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 traditional `fold-left` implementation with two other approaches we refer to as `pair-wise-fold` and `tree-fold`. It is often implicitly supposed that the binary operation in question has constant complexity. We explore two application areas which diverge from that assumption: ratio arithmetic and Binary Decisions Diagram construction. These are binary operations which degrade in performance as the computation progresses. We show that these types of binary operations are good candidates for `tree-fold`.

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

1 Introduction

The higher-order function[1, Sec 1.3], `fold` [15], is present in many programming languages. Definition 1 shows the essence of the function including the features which we use in this article.

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

$$fold\big(f, V_{[1,n]}\big) = \begin{cases} x_1 & \textit{if } n = 1 & \text{(1a)} \\ f\big(fold\big(f, V_{[1,n-1]}\big), x_n\big) & \textit{otherwise} & \text{(1b)} \end{cases}$$

If D is a monoïd [36], it is customary to define `fold` for the empty sequence to be the neutral element of the D. We avoid this degenerate case as we have no need for it in this article.

The `fold` 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. Note that it is generally not supposed that the binary function in question be commutative.

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 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). The first grouping, `fold-left`, implements the *standard* algorithm used in most programming language implementations, and implements Definition 1 by following the computation directly as described in Equations (1b) and (2). The second two groupings, `pair-wise-fold` and `tree-fold`, implement Definition 1 by grouping pairs such as in Equation (3), evaluating those pairs to obtain another sequence of values, to which Equation (3) is applied again and again until arriving at a final value. The difference between `pair-wise-fold` and `tree-fold` is that the parenthesized terms are resolved in different order.

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.

In [23], we presented a technique for reasoning about the types of heterogeneous sequences in Common Lisp. We modeled sequences of types with deterministic symbolic finite automata [8], *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 [10], except that we are dealing with a dynamically typed language rather than a statically typed one.

The fact that values in Common Lisp are typed dynamically means that certain type-based decisions can be made at compile time, while others must be delayed until run-time. We use BDDs to represent these 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.

An important consideration for the programmer in implementing these type-based computations is to understand the time and memory requirements for constructing BDDs [27]. In [23, Sec 5.5] we reported that the particular implementation of `fold` which was used in BDD construction, sometimes made an *unexpected* difference in construction time. We further remarked in that this unexpected behavior deserves further systematic study. In the current article, we analyze this effect in the context of BDD construction, and also in the context of rational arithmetic. For additional details we recommend [24–28].

In [23] 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 [29, 7] 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 [23] we constructed BDDs from a Boolean formula, usually given in a DNF (sum of products) form such as: $x_1 x_2 \overline{x_3} + \overline{x_1} x_2 x_3 + x_1 \overline{x_2} x_3 \overline{x_4}$. As Bryant *et al.* [4, 12] explain, such BDD construction can be exponential in complexity. To gather heuristics about time and space complexity ob BDD construction in practice, in [27] 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
$\Gamma = \{x_1,\ \overline{x_1},\ x_2,\ \overline{x_2},\ ...,\ x_n,\ \overline{x_n}\}$.

Definition 2. *A subset, γ, of Γ is called* Γ-contradictory *(or simply* contradictory*) if $\{x_i, \overline{x_i}\} \subset \gamma$ for some $1 \leq i \leq n$. On the contrary, a subset of Γ which is not Γ-contradictory is called* Γ-consistent *(or simply* consistent*).*

Suppose $S = \{\gamma_1, \gamma_2, ..., \gamma_m\}$ is a set of consistent subsets of Γ. We wish to consider a Boolean formula in DNF (disjunctive normal form) such as:

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

Programmatically, this sum of products is computed as two concentric `fold` operations,[1] as shown in Figure 1. 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.

