algorithm computes more digits than the others. In terms of number of digits computed, it is the worst of the three alternatives. However, when we observe the **Digits Retained** column of Table 3.1 and the corresponding curve in Figure 5 we see that it retains the least amount of heap storage.

3.2 Pair-wise `fold`

```scala
def pairWiseFold[A](z: A)(mList: List[A], f: (A, A) => A): A = {
  val (pairs: List[(A, A)], leftover: Option[A]) = paired(mList)
  if (mList.isEmpty)
    z
  else {
    @scala.annotation.tailrec
    def recur(li: List[(A, A)], maybeB: Option[A]): A = {
      val reduced: List[A] = li.map { case (b1: A, b2: A) => f(b1, b2) }
      if (reduced.tail.isEmpty)
        maybeB match {
          case None => reduced.head
          case Some(b) => f(reduced.head, b)
        }
      else {
        val (pairs: List[(A, A)], leftover: Option[A]) = paired(reduced)
        val last: Option[A] = (leftover, maybeB) match {
          case (Some(b1), Some(b2)) => Some(f(b1, b2))
          case (None, Some(_)) => maybeB
          case (Some(_), None) => leftover
          case (None, None) => None
        }
        recur(pairs, last)
    }}
    recur(pairs, leftover)
}}
```

Fig. 2. Scala implementation of `pair-wise-fold`.

The default `fold-left` algorithm, discussed in Section 3.1, groups terms toward the left, as shown in Equation (5). As an alternative, we might instead compute the sum in Equation (2) by first grouping consecutive terms, computing those intermediate results, and repeating the process. Each such iteration involves roughly half as many applications of the binary function as the previous iteration. In the case of ratio arithmetic, even though each iteration of the algorithm performs half as many additions as the previous iteration, these additions involve larger numbers with each successive iteration.

This type of grouping corresponds to inserting parentheses as shown in Equation (6).

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

Just as in Section 3.1, this computation again involves eight intermediate results. However, we notice that these intermediate results tend to be smaller than those resulting from the default `fold-left` approach.

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	$\frac{1}{41}+\frac{1}{43}$	$=\frac{84}{1763}$	$+6=17$	17
# 4	$\frac{1}{47}+\frac{1}{53}$	$=\frac{100}{2491}$	$+7=24$	24
# 5	$\frac{1}{57}+\frac{1}{67}$	$=\frac{124}{3819}$	$+7=31$	31
# 6	$\frac{52}{667}+\frac{68}{1147}$	$=\frac{105{,}000}{765{,}049}$	$+12=43$	32
# 7	$\frac{84}{1763}+\frac{100}{2491}$	$=\frac{385{,}544}{4{,}391{,}633}$	$+13=56$	32
# 8	$\frac{105{,}000}{765{,}049}+\frac{385{,}544}{4{,}391{,}633}$	$=\frac{756{,}081{,}516{,}656}{3{,}359{,}814{,}435{,}017}$	$+25=81$	31
# 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

Table 2. Intermediate and final values of added 10 ratios using default `pair-wise-fold` algorithm, computed as shown in Equation (6).

We observe two additional features of the `pair-wise-fold` approach. The first observation, which we consider an advantage, is that the computation is potentially parallelizable. I.e., computations # 1, # 2, # 3, and # 4 could be done in parallel. Of course we do not suspect that such parallelization would be of significant benefit for this small example. There are also problematic cases, such as the BDD examples which are the prime motivating factors for our research. Even though the binary operations of `AND` and `OR` are associative, they are not parallelizable. In the case of BDDs, many implementations [2] prohibit parallelized construction. Some work has been done to lift this restriction [7, 13, 9].

The second observation, which we consider a disadvantage, is that in a naïve implementation of this algorithm, intermediate results must be stored in memory until they are used. *I.e.*, results # 1, # 2, # 3, and # 4, must be stored in memory until computations # 4 and # 6 are performed. It is generally assumed that such memory retaining is not an issue, but in the case of huge data structures such as BDDs, we explicitly do not assume such memory retaining is free.

