Deciding associativity in L0

Topics in Programming Languages
Datalogisk institut, Copenhagen University (DIKU)

Kristoffer Søholm & Oleksandr Shturmov
{soeholm,oleks}@diku.dk

June 12, 2013.

Abstract

In the context of parallelizable second-order array combinators, it is often necessary for the function argument to be an associative function. We describe a partial decider for the associative property of functions in the language L0, for implementation in the parallelizing compiler l0c, developed under the HIPERFIT project. The problem is undecidable in general. This paper explores the practical feasibility of such a decider in the context of function arguments to second-order array combinators.

Keywords: Assosiativity, parallelism, second-order array combinators, L0, l0c, HIPERFIT.

1 Introduction

An infix operator $\oplus : S \times S \to S$ is associative iff.

$$\forall\, x, y, z \in S\,.\ (x \oplus y) \oplus z = x \oplus (y \oplus z)\,.$$

One of the benefits of such functions is that a sequence of values $(x_k)_{k=1}^{n}$, $x_k \in S$, can be aggregated into a single value $x \in S$, using $n-1$ applications of $\oplus$, i.e. $x = x_1 \oplus \cdots \oplus x_{n-1} \oplus x_n$. This is more commonly known as a *reduction*. For instance, we can compute the sum of n integers using $n-1$ binary additions.

A straight-forward reduction strategy is $(\cdots((x_1 \oplus x_2) \oplus x_3) \cdots \oplus x_{n-1}) \oplus x_n$, or equivalently, $x_1 \oplus (x_2 \oplus \cdots (x_{n-2} \oplus (x_{n-1} \oplus x_n)) \cdots)$. This is known as a linear left or right *fold*, respectively.

Reductions get more interesting on single-instruction multiple data (SIMD) architectures. A reduction on a SIMD machine can be performed in parallel time $O(\log n)$ on n processors, or $n/m + O(\log m)$ on m processors, using a technique called *parallel prefix scan*, or simply scan.

Scan is an interesting technique in and of itself as it can serve as a basic building block in the design of parallel algorithms, replacing primitive parallel memory referencing. If applicable, this technique typically reduces the asymptotic bounds by a factor of $O(\log n)$ [Blelloch].

Scan takes an infix operator $\oplus : S \times S \to S$, an identity value $i \in S$, such that $i \oplus i = i$, and a sequence $(x_k)_{k=1}^{n}$, $x_k \in S$. It produces the sequence $(i, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus \cdots \oplus x_{n-1} \oplus x_n)$.

To do this, parallel prefix scan performs a tree-like reduction rather than a linear reduction. The sequence $(x_i \oplus x_{i+1})_{i=1}^{n-1}$ is computed first, by pairwise applying $\oplus$ to the elements of the sequence. The sequence $((x_i \oplus x_{i+1}) \oplus (x_{i+2} \oplus x_{i+3}))_{i=1}^{n-3}$ is computed second, by pairwise applying $\oplus$ to the elements of the resulting sequence, and so on until a scalar is reached.

It is important for the correctness of the method that the function supplied to scan is an associative function. The requirements could be relaxed to making sure the function susceptible to tree-like reduction, but that would render the functions identified by this method not necessarily suitable for linear reductions. This would in turn discourage us from applying transformations to turn linear reductions into tree-like reductions: we could speed up the program by using parallel prefix scans rather than folds.

Most compilers will assume that the user has supplied an associative function. Others will require for the function to be explicitly marked as "associative". Others still will try to deduce the associativity property, and warn the user if the function isn't clearly associative. The last is clearly preferable, especially for scien-

tific computing languages such as L0, since such a bug can be very hard to find (the user might not be aware that associativity is a requirement at all).

L0 is an eagerly evaluated, purely functional, second order, array programming language. The language encourages the use of arrays and built-in second-order array combinators for all practical problems. L0 has a focus on efficient execution on vector hardware.