[1] 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)
```

Fig. 1: Scala implementation of `sum-of-products` and usage examples.

This close connection between the `fold` operations and BDD operations motivated the investigation leading to this article. During that research we noticed that changing the association (moving the parentheses) of Boolean functions *sometimes* had a noticeable effect of construction times. Unfortunately, we could not infer any practical characterization of this phenomenon. We noted in [23] that more research was needed. The BDD construction computations in Section 4.2, 4.3, and 4.4 are first steps in systematically investigating these effects.

3 Strategies for Computing a Fold Operation

In Section 4 we will investigate operations on BDDs as alluded to in Section 2. Rather doing so straightaway, instead we have first devised experiments based on arithmetic of rational 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 Equation (5) involves representing the numerator and denominators as a `bignum` integer type. [17, Sec 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`.

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

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

Each of these operations is a `bignum` computation, including explicit or implicit fraction simplification—dividing the numerator and denominator by their common factors. 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 (7). If $g = gcd(d_1, d_2)$, then we can precompute $d_3 = \frac{d_1}{g}$ and $d_4 = \frac{d_2}{g}$, which are integers. The quotient can thus be rewritten as

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

According to Theorem 1, Equation (6) can be computed by an application of Equation (7), 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 proof of Dirchlet's theorem is provided as Theorem D in Section 4.5.2 of Knuth's *Art of Computer Programming* [17, page 342]. Finally, if $gcd(d_1, d_2) \neq 1$, Knuth [17, 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 Equation (7) in the case numerator and denominator of Equation (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 compute intensive as the denominators grow. *E.g.*, it is easier to add $\frac{1}{2} + \frac{2}{3}$ than to add $\frac{105{,}000}{765{,}049} + \frac{385{,}544}{4{,}391{,}633}$.

Mollin [22] 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.

3.1 Strategy using `fold-left`

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 and final values of added 10 ratios using default `fold-left` algorithm, computed as shown in Equation (10).

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

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

Such a computation computes eight intermediate values before arriving at the final value. Table 1 indicates these eight intermediate values as computations # 1 through # 8 before arriving at the 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 3 (top). We compare the cumulative number of digits also for the analogous experiments which follow in Sections 3.2 and 3.3. 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 might have an effect on the computation time. I.e., we suppose 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.

3.2 Strategy using `pair-wise-fold`

The default `fold-left` algorithm, discussed in Section 3.1, groups terms toward the left, as shown in Equation (10). As an alternative, we might instead compute the sum in Equation (5) 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.

Such grouping corresponds to inserting parentheses as in Equation (11).

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

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.

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. The binary operations, `AND` and `OR`, even though associative, are *not* parallelizable. In the case of BDDs, many implementations [2] prohibit parallelized construction. Some work has been done to lift this restriction [6, 14, 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 # 6 and # 7 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.

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 in Equation (11).

3.3 Strategy using `tree-fold`

The `tree-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-fold` consumes the values as soon as possible while still respecting the same grouping. The parenthesized grouping of the `tree-fold` shown in Equation (12) is exactly the same as Equation (11). However, the order which computations are performed is different.

$$\underbrace{\Bigg(\underbrace{\Big(\underbrace{(\underbrace{\frac{1}{23}+\frac{1}{29}}_{\#\ 1}) + (\underbrace{\frac{1}{31}+\frac{1}{37}}_{\#\ 2})\Big)}_{\#\ 3} + \Big(\underbrace{(\underbrace{\frac{1}{41}+\frac{1}{43}}_{\#\ 4}) + (\underbrace{\frac{1}{47}+\frac{1}{53}}_{\#\ 5})\Big)}_{\#\ 6}}_{\#\ 7}\Bigg) + (\underbrace{\frac{1}{57}+\frac{1}{67}}_{\#\ 8})}_{computation\#\ 9} \qquad (12)$$

Both the `pair-wise-fold` and the `tree-fold` coincide about computations # 1 and # 2. However, whereas `pair-wise-fold` retains these two intermediate values while performing computations # 3 and # 4, `tree-fold` consumes # 1 and # 2, immediately in 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` retains $\frac{n}{2}$ intermediate results (n being total length of the sequence being combined in Equation (5)), `tree-fold` retains 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 and final values of added 10 ratios using default `tree-fold` algorithm, computed as shown in Equation (12).

As in Sections 3.1 and 3.2, once again we look at the number of digits computed and retained by `tree-fold`. When we observe the **Digits Computed** column of Table 3 and the curve in Figure 3 (top) corresponding to `tree-fold`, we see at computation # 9, that this algorithm computes the exact same number of digits as `tree-fold` but fewer than `fold-left`. In fact the actual computations performed by `tree-fold` and `pair-wise-fold` are exactly the same, but the order is different. In terms of number of digits retained, `tree-fold` again falls in between the other two implementations.

A recognizable advantage of `tree-fold` over `pair-wise-fold` is that the code (Figure 2) in its implementation is much more concise.[2] The implementation of `pair-wise-fold` is 36 lines, whereas `tree-fold` is 11 lines.

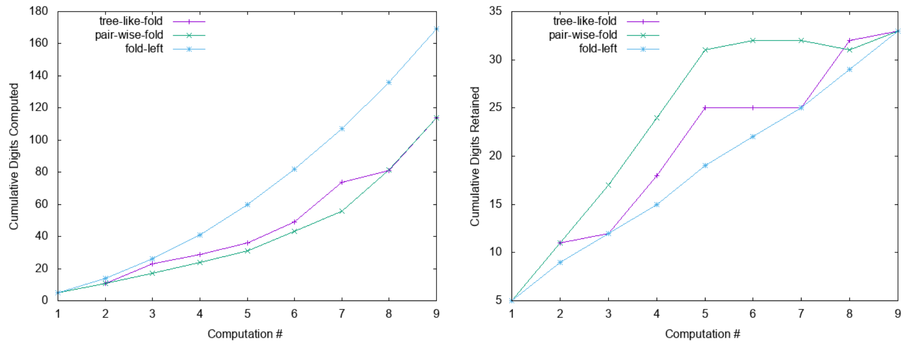
3.4 Summarizing the three `fold` strategies

The three sequences of computations outlined in Sections 3.1, 3.2, and 3.3 are recapped in Figure 3. When we observe the **Digits Computed** column of Tables 1, 2, and 3 and the curve in Figure 3 (left), we see that `fold-left` computes the most digits of the three strategies. 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 1, 2, and 3 and the corresponding curve in Figure 3 (right) we see that it `fold-left` retains the least amount of heap storage during the computation.

[2] The Scala `pair-wise-fold` source code is elided for lack of space. The curious reader may refer to https://scastie.scala-lang.org/p7WeM5vlSXmgKKM3prZiuw.