When we observe the **Digits Computed** column of Table 3.2 and the curve in Figure 5 (top) corresponding to `pair-wise-fold`, we see that this algorithm computes more digits than `fold-left` and more than `tree-like-fold`. When we observe the **Digits Retained** column of Table 3.2 and the corresponding curve in Figure 5 we see similar results—the data and the curve corresponding to is more positive than the other two thus we conclude that at least for this example, `pair-wise-fold` is the most heap-stressing of the alternatives.

```
def paired[A](data: List[A]): (List[(A, A)], Option[A]) = {
  val none: Option[A] = None
  val nil: List[(A, A)] = Nil
  data.foldLeft((nil, none)) {
    case ((stack: List[(A, A)], Some(b)), a) => (((b, a) :: stack), none)
    case ((stack: List[(A, A)], None), a) => (stack, Some(a))
  } match {
    case (stack, leftover) => (stack.reverse, leftover)
  }
}
```

Fig. 3. Scala implementation of the `paired` function

The Scala implementation of `pair-wise-fold` is shown in Figure 3.2. This implementation uses a helper function `paired` whose implementation is shown in Figure 3. The `paired` function, takes a `List` of elements of type `A`, and returns a `List` of pairs of the same type, following by an `Option[A]`. If the given list has even length, the `Option` is empty; otherwise the `Option` contains the final (odd) element of the `List`.

This function is used in the function `pairWiseFold` whose task it is to apply the given function on consecutive pairs, but must *remember* the left-over element for the next iteration in the case that exact pairing is impossible.

3.3 Tree-like `fold`

The `tree-like-fold` described here alleviates one of disadvantages of the `pair-wise-fold` as described in Section 3.2—namely rather than retaining intermediate values, the `tree-like-fold` consumes the values as soon as possible while still respecting the same grouping. The parenthesized grouping of the `tree-like-fold` shown in Equation (7) is exactly the same as Equation 6. However, the order which computations are performed is different.

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

Both the `pair-wise-fold` and the `tree-like-fold` coincide about computations # 1 and # 2. However, whereas `pair-wise-fold` retains these two intermediate values while performing computations # 3 and # 4, `tree-like-fold` consumes # 1 and # 2, immediately as computation # 3. Admittedly, 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. Whereas `pair-wise-fold` must retain $\frac{n}{2}$ intermediate results (n being total length of the sequence being combined in Equation (2)), `tree-like-fold` must retain at most $\log_2 n$.

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

Table 3. Intermediate find final values of added 10 ratios using default `tree-like-fold` algorithm, computed as shown in Equation (7).

As in Sections 3.1 and 3.2, once again we look at the number of digits computed and retained by `tree-like-fold`. When we observe the **Dig-**

its **Computed** column of Table 3.3 and the curve in Figure 5 (top) corresponding to `tree-like-fold`, we see at computation # 9, that this algorithm computes the exact same number of digits as `tree-like-fold` but fewer than `fold-left`. In fact the actual computations performed by `tree-like-fold` and `pair-wise-fold` are exactly the same, but the order is different. In terms of number of digits retained, `tree-like-fold` again falls in between the other two implementations.

A recognizable advantage of `tree-like-fold` over `pair-wise-fold` is that the code in its implementation is much more concise, as shown in the listing in Figure 3.3.

```scala
def treeFold[A](m: List[A])(z: A)(f: (A, A) => A): A = {
  def consumeStack(stack: List[(Int, A)]): List[(Int, A)] = {
    stack match {
      case (i, b1) :: (j, b2) :: tail if i == j => consumeStack((i + 1,
          f(b2, b1)) :: tail)
      case _ => stack
  }}
  val stack = m.foldLeft((1, z) :: Nil) { (stack: List[(Int, A)], ob: A)
      =>
    consumeStack((1, ob) :: stack)
  }
  stack.map(_._2).reduce { (a1: A, a2: A) => f(a2, a1) }
}
```

Fig. 4. Scala implementation of the `tree-like-fold` algorithm.

3.4 Summarizing the three `fold` strategies

The tree sequences of computations outlined in Sections 3.1, 3.2, and 3.3 are recapped in Figure 5. We see that in the end `fold-left` computes more digits than either `pair-wise-fold` or `tree-like-fold`, and we see that in the end (at computation # 9) `pair-wise-fold` and `tree-like-fold` have computed the exact same number of digits. This latter fact is not surprising, as the computations are determined by the parentheses which are the same in Equations (6) and (7). In the case of adding up these particular ratios, it happens that `tree-like-fold` computes the digits more greedily than does `pair-wise-fold`. The ratios in question (Equation (2)) have been especially chosen to be a worst case in some sense. 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 1, 40% of the time, the gcd will be different than 1, so we suspect cases exist for which the plots of `tree-like-fold` and `pair-wise-fold` might be oriented differently.