Section 2 provides a proof of the undecidability of the associative property. Section 3 defines the associative type, expanding the definition of associativity in general. Section 4 covers the built-in operators in L0, and whether each of them is associative under chaining. Section 5 covers Light's associativity test which allows testing the associativity of functions built solely from bitwise operators. Section 5 also suggests using a lightweight Light's associativity test in a manner known as property-based testing. This technique works for arbitrary functions. Section 6 covers a more straightforward approach, based on rewriting. Section 7 describes the overall algorithm we suggest. Section 8 discusses possible future work. Section 9 concludes the paper.

2 Undecidability

Undecidability of the property is proven similar to the halting problem.

Proof. We construct a binary Turing machine (TM) M that takes as input a pair of values $x, y \in S$.

Assume there exists a TM A, that given an binary TM M, accepts if M is associative, and rejects otherwise. That is, A is a decider for the associativity problem.

We construct M such that it returns x if $A(M)$ accepts, and a constant $z \in S$ otherwise:

$$M(x, y) = \begin{cases} x & \text{if } A(M) \\ z & \text{otherwise} \end{cases}$$

If $A(M)$ accepts, M is not associative, if $A(M)$ rejects, M is associative, which contradicts our assumptions about A. □

The associative property is Turing-recognizable however, as we can check whether the associative property holds for $\oplus$ for all possible x, y, and z. Our approach is to choose some subset of L0 for which we can decide associativity within reasonable time.

3 The associative type

We introduce the concept of the *associative type* — a function type that unifies with the type of any associative function. We identify this type from the following observation: an associative function consumes at least two values of some type S, and returns a value of type S.

We define it formally as follows:

$$\oplus : S \boxtimes S \boxtimes T_1 \boxtimes T_2 \boxtimes \cdots \boxtimes T_n \to S,$$

where

- $(T_k)_{k=1}^{n}$ for $n \geq 0$ is a sequence of arbitrary types.
- $\boxtimes$ is the commutative equivalent of the regular $\times$, i.e. the arguments to $\oplus$ can be rearranged under unification.

For example, the following types unify with the associative type: $f : S \times S \times S \to S$, $g : S \times S \times \mathcal{Z} \to S$, $h : S \times \mathcal{Z} \times S \to S$, etc.

A function with a type that unifies with the associative type is not necessarily associative, but a function that doesn't definitely is not.

4 Chaining

We consider first whether simply chaining applications of an operator is associative. That is, for each binary operator $\oplus : S \times S \to S$, we consider whether $\forall\, x, y, z \in S\,.\ (x \oplus y) \oplus z = x \oplus (y \oplus z)$ by the L0 specification.

Unary operators are trivially associative under chaining. For binary operators with arguments of two different types, as well as ternary and n-ary operators, it depends. Later, we'll allow intermixing of operators.

Note that even though the binary functions in L0 we are interested in work on arrays, we will mostly focus on the associativity of binary operators working on single values, and thus the examples in the text will predominantly be integer based.

4.1 Arithmetic operators

L0 has a range of binary arithmetic operators, namely `+`, `*`, `-`, `/`, `%`, `pow`, and `~`, where the last operator is a unary negation operator. The operators defined on arguments of type either `int` or `real` (L0 is strongly-typed), yielding a value of the same type.

As of now, there are no restrictions in the L0 standard as to how `reals` are to be represented, so we confine

our attention to `int`s. For `int`s we assume a p-bit two's complement representation.

Our observations are summarized in Table 1/3.

Operator	Chaining is associative
`+`	yes
`*`	yes
`-`	no
`/`	no
`%`	no
`pow`	no
`~`	yes*

Table 1: Associativity of chaining arithmetic operators.

4.2 Relational operators

L0 has a couple binary relational operators, namely `<=`, `<`, and `=`, having the obvious semantics. The operators are defined on arguments both of type either `int` or `real`, yielding a value of type `bool`.

Relational operators are not chainable in L0. These operators are only interesting in combination with other operators.†

4.3 Logical operators

L0 has a couple logical operators, namely `&&`, `||`, and `not`, where the last is a unary negation operator. The operators are defined on arguments of type `bool`, yielding a value of type `bool`.

Our observations are summarized in Table 2/3.

Operator	Chaining is associative
`&&`	yes
`||`	yes
`not`	yes*

Table 2: Associativity of chaining logical operators.