```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. 2: Scala implementation of the `tree-fold` algorithm.



(a) Left (b) Right

Fig. 3: (Left) Cumulative Digits Computed For Each Fold Strategy. (Right) Quantity of Digits Retained at each Computation State

It is not surprising that (at computation # 9) `pair-wise-fold` and `tree-fold` have computed the exact same number of digits. In fact, the computations are determined by the parentheses which are the same in Equations (11) and (12). In the case of summing rational numbers, by design, `tree-fold` computes the digits more greedily than does `pair-wise-fold`. The rational numbers in question, Equation (5), 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-fold` and `pair-wise-fold` might be oriented differently.

From the plot in Figure 3 (right), we see that all three algorithms retain exactly the same number of digits at computation # 9. This is obvious because the only thing retained 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-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).

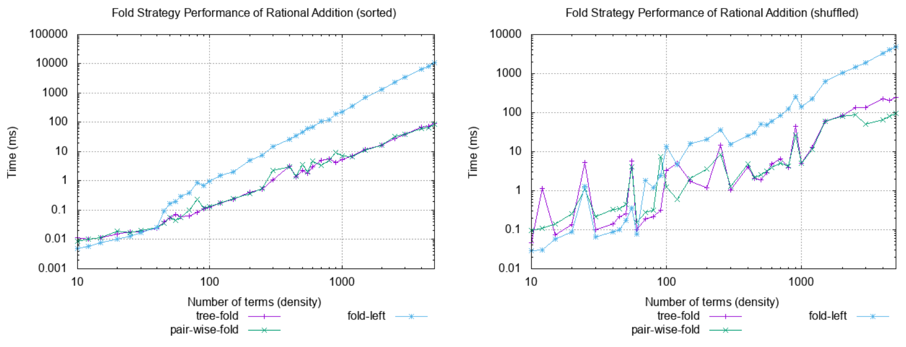
Hardware Overview	
Model Name:	MacBook Pro
Operating System:	macOS Catalina
Version:	10.15.1
Processor Name:	Quad-Core Intel Core i7
Processor Speed:	2.7 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	8 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB