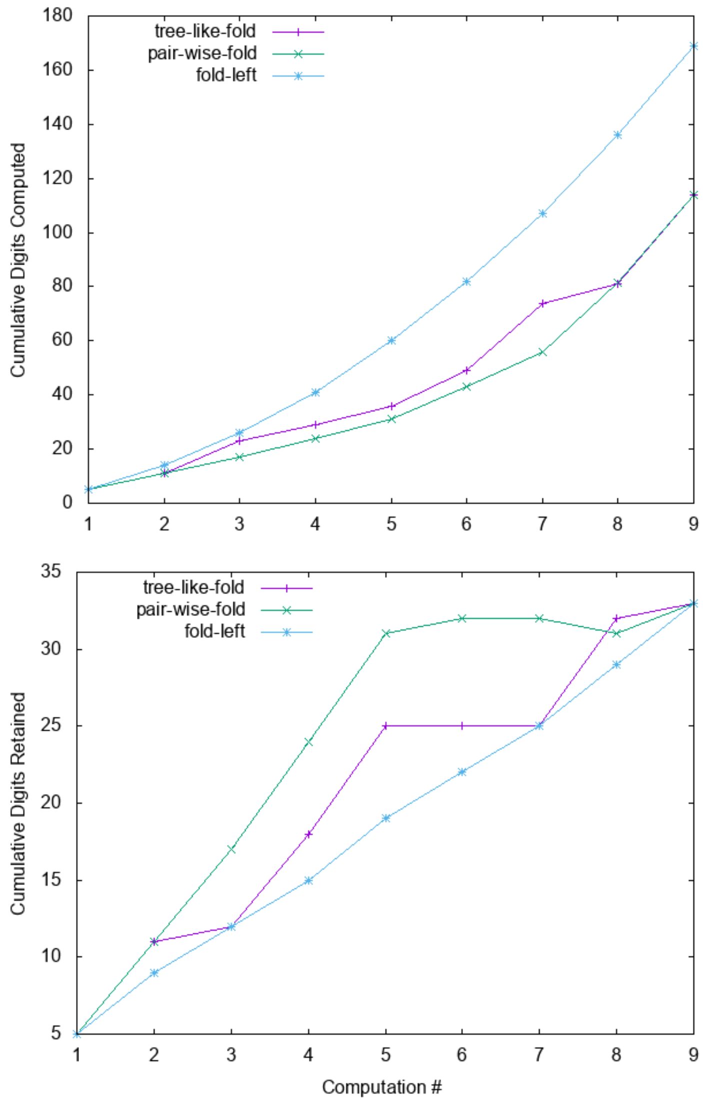

Fig. 5. (Top) Cumulative Digits Computed For Each Fold Strategy. (Bottom) Quantity of Digits Retained at each Computation State

In summary of the bottom plot in Figure 5, all three algorithms retain exactly the same number of digits at computation # 9. This is obvious because the only thing the retain is the 33 digits of the final result $\frac{3,304,092,302,051,372}{12,831,131,327,329,923}$. However, it appears, at least for the single example computation, that `tree-like-fold` is a happy medium between `fold-left` and `pair-wise-fold` as far as greedy release of heap allocation is concerned.

4 Experimental Results

First in Section 4.1, we examine the computation time results of the three `fold` algorithms, first when applied to ratio additions as explained in Sections 3.1, 3.2, and 3.3. Thereafter, in Sections 4.2, 4.3, and 4.4 we examine the results when we apply the same techniques to BDD construction.

All timing tests mentioned in the following sections were performed using Scala version 2019.2.36, running within IntelliJ IDEA 2019.2.4 (Ultimate Edition), on the same computer (with the hardware overview shown below).