4.4 Bitwise operators

L0 has a range of bitwise operators, namely `^`, `&`, `|`, `>>`, and `<<`, having semantics as in C. The operators are defined on arguments of type `int`, yielding a value of type `int`.

Our observations are summarized in Table 3/3.

Operator	Chaining is associative
`^`	yes
`&`	yes
`|`	yes
`>>`	yes
`<<`	yes

Table 3: Associativity of chaining bitwise operators.

4.5 Other functions

L0 has a range of other builtins, e.g. various second-order array combinators. Of particular interest to us are the functions `concat`, `min`, and `max`, as they are clearly associative.

The other builtins do not meet the requirement of unifiability of their type with an associative type, or fall prey to the fact that functions are not first-class citizens in L0.

Our observations are summarized in Table 5/4.

Operator	Chaining is associative
`concat`	yes
`min`	yes
`max`	yes

Table 4: Associativity of second-order array combinators

5 Light's associativity test

Light's associativity test is an algorithm for testing if a binary operator defined in a finite set is associative. The idea is to take an element a from the set S with binary operator $\cdot$ and define two binary operations, $\circ$ and $*$ (see below), construct their Caley tables, and check if they are the same. It is sufficient to check a proper generating subset of S [Clifford & Preston].

$$x \circ_a y = (x \cdot a) \cdot y$$
$$x *_a y = x \cdot (a \cdot y)$$

*Unary operators can be regarded as associative by definition.

†This is no longer true at the time of our submission. <=, <, and = are now arithemtic operators, all left-fixed, and returning integral values.

For bitwise operations only four Caley tables of size 2×2 are needed. As an example, we check the associativity of the operator $x \oplus y = x \wedge y \mid x \wedge y$:

$\oplus$	0	1
0	0	1
1	1	0

Table 5: Cayley table for the example operator.

Because $\{0\}$ is a proper generating subset of $\{0, 1\}$, it suffices to create the following two tables:

$\circ_0$	0	1
0	0	1
1	1	0

$*_0$	0	1
0	0	1
1	1	0

Table 6: Light's associativity test, showing $\oplus$ is associative.

Because calculations are done on fixed size integers, you could in principle use Light's associativity test to completely determine associativity. This would, however, be intractable for practical purposes - for 32bit integers, this would require at least 2^{32^2} evaluations of the expression in order to fill a single Caley table, which would need to be done numerous times.

The technique can be made tractable by turning it into a heuristic which cannot guarantee that the operator is associative, but instead tries to find a counterexample which proves that it is not associative. This can be done by taking a sample of random values from the given integer range and constructing a partial Caley table, which is then used by Light's test. This approach is inspired by [Claessen & Hughes] and similar property-based testing tools.

As an example, two of the Caley tables for the test of the operator $\texttt{max}(x - y, x \wedge y)$ with 3 randomly generated values in the range -100 to 100 is given below:

$\circ_{-24}$	-44	-24	82
-44	104	84	110
-24	44	24	82
82	150	130	56

$*_{-24}$	-44	-24	82
-44	-24	-44	110
-24	-44	-24	82
82	110	82	152

Table 7: Example test of a binary operator which shows that it is not associative.

6 Rewriting

The straight-forward way of testing associativity is by *rewriting*. We can test whether $\oplus$ is associative by considering whether $(x \oplus y) \oplus z$ and $x \oplus (y \oplus z)$ can be rewritten to be syntactically equivalent.

For instance, if $\oplus(x, y) = (c * x) + (c * y)$, for some constant $c \in S$, we consider whether the equality in Figure 1/5 holds for all x, y, and z.

This analysis requires *rewriting* both expressions as a *sum of products*, followed by *common term elimination*. If c is a constant, we can complete the analysis by *constant folding*, and considering whether the left-hand side is syntactically equivalent to the right-hand side.

Note, we didn't need to utilize the properties that addition and multiplication are also commutative. This is because the variables appear in the same order on either side of the equality. So long as our rewriting rules respect this order, we can proceed freely.