4.1 Results of Ratio Addition

The first experiment we performed entailed summing a sequence of rational numbers of incrementally increasing length. The plots in Figure 4 show the computation time, in milliseconds, of computing sums of different length sequences, using three different folding algorithms. The left and right plots in Figure 4 differ only in that the plot on the left is the result of summing the sequence in sorted order, and the right sums the same sequence shuffled into random order. The x-axis indicates the value of n and the y-axis indicates the time needed to compute the sum

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

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 $1/0$.



(a) Left (b) Right

Fig. 4: Performance of fold strategy on rational addition. Left shows addition in sorted order. Right shows addition in randomized/shuffled order.

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 4, that `tree-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-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.

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 construction of random BDDs of increasingly many Boolean variables is known to have exponential complexity [4, 12]. 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, and 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.

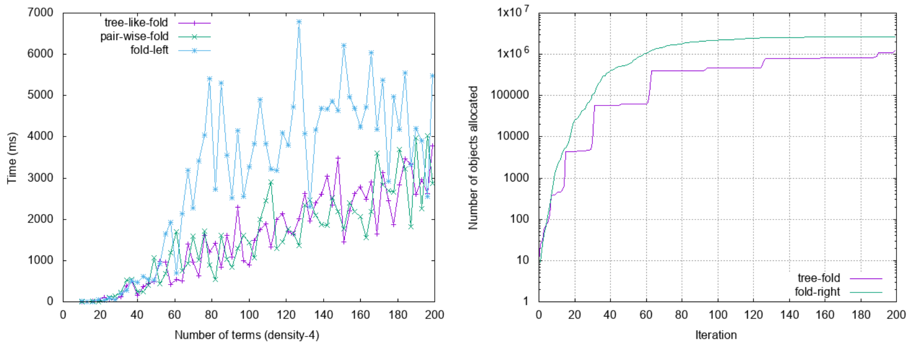
For performance and memory reasons, it is not necessary to compute j explicitly. It suffices to iterate through a sequence of 2^n random bits, generated lazily.

On the average a randomly selected DNF 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.

We did not observe any significant difference between the performance of the implementations based on different `fold` strategies.

4.3 Results of Fixing the Term Length

The experiments discussed in Section 4.2 are based on uniform samples of a space of n-variable Boolean functions. In such a sample, a minterm is likely to contain $\frac{n}{2}$ true literals and $\frac{n}{2}$ false literals. However, in many *real-world* applications the Boolean variables are correlated in a way that each term contains a small number of literals. Langberg *et al.* [19] refer to such Boolean formulas as *simple*. Knuth [18, 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.



(a) Left (b) Right

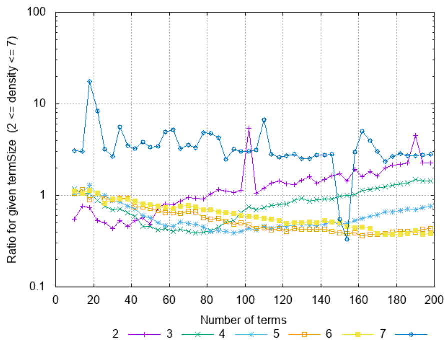
Fig. 5: Performance of fold strategy on construction of BDDs. Each BDD is constructed as an OR of ANDs. Each AND-term is limited to 4 literals. (Left) Each point plotted shows the average time of 3 runs of the `fold` function in question. (Right) Cumulative memory in terms of number of `BddNode` objects allocated. The allocation increases slower for `tree-fold` than for `fold-right`.