Hardware Overview	
Model Name:	MacBook Pro
Operating System:	macOS Catalina
Version:	10.15.1
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

4.1 Results of Ratio Addition

The first experiment we performed entailed summing a sequence of rational numbers of incrementally increasing length. The plot in Figure 6 shows the computation time, in milliseconds, of computing sums of different length sequences, using three different folding algorithms. The x-axis indicates the value of n and the y-axis indicates the time needed to compute the sum

$$\sum_{-n\leq\ i\ \leq -1} \frac{1}{i} + \sum_{1\leq\ i\ \leq n} \frac{1}{i} = 0$$

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 three different ways as outlined in Sections 3.1, 3.2, and 3.3. It is fairly clear from Figure 6, that `tree-like-fold` and `pair-wise-fold` perform better especially as the value of n grows, particularly for values of $n > 100$. Moreover, the benefit gained from the `tree-like-fold` and `pair-wise-fold` algorithms increases as measured by the gap between the `fold-left` and the other two curves. This gap widens as n increases. We also see a strange effect regarding the left-most points of each curve, which we attribute to the so-called *warm-up time* which is a well known consideration when for JVM-base (Java Virtual Machine) benchmarking.[4]

[4] `https://stackoverflow.com/questions/36198278/why-does-the-jvm-require-warmup` JVM warm-up-time discussed in `stackoverflow` with regard to benchmarking.

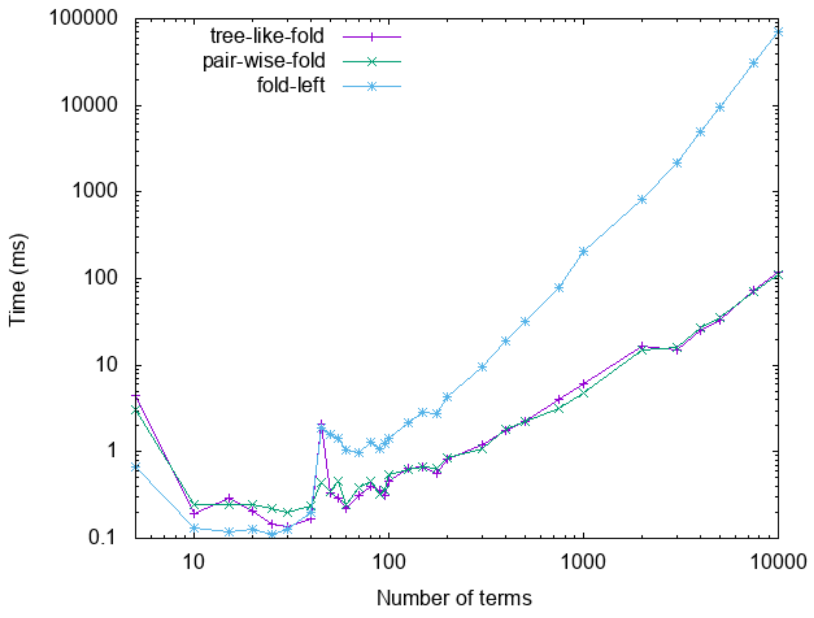

Fig. 6. Performance of Fold Strategy on Rational Addition

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

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

4.2 Results of Constructing Random BDDs

The plot in Figure 7 shows the computation time in milliseconds of constructing random BDDs of increasingly many Boolean variables. This construction is known to have exponential complexity [5, 11]. For a given number of Boolean variables, n, the truth table has 2^n rows. Each row which has `true` in the function value column corresponds to a minterm of the underlying Boolean function. *I.e.*, the set of Boolean functions of n variables is isomorphic to the set of subsets of the set of all minterms. There are 2^{2^n} such subsets, thus as many Boolean functions, a thus as many n-variable BDDs.

How do we generate a random BDD? To choose a random BDD, we effectively fill out this truth table by forming subsets containing minterms; *i.e.*, with balanced distribution include or exclude a given minterm. This selection process is equivalent to choosing a random integer j between 0 and $2^{2^n} - 1$ (inclusive), and then selecting all the minterms, k, for which the k^{th} bit of j (in base-2) is 1.