Another property that our rewriting rules must respect are overflows. We should keep in mind that we're dealing with modulo arithmetic. Rewriting should not change the evaluation of any expression with overflow.

6.1 Algorithm

We now specify the overall algorithm:

1. Let $f(x, y) = e$.
2. Let z be a variable name that does not appear in e.
3. Let e_1 be a copy of e.
4. Let e_2 be a copy of e, where all occurrences of y have been replaced by z, and all occurrences of x have been replaced by the program text (e_1) (in that order). Add all occurrences to Q.
5. Let e_3 be a copy of e, where all occurrences of y have been replaced by z, and all occurrences of x have been replaced by y (in that order).
6. Let e_4 be a copy of e, where all occurrences of y have been replaced by the program text (e_3). Add all occurrences to Q.

$$(c * ((c * x) + (c * y))) + (c * z) = (c * x) + (c * ((c * y) + (c * z)))$$
$$((c * (c * x)) + (c * (c * y))) + (c * z) = (c * x) + ((c * (c * y)) + (c * (c * z)))$$
$$((c * (c * x)) + (c * (c * y))) + (c * z) = ((c * x) + (c * (c * y))) + (c * (c * z))$$
$$((c * c) * x) + ((c * c) * y) + (c * z) = ((c * x) + ((c * c) * y)) + ((c * c) * z)$$
$$((c * c) * x) + (c * z) = (c * x) + ((c * c) * z)$$

Figure 1: If $\oplus(x, y) = (c * x) + (c * y)$ is associative, the above should hold for all x, y and z.

7. Use Q to determine whether e_2 and e_4 are semantically equivalent.

For instance, if $f(x, y) = (c * x) + (c * y)$:

$$e_1 = (c * x) + (c * y)$$
$$e_2 = \underbrace{(c * ((c * x) + (c * y)))}_{q_1} + (c * z)$$
$$e_3 = (c * y) + (c * z)$$
$$e_4 = (c * x) + \underbrace{(c * ((c * y) + (c * z)))}_{q_2}$$
$$Q = [q_1, q_2]$$

Q is a working queue which we will use to attempt to rewrite e_2 and e_4 into syntactically equivalent expressions. We proceed to rewrite the subexpressions q_1 and q_2 by applying various rewrite rules. A rewrite rule will typically introduce new parentheses, which designate monotonically nonincreasing subexpressions to be rewritten next.

For instance, from the rewrite rules in Figure 3/6, we see that for q_1, in the example above, we apply the rule T-PLUS-L. This rewrites q_1 into

$$\underbrace{(c * (c * x))}_{q_3} + \underbrace{(c * (c * y))}_{q_4},$$

where q_3 and q_4 get queued for rewriting. q_3 and q_4 both fall prey to the associative chaining property of multiplication. Figure 2/6 indicates that we rewrite such chains to be left-associative. If no rewrite rules match a subexpression on the queue, the subexpression is simply popped from the queue.

The algorithm terminates if there are no circular rewrites. This can be checked by building a *rewrite rule graph* where the nodes are the rewrite rules. There is a directed edge from one rewrite rule to another if the rewrite yields an expression rewritable by the other. If there are no cycles in this graph, the finite nature of syntax trees of finite programs makes the algorithm terminate.

6.2 Rules

The soundness of the algorithm depends on the soundness of the rewrite rules. The incompleteness of the method was proven in § 2/2.

In general, the rewrite rules are based on the observations made in § 4/2, as well as the distributive laws of various operators. We keep the algorithm deterministic such that there is at most one edge going out of any rewrite rule in the rewrite rule graph.

An operator $\otimes : S \times S \to S$ is distributive over $\oplus : S \times S \to S$ iff $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$(it is left-distributive) and $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$(it is right-distributive).

First, there are the rewrite rules that follow directly from § 4/2 in Figure 2/6. We specify only their left-associative parts because otherwise this would lead to cycles in the rewrite rule graph. We chose to rewrite all chains into left-associative chains as most operators in L0 are left-associative. We hypothesise this will make the recursion bottom out faster in practice.