The next experiment, whose results are shown in Figure 5 (Left), explores these so-called *simple* Boolean formulas. The experiment is similar to that 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 2) 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.

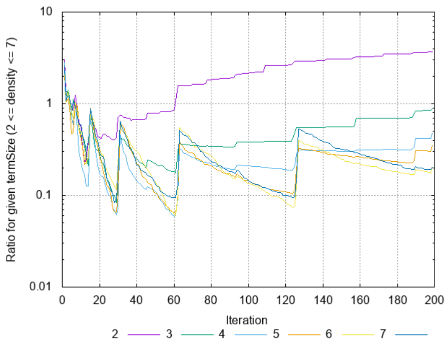
The plot in Figure 5 (Left) shows BDD construction times (in milliseconds) for a range of number of terms going from 10 terms to 200 terms. The results are somewhat surprising. Although we observe a high degree of noise in the curves, we can observe a tendency between about 50 and 180 terms where the `fold-left` algorithm is slower than other others by a factor of 2 to 3.

In Figure 5 (Right) we see the memory allocation growing as one iteration progresses. *I.e.*, the experiment starts with a single 4-term min-term, and at each iteration of the `fold` one additional 4-term minterm is added (Boolean OR operation), causing a larger BDD to be computed. As we expect, the memory allocation exists as the iteration progresses. What is remarkable is that the memory consumed by `fold-right` is at times roughly 10 times that of `tree-fold`. This progression is reminiscent of the plot in Figure 3 (left).

4.4 Results of Varying Term Length



(a) Left



(b) Right

Fig. 6: Performance (left) and relative memory allocation (right) of fold strategy on BDD construction as function of term length. A *ratio* < 1 means `tree-fold` performed better than `fold-left`. Each point plotted is obtained by selecting 10 random DNFs and constructing BDDs from each using the two fold strategies.

In this experiment, whose results are plotted in Figure 6 (Left), 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-fold` and plotted the normalized, relative time consumption.

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

Where a curve lies below $y = 1$, is a region where `tree-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-fold` performs much worse than `fold-left`.

The plot in Figure 6 (Right) we see the relative memory allocation of the `fold-left` vs `tree-fold` for various term densities. A value greater than 1 means `fold-left` is more efficient (allocates less memory), and a value less than 1 means `tree-fold` allocates less memory. We observe in the plot that for densities greater than 2 that the `tree-fold` based approach allocates less memory than does `fold-left`, at least for the iteration range we investigated.

4.5 Reproducing our Results

The code used in the experiments 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.[3] 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 4	`treereduce.RationalFoldTest`	`rationalFoldTest`
Figure 5 Left	`bdd.ReducePerf`	`testLimitedBddConstruction`
Figure 5 Right	`bdd.ReducePerf`	`testGenSizePlotPerFold`
Figure 6 Left	`bdd.ReducePerf`	`testNumBitsConstruction`
Figure 6 Right	`bdd.ReducePerf`	`testGenSizePlotPerFold`

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: `.gnu`, `.png`, and `.csv`. These data files can be found relative to the top of the git repository in cl-robdd/data/fold-performance.

[3] 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.

5 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 [20, 21] did have the functions `MAP` and `MAPCAR` [31], we found no reference to the `fold` function.

As far as we can determine, David Turner (author of SASL and Miranda), seems to be[4] the inventor of `fold` [34]. In 1986, Turner [33] shows how to implement `foldr` in Miranda. The earliest mention of `fold` that we have found, was from 1979 where Turner [35] mentions that "folding a list to the right" is a "commonly occurring pattern" and encapsulates the pattern by defining `foldr` in SASL.

In Common Lisp [3] the function is called `REDUCE`; Scala [29, 7] offers several variants `foldLeft`, `foldRight` and others; Haskell [16, p. 115] offers `foldl` and `foldr` and others; and OCaml [30, p. 63] offers `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 [32, 11]. Although it is not a definitive source of information, we note that the Wikipedia article on `Fold`[5] lists ≈ 44 programming languages which support this feature, sometimes with different names such as `reduce`, `accumulate`, `aggregate`, `compress`, or `inject`.