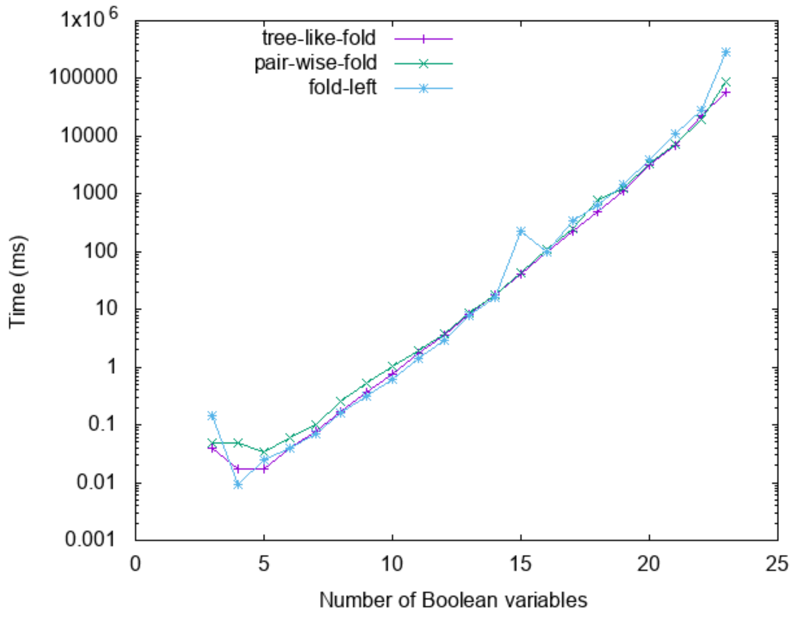

Fig. 7. Performance of Fold Strategy on Construction of Random BDD on n variables. We observe that starting at about $n = 20$ the execution of `pair-wise-fold` and `tree-fold` are significantly faster than that of default `fold-left`.

For performance and memory reasons, it is not necessary to compute j explicitly. It suffices to iterate through a sequence of 2^n random bits, generated lazily.

On the average a randomly selected DNF (disjunctive normal form) 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.

From the plot in Figure 7, we observe that for BDD sizes of 20 Boolean variables or less, there is no clear winner among the three algorithms. This is the point in the computation where each DNF has approximately 0.525×10^6 or half a million minterms each containing as many literals. At this point the inner `fold` iterates $(0.525 \times 10^6)^2 = 275 \times 10^9$ times, or 275 trillion.

As we saw before, in Figure 6, there seems to be some initial JVM warm-up time for the first point in each curve. Starting at about 20 Boolean, we see a trend in Figure 7, that the two algorithms `pair-wise-fold` and `tree-like-fold` are both orders of magnitude faster than the algorithm based on `fold-left`. The gap between the curves is not as striking as was the case in Figure 6. The gap is, nevertheless, significant if we recall that the plot is using log-scale on the y-axis. For example to construct a 23-variable BDD, the `fold-left` algorithm requires 284 seconds while the `tree-like-fold` algorithm requires only 62 seconds.

The gap is difficult to see with the eyes because the curves are drawn very close together in the plot. We eliminate this difficulty in the Section 4.4 by normalizing the curves with respect to the timing of `fold-left`.

4.3 Results of Fixing the Term Length

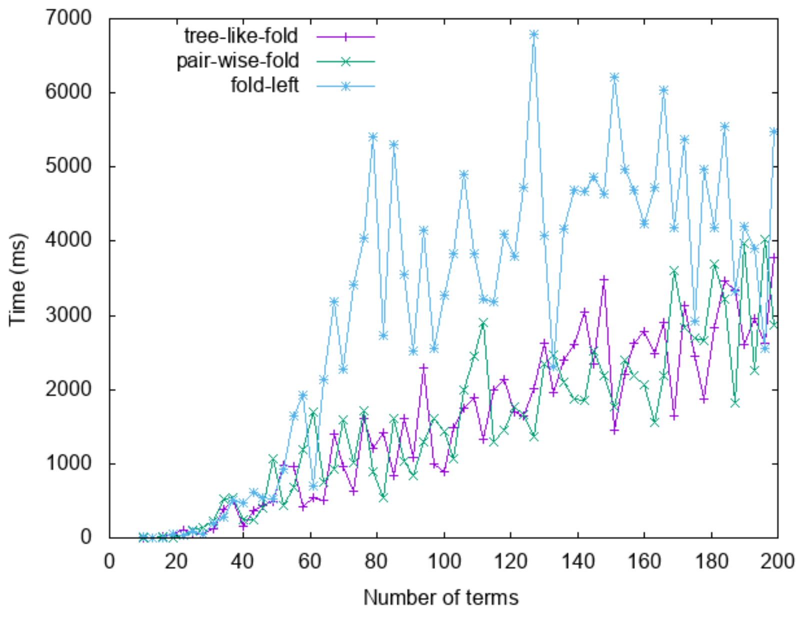

Fig. 8. Performance of Fold Strategy on Construction of BDDs. Each BDD is constructed as an OR of ANDs, where each AND-term is limited to exactly 4 literals.