Second, we have the rules for various arithmetic, logical, and bitwise operators in Figure 3/6. To construct the rules we proceed as follows: for every operator $\otimes$, consider every other operator $\oplus$, as to whether $\otimes$ is distributive over $\oplus$.

Similar rules can be written for `min`, `max`, and `if-then-else`. `concat` however does not distribute over any of the operators.

We do not claim that this list of rewrite rules is complete in any way. Our approach allows for rules to be added once they are discovered.

$$\textsc{Plus}: \frac{}{e + (e_1 + e_2) \to (e + e_1) + e_2} \qquad \textsc{Times}: \frac{}{e * (e_1 * e_2) \to (e * e_1) * e_2}$$

$$\textsc{Land}: \frac{}{e \,\&\&\, (e_1 \,\&\&\, e_2) \to (e \,\&\&\, e_1) \,\&\&\, e_2} \qquad \textsc{Lor}: \frac{}{e \mid\mid (e_1 \mid\mid e_2) \to (e \mid\mid e_1) \mid\mid e_2}$$

$$\textsc{Xor}: \frac{}{e \,\hat{}\, (e_1 \,\hat{}\, e_2) \to (e \,\hat{}\, e_1) \,\hat{}\, e_2}$$

$$\textsc{Band}: \frac{}{e \,\&\, (e_1 \,\&\, e_2) \to (e \,\&\, e_1) \,\&\, e_2} \qquad \textsc{Bor}: \frac{}{e \mid (e_1 \mid e_2) \to (e \mid e_1) \mid e_2}$$

$$\textsc{BSL}: \frac{}{e << (e_1 << e_2) \to (e << e_1) << e_2} \qquad \textsc{BSR}: \frac{}{e >> (e_1 >> e_2) \to (e >> e_1) >> e_2}$$

$$\textsc{Min}: \frac{}{\texttt{min}\,(e, \texttt{min}\,(e_1, e_2)) \to \texttt{min}\,(\texttt{min}\,(e, e_1), e_2)}$$

$$\textsc{Min}: \frac{}{\texttt{max}\,(e, \texttt{max}\,(e_1, e_2)) \to \texttt{max}\,(\texttt{max}\,(e, e_1), e_2)}$$

$$\textsc{Concat}: \frac{}{\texttt{concat}\ (e, \texttt{concat}\ (e_1, e_2)) \to \texttt{concat}\ (\texttt{concat}\ (e, e_1), e_2)}$$

Figure 2: Rewriting rules for operators that are associative under chaining.

$$\textsc{P-Minus-L}: \frac{}{e + (e_1 - e_2) \to (e + e_1) - e_2}$$

$$\textsc{P-Min-L}: \frac{}{e + \texttt{min}\,(e_1, e_2) \to \texttt{min}\,((e + e_1), (e + e_2))} \qquad \textsc{P-Max-L}: \frac{}{e + \texttt{max}\,(e_1, e_2) \to \texttt{max}\,((e + e_1), (e + e_2))}$$

$$\textsc{T-Plus-L}: \frac{}{e * (e_1 + e_2) \to (e * e_1) + (e * e_2)} \qquad \textsc{T-Minus-L}: \frac{}{e * (e_1 - e_2) \to (e * e_1) - (e * e_2)}$$

$$\textsc{T-Mod-L}: \frac{}{e * (e_1 \,\%\, e_2) \to (e * e_1) \,\%\, (e * e_2)} \qquad \textsc{T-ShiftL-L}: \frac{}{e * (e_1 << e_2) \to (e * e_1) << e_2}$$

$$\textsc{T-Min-L}: \frac{}{e * \texttt{min}\,(e_1, e_2) \to \texttt{min}\,((e * e_1), (e * e_2))} \qquad \textsc{T-Max-L}: \frac{}{e * \texttt{max}\,(e_1, e_2) \to \texttt{max}\,((e * e_1), (e * e_2))}$$

$$\textsc{A-Or-L}: \frac{}{e \,\&\&\, (e_1 \mid\mid e_2) \to (e \,\&\&\, e_1) \mid\mid (e \,\&\&\, e_2)} \qquad \textsc{BA-BOr-L}: \frac{}{e \,\&\, (e_1 \mid e_2) \to (e \,\&\, e_1) \mid (e \,\&\, e_2)}$$

Figure 3: Rewriting rules for arithmetic, logical, and bitwise operators. The above rules cover only left-distributive laws. Right-distributive laws have symmetrical rewrite rules. (Bit shifting to the left by e is equivalent to multiplying by 2^e.)