6 Conclusion

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 significant difference and in other cases it does not.

We have investigated `fold`-based computations whose binary operation degrades in performance due to growth in intermediate values. The tree-based fold implementations (`tree-fold` and `pair-wise-fold`) outperform the traditional `fold-left` implementation in some cses. We have no data to suggest that tree-based folds are 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 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 4.2, 4.3, and 4.4 are as of yet inconclusive.

[4] 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.

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

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. While we observe a correlation between minterm density and BDD construction, we do not have enough evidence at this point to make any predictions with confidence. More research is needed in this area.

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

```scala
// 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)
```

Fig. 7: Scala example of using the `sum-of-products` function to compute the product of sums, simply by swapping the arguments at the call site.

7 Perspectives

In this article we have investigated computations whose *performance* degrades over time due to the progressive growth of intermediate results. Can our method can be applied to computations whose *precision* degrades over time due to accumulating round-off error of intermediate values? An example would be summing or multiplying a sequence of floating point numbers[6]. If intermediate values contain round-off error, then that error compounds each time those values get reused in successive computations. One might suspect that a `tree-fold` approach could mitigate the negative effects of this kind of accumulating round-off, since the technique would limit the number of times such inaccurate intermediate values get reused. This topic deserves investigation.

Our experiments on BDDs have been based on randomly generated samples, albeit with certain constraints which effect the distribution. There is reason to

[6] This idea was suggested by John Hughes.

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 (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), as Equation (13).

$$CNF = \prod_{i=1}^{m} \sum \gamma_i = \prod_{i=1}^{m} \sum_{x \in \gamma_i} x\,. \tag{13}$$

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 7. 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 not investigated how our computations interact with the heap, garbage collection, nor run-time optimizations of the JVM. Previous work [23] 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. 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. 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. Castagna, G.: Covariance and Contravariance: a fresh look at an old issue. Tech. rep., CNRS (2016), https://arxiv.org/pdf/1809.01427.pdf
7. Chiusano, P., Bjarnason, R.: Functional Programming in Scala. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2014)
8. 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/
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. 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
11. Haveraaen, M., Morris, K., Rouson, D., Radhakrishnan, H., Carson, C.: High-Performance Design Patterns for Modern Fortran. Scientific Programming p. 14 (2015)
12. Heap, M.A., Mercer, M.R.: Least Upper Bounds on OBDD Sizes. IEEE Transactions on Computers **43**, 764–767 (June 1994)
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. 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
15. 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
16. Jones, S.P. (ed.): Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (September 2002), http://haskell.org/definition/haskell98-report.pdf
17. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
18. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 12th edn. (2009)
19. 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
20. McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press (1962)
21. McCarthy, J.: History of LISP, p. 173–185. Association for Computing Machinery, New York, NY, USA (1978), https://doi.org/10.1145/800025.1198360
22. Mollin, R.A.: Fundamental Number Theory with Applications, Second Edition. Chapman and Hall/CRC, 2nd edn. (2008)
23. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
24. Newton, J., Demaille, A., Verna, D.: Type-Checking of Heterogeneous Sequences in Common Lisp. In: European Lisp Symposium. Kraków, Poland (May 2016)

25. Newton, J., Verna, D.: Recognizing hetergeneous sequences by rational type expression. In: Proceedings of the Meta'18: Workshop on Meta-Programming Techniques and Reflection. Boston, MA USA (Nov 2018)
26. Newton, J., Verna, D.: Strategies for typecase optimization. In: European Lisp Symposium. Marbella, Spain (Apr 2018)
27. 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
28. Newton, J., Verna, D., Colange, M.: Programmatic Manipulation of Common Lisp Type Specifiers. In: European Lisp Symposium. Brussels, Belgium (Apr 2017)
29. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The scala language specification (2004)
30. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
31. 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
32. 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
33. 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
34. 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
35. 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
36. 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