The next experiment, whose results are shown in Figure 8, is similar to the experiment in Section 4.2, but we limit our samples in the number of terms and the size of each term. We fix the number of Boolean variables to $n = 30$, and randomly select terms having exactly 4 literals. *I.e.*, we consider $\Gamma = \{x_1,\ \overline{x_1},\ x_2,\ \overline{x_2},\ ...,\ x_{30},\ \overline{x_{30}}\}$ (See Definition 1) and limit ourselves to Γ-consistent subsets of size 4, such a subset being $\{x_{10}, \overline{x_{13}}, \overline{x_{19}}, x_{25}\}$. During this computation, assuming that m is the number of terms, the inner `fold` iterates $(m-1) * (4-1) = 3m - 3$ times, and the outer `fold` iterates $m - 1$ times.

The plot in Figure 8 shows BDD construction times (in milliseconds) for a range of number of terms going from 10 terms to 200 terms. The results are somewhat surprising. Although we observe a high degree of noise in the curves, we can observe a tendency between about 50 and 180 terms where the `fold-left` algorithm is slower than other others by a factor of 2 to 3.

4.4 Results of Varying Term Length

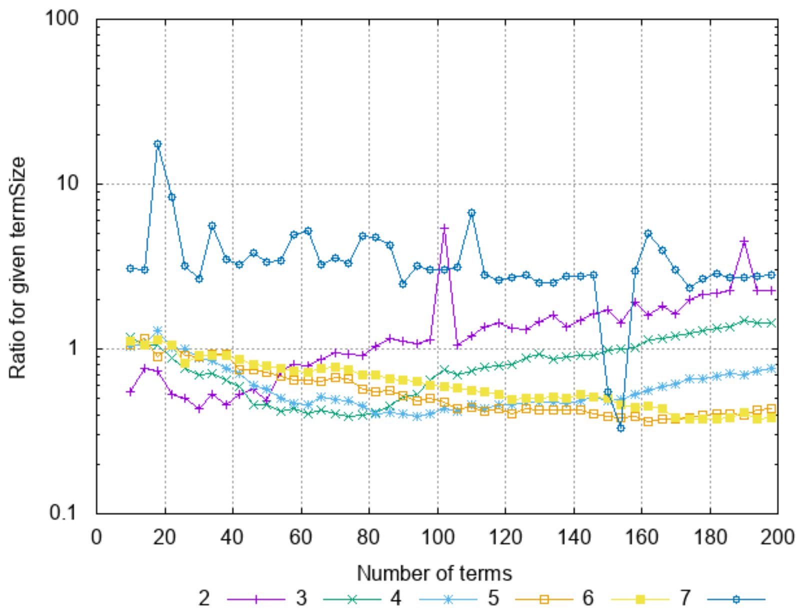

Fig. 9. Performance of Fold Strategy on Construction of BDD as function of Term Length. Smaller values of the ratio indicate better performance than default `foldLeft`. A value of 1 indicates performance equivalent to `foldLeft`.

In this experiment, whose results are plotted in Figure 9, we hold the number of Boolean variables constant at $n = 30$, and test construction of BDDs of varying number of terms, from 10 to 200, and repeat this process while varying the term size from 2 to 7. We only tested this using `fold-left` and `tree-like-fold` and plotted the normalized, relative time consumption.

$$relative\ time = \frac{time\ of\ \text{tree-like-fold}}{time\ of\ \text{fold-left}}\ .$$

Where a curve lies below $y = 1$, is a region where `tree-like-fold` is faster (fewer milliseconds) than `fold-left`. We see that this is more often than not the case for term sizes of 4, 5, and 6. Notably, and curiously, for term size of 7, `tree-like-fold` performs much worse than `fold-left`.

4.5 Reproducing our Results

The code used to construct the experiments discussed in Section 4 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[5] 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 the figures in Sections 4.1, 4.2, 4.3, and 4.4 may be reproduced using the Scala functions indicated in Table 4. Each function is provided in two forms: a 0-ary form which uses default arguments, and an n-ary form for which you may specify custom values.

Figure No.	**Package.Object path**	**Scala Function**
Figure 6	`treereduce.RationalFoldTest`	`rationalFoldTest`
Figure 7	`bdd.ReducePerf`	`testRandomBddConstruction`
Figure 8	`bdd.ReducePerf`	`testLimitedBddConstruction`
Figure 9	`bdd.ReducePerf`	`testNumBitsConstruction`

Table 4. Scala functions to reproduce the plots in Section 4. The functions may be called with various arguments limiting the bounds of the timing tests.