7 Future work

One possible direction of future work is to expand Light's test beyond bitwise operators. We can make the observation that a binary addition is a ternary bitwise operator, where for every pair of bits there is a third, carry bit.

We can state addition as a bitwise operation, i.e. $f : \{0,1\} \times \{0,1\} \times \{0,1\} \to \{0,1\} \times \{0,1\}$, and $f(x,y,c) = ((x \oplus y) \oplus c, (x \wedge y) \vee ((x \vee y) \wedge c))$, where the first element is the result bit, and the second element is the carry bit. Note, a function like f in L0, using bitwise operators over integers, is by no means the same as addition over integers.

If we apply the idea of the associative type, we can state the problem of testing the associativity of integer addition as testing whether for all $x, y, z, c \in \{0,1\}$:

$$f(f(x,y,c),z,c) = f(x,f(y,z,c),c).$$

Light's associativity can be applied here by first letting $c = 0$ and then letting $c = 1$. If f turns out to be associative in both these cases, we can say that f is associative in general.

This approach requires time linear in the number of carry-based operations in the function body, i.e. there is an independent carry bit for each such operation. This makes the problem feasible such operations, unlike considering the entire integer domain.

Subtraction can be dealt with in a similar manner, if we keep in mind that $x + y = x + (\sim y)$. Other operators however, get more tricky as we have to keep more than a single carry bit in mind when doing e.g. multiplication, division, raising to various powers, or bit shifting.

Another possible direction of future work is expading the set of rewriting rules. Especially useful would be generalised rules about `min`, `max`, and `if-then-else`.

A third, perhaps less useful direction is to generate variable constraints for unconstrained variables if the function were to be made associative.

If $\oplus(x,y,c) = (c * x) + (c * y)$, and we would like to know whether $\oplus$ is associative wrt. x and y, we reach no useful conclusion with the above method. We can find out whether there exists a c for which this property holds:

$$\begin{aligned}
((c * c) * x) + (c * z) &= (c * x) + ((c * c) * z) \\
(c * (c * x)) + (c * z) &= (c * x) + (c * (c * z)) \\
(c * x) + z &= x + (c * z) \\
(c * x) - (c * z) &= x - z \\
c * (x - z) &= x - z \\
c &= (x - z) \,/\, (x - z) \\
c &= 1
\end{aligned}$$

The compiler could at this point suggest that c be replaced by a constant if the function is to be associative. Such an analysis may reveal a constant, a range, or a set of possible values, or it may fail completely.

8 Conclusion

The undecidability of the associative property puts a bound on it's theoretical, but not necessarily practical feasibility.

We have looked into mutliple methods for deciding associativity. Only one of these methods is applicable to L0 programs in general. The cost is loss of soundness, i.e. functions may be deemed associative when they in fact ar not. Other methods are sound (no false positives), but constrained to a rather mundane subset of the L0 language. These methods also do not in general allow for the function in question to use other user-defined functions, unless these functions can be efficiently inlined.

The techniques presented in this paper are by no means constrained to L0. They are aplicable to all languages with similar semantics.

References

[Blelloch] Guy Blelloch. *Scans as Primitive Parallel Operations*. 1987. IEEE Transactions on Computers. Volume 38. 1526–1538.

[Clifford & Preston] Alfred Hoblitzelle Clifford and Gordon Bambord Preston. *The Algebraic Theory of Semigroups, Volume 1*. 1961. American Mathematical Society. 1–9. URL: http://books.google.dk/books?id=IbOmAAAAMAAJ.

[Claessen & Hughes] Koen Claessen and John Hughes. *QuickCheck: a lightweight tool for random testing of Haskell programs*. 2000. SIGPLAN Not. 35, 9. 268–279.