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 scientific 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).

1.1 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. □

1.2 The associative type

For ease of analysis, 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.

2 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.

2.0.1 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 `ints`. For `ints` we assume a p-bit two's complement representation.

Our observations are summarized in Table 1.

2.0.2 Relational operators

L0 has a couple binary relational operators, namely `<=`, `<`, and `=`, having the obvious semantics. The operators are defined on arguments both

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

Table 1: Associativity of chaining arithmetic operators.

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.†

2.0.3 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.

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

Table 2: Associativity of chaining logical operators.

2.0.4 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.

*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.

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

Table 3: Associativity of chaining bitwise operators.

2.0.5 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 6.

Operator	Chaining is associative
concat	yes
min	yes
max	yes

Table 4: Associativity of second-order array combinators

3 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 $\oplus(x, y) = x * c + y * c$, for some constant $c \in S$, we consider whether the equality in Figure 1 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.

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

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

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.

Unfolding all parens is straight-forward. Unparenthesise in a bottom-up fashion. An expression is represented as a tree, traverse the tree, identify all parentheses. Sort the parenetheses by depth in the tree. Take the parentheses in order. Apply parenthesing rewritings first.

No need to unfold all parens, just all those that replace variables. Let x and y be the variable names of the arguments in question. Let z be variable name that does not occur in the program text of f. Let f1 be the original program text f. Let f2 be the original program text f with y replaced by z, and x replaced by (f1) (in that order). Let f3 be the original program text f with y replaced by z, and x replaced by y (in that order). Let f4 be the original program text f with y replaced by (f3). This leads to exactly 4 parentheses that we wish unfolded. To do this, we follow the P- rewriting rules.

We need to use a work queue. Mention precedance rules.

The general algorithm: is the operator chain-associative? is the operator distributive over any other operators in L0?

We summarise the rules for arithmetic operators in Figure 3. The approach to constructing the rules in general is to consider every operator $\oplus$, associative under chaining. For each $\oplus$ we consider whether the $\oplus$ is distributive under every $\otimes$. This will allow us to apply the rules by following the rules of precedence of the language. We add an additional rule to transform chains.

Precedence	Operators	Fixity
1	`not`, ~	None
2	`pow`	Left
3	`*`, `/`, `%`	Left
4	`+`, `-`	Left
5	`<<`, `>>`	Left
6	`<=`, `<`, `=`	None†
7	`&`, `^`, `|`	Left
8	`&&`	Left
9	`||`	Left
10	`if`	None

Table 5: The precedence and fixity of operators in L0. Lower precedence means stronger binding.

We don't need to define rules for expanding parentheses over non-assoc operators, as such functions won't associate anyways.

For bitwise operators we simply use Light's associativity test.

For the `min` and `max` functions, the rewrite rules are a bit more complicated. First off, we need to do some extra work

4 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 [A.H. Clifford & G.B. Preston].

$$\text{P-PLUS-PLUS}: \frac{}{(e_1 + e_2) + e \to e_1 + (e_2 + e)}$$
$$\text{P-TIMES-TIMES}: \frac{}{(e_1 * e_2) * e \to e_1 * (e_2 * e)}$$

Figure 2: Rewriting rules for chainable operators.

$$\text{P-TIMES-MOD-R}: \frac{}{(e_1 \,\%\, e_2) * e \to (e_1 * e) \,\%\, (e_2 * e)}$$
$$\text{P-TIMES-PLUS-R}: \frac{}{(e_1 + e_2) * e \to (e_1 * e) + (e_2 * e)}$$
$$\text{P-TIMES-MINUS-R}: \frac{}{(e_1 - e_2) * e \to (e_1 * e) +\sim (e_2 * e)}$$
$$\text{P-TIMES-SHIFTL-R}: \frac{}{(e_1 << e_2) * e \to (e_1 * e) << e_2}$$

Figure 3: Rewriting rules for arithmetic operators. The above are only right-distributions. Left-distributions are symmetrical. Note, bit shifting to the left by e is equivalent to multiplying by 2^e.

$$\text{P-ANDR}: \frac{}{(e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n) \,\&\&\, e \to e_1 \,\&\&\, e \,||\, e_2 \,\&\&\, e \,||\, \cdots \,||\, e_n \,\&\&\, e} \; (n \geq 2)$$
$$\text{P-ANDL}: \frac{}{e \,\&\&\, (e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n) \to e \,\&\&\, e_1 \,||\, e \,\&\&\, e_2 \,||\, \cdots \,||\, e \,\&\&\, e_n} \; (n \geq 2)$$

$$\text{P-ORR}: \frac{}{(e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n) \,||\, e \to e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n \,||\, e} \; (n \geq 2)$$
$$\text{P-ORL}: \frac{}{e \,||\, (e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n) \to e \,||\, e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n} \; (n \geq 2)$$

$$\text{P-NOT}: \frac{}{\texttt{not}\ (e_1 \,||\, e_2 \,||\, \cdots \,||\, e_n) \to \texttt{not}\, e_1 \,||\, \texttt{not}\, e_2 \,||\, \cdots \,||\, \texttt{not}\, e_n} \; (n \geq 2)$$

Figure 4: Rewriting rules for logical operators.

$$\text{S-MIN}: \frac{}{\texttt{min}\,(e_y, e_x) \to \texttt{min}\,(e_x, e_y)}$$

$$\text{P-MINL}: \frac{}{\texttt{min}\,(e_x, e_1 \cdots \texttt{min}\,(e_y, e_z) \cdots e_n) \to \texttt{min}\,(\texttt{min}\,(e_x, e_1 \cdots e_y \cdots e_n), e_1 \cdots e_z \cdots e_n)} \; (n \geq 0)$$

Figure 5: Rewriting rules for `min`. Rewrite rules for `max` are symmetrical. e_x, e_y, and e_z represent expressions that contain a variable x, y, and z, respectively. The sequences $(e_k)_{k=1}^{n}$ are independent of e_x, e_y, and e_z.

$$x \circ y = (x \cdot a) \cdot y$$
$$x * y = x \cdot (a \cdot y)$$

For binary operations only four Caley tables of size 2×2 are needed. For example, we can 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 6: 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	1
0	0	1
1	1	0

$*$	0	1
0	0	1
1	1	0

Table 7: Caley tables for 0

References

[Kehayopulu & Argyris] Niovi Kehayopulu and Philip Argyris. *An algorithm for Light's associativity test using Mathematica*. Journal of computing and information, 3(1): 87–98. ISSN 11803886.

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

[A.H. Clifford & G.B. Preston] Clifford, A.H. and Preston, G.B., *The Algebraic Theory of Semigroups, volume 1*. 1961. lc61015686i. http://books.google.dk/books?id=IbOmAAAAMAAJ. American Mathematical Society. 1–9.