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
```

In addition to the Scala code for reproducing the results, sample data is available in several formats including `.gnu`, `.png`, and `.csv`. These data files can be found relative to the top of the git repository in cl-robdd/data/fold-performance.

5 Conclusions

In this article we have looked at three implementations of the `fold` function which agree on their semantics but differ as far as how they group expressions and which order evaluation occurs. We have looked at several experiments which measure execution time of the various approaches. We found that in some cases the approach makes a big difference and in other cases it does not.

We were able to demonstrate computations of rational numbers (ratios of integers) where a `tree-like-fold` or `pair-wise-fold` unambiguously outperforms the linear `fold-left`. However, as far as the motivating example, BDD construction, is concerned, we have not demonstrated consistent results. There are some situations where a linear fold out-performs a tree-fold, and vise versa.

[5] git clone https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git -b fold-strategy. Commit SHA id 4f68cd7e5 marks time this article was submitted.

There is some evidence that for DNF formulations of certain range of term lengths, a tree-fold is superior, but our findings are not conclusive. More work is needed to characterize definitive predictions or even rule-of-thumb advise for potential users.

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

```
;; example usage, returns integer product of sums 216000
(sum-of-products (list (list 1 2 3)
                       (list 10 20 30)
                       (list 100 200, 300))
   :+ #'*   :0+ 1
   :* #'+   :1* 0)

;; example usage, calling the sum-of-products to compute the
;; product of sums, returning an AND of ORs
(sum-of-products (list list-1-of-bdds
                       list-2-of-bdds
                       list-3-of-bdds)
   :+ #'BddAnd   :0+ BddTrue
   :* #'BddOr    :1* BddFalse)
```

Fig. 10. Scala and Lisp examples of using the `sum-of-products` function to compute the product of sums, simply by swapping the arguments at the call site.

6 Perspectives

Our experiments on BDDs have been based on randomly generated samples, albeit with certain constraints which effect the distribution. There is reason to doubt whether real-world problems can be well modeled by random sampling. For example, in real-world problems, the Boolean variables have correlations which we have not attempted to mimic. We plan to test our process on some large examples of BDD construction which are more consistent with reality.

In this article we have addressed constructing BDDs only as a sum of products (Equation (1)), which we referred to earlier as DNF (disjunctive normal form).

BDDs used in model checking [6] and SAT solving [12] are most often constructed based on a product of sums, referred to as CNF (conjunctive normal form). Such as:

$$CNF = \prod_{i=1}^{m} \sum \gamma_i = \prod_{i=1}^{m} \sum_{x \in \gamma_i} x\,. \tag{8}$$

The computation necessary to construct a BDD from a CNF form can be done using the code in Figure 1, simply by swapping the keyed arguments, as in Figure 10. We would hope to get the same performance characteristics using CNF rather than DNF, but admittedly we have not tested this hypothesis.

References

1. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edn. (1996)
2. Andersen, H.R.: An introduction to binary decision diagrams. Tech. rep., Course Notes on the WWW (1999)
3. Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)
4. Bird, R.: Pearls of Functional Algorithm Design. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
5. 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
6. 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
7. Castagna, G.: Covariance and Contravariance: a fresh look at an old issue. Tech. rep., CNRS (2016), https://arxiv.org/pdf/1809.01427.pdf
8. Chiusano, P., Bjarnason, R.: Functional Programming in Scala. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2014)
9. van Dijk, T., van de Pol, J.: Sylvan: Multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 677–691. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
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. 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)
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. ACM Trans. Program. Lang. Syst. **27**(1), 46–90 (Jan 2005). https://doi.org/10.1145/1053468.1053470

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. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
16. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
17. Newton, J., Verna, D.: A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. ACM Trans. Comput. Logic **20**(1), 6:1–6:36 (Jan 2019). https://doi.org/10.1145/3274279
18. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The scala language specification (2004)
19. 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
20. Turner, D.A.: Some history of functional programming languages. In: Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829. pp. 1–20. TFP 2012, Springer-Verlag New York, Inc., New York, NY, USA (2013). https://doi.org/10.1007/978-3-642-40447-4_1