The Undecidability of Associativity and Commutativity Analysis

ARTHUR CHARLESWORTH
University of Richmond

Associativity is required for the use of general scans and reductions in parallel languages. Some systems also require functions used with scans and reductions to be commutative. We prove the undecidability of both associativity and commutativity. Thus, it is impossible in general for a compiler to check for those conditions. We also prove the stronger result that the resulting relations fail to be recursively enumerable. We prove that these results hold for the kind of function subprograms of practical interest in such a situation: function subprograms that, due to syntactical restrictions, are guaranteed to halt. Thus, our results are stronger than one can obtain from Rice's Theorem. We also obtain limitations concerning the construction of functions and limitations concerning compiler-generated run-time checks. In addition, we prove an undecidability result about programmer-constructed run-time checks.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*general scan operators, general reduction operators*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computability theory*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*decision problems*

General Terms: Languages, Theory

Additional Key Words and Phrases: Associative function, commutative function, loop program, parallel prefix, primitive recursive, recursive, recursively enumerable, reduction, scan, sequence, undecidable

1. INTRODUCTION

Many problems related to software are known to be undecidable. The lasting contribution of discovering and proving such results is to help define the boundaries within which software developers must work, rather than to provide techniques applicable in software development. Recent such results in this journal concern the undecidability of flow-sensitive alias analysis [Ramalingam 1994],

This work was partially supported by a grant from the University of Richmond and is also related to research funded by NASA grant NAG-1-774.
Author's address: Department of Mathematics and Computer Science, University of Richmond, Richmond, VA 23173; email: charlesworth@acm.org.

the undecidability of context-sensitive data-dependence analysis [Reps 2000], and the undecidability of context-sensitive synchronization-sensitive analysis [Ramalingam 2000]. Our undecidability results concern associativity and commutativity analysis.

It is well known that reductions and scans facilitate solving a variety of problems [Kruskal et al. 1985; Hillis and Steele Jr. 1986]. These operations, in conjunction with just selection and broadcasting, yield an elegant, yet powerful model of parallel computation [Akl and Stojmenovic 1996]. (Reductions are sometimes called "generalized sums" and scans are sometimes called "prefix sums".)

Software platforms support the use of programmer-defined functions in reductions and scans by providing general operators that take a sequence and the programmer-defined function f and produce the resulting reduction or scan. By compiling high-level software platforms into low-level primitives on their machines, parallel computer vendors can compete in the arena of efficiency to encourage programs written for those software platforms to be ported to their machines.

Functions that fail to be associative can produce correct answers on one implementation of a general reduction or scan operation and incorrect answers on another, even though both implementations are correct. Thus, such software platforms require that a function used with a scan or reduction must be associative and this assumption is made by the implementors of the platform. It is not just a theoretical possibility that programmers introduce errors into software by incorrectly determining the associativity of functions. According to the chief implementor of the MPI general reduction operator on the IBM SP2, the correctness of that implementation was questioned by some of its users. But in each case, the reason for the problem was the failure on the part of programmers to properly ensure the associativity of functions (H. Franke, personal communication).

Some systems require f to be both associative and commutative [IBM 1994; Intel 1994; Parasoft 1990]. Other systems just require f to be associative [Bala et al. 1995] and some of these also give the programmer the option of asserting that f is commutative so the implementation can use commutativity to advantage [Snir et al. 1996].

We prove that associativity is undecidable. Thus, it is impossible in general for compile-time checks to verify the associativity of functions. Moreover, we prove the stronger result that the set of function subprograms that are associative is not recursively enumerable. We also prove the same impossibility results for commutativity. We prove that these results hold for the kind of function subprograms of practical interest in such a situation: function subprograms that, due to syntactical restrictions, are guaranteed to halt. Thus, our results are stronger than one can obtain from Rice's Theorem. We also show that straightforward construction from a base set of functions does not provide a satisfactory, general approach to obtaining associative functions. We also obtain limitations, for both associativity and commutativity, concerning compiler-generated run-time checks and undecidability results for programmer-constructed run-time checks.

Many researchers have developed techniques for the automatic extraction of reductions and scans from the loops of sequential programs. In contrast, the focus of this paper is on programmer-defined function subprograms.

The paper is organized as follows: Section 2 provides background information, Section 3 considers compile-time results, Section 4 considers construction from a base set of functions, Section 5 considers compiler-generated run-time checks, Section 6 considers programmer-constructed run-time checks, and Section 7 presents conclusions.

2. BACKGROUND

Let X be a set and let f be a function subprogram that defines a function from $X \times X$ to X. Since such a function subprogram defines a unique such function, for convenience in notation we let f also denote that function; our treatment is not affected by the fact that a single function can be implemented by infinitely many function subprograms. Recall that f is associative if and only if the following *associativity equation* holds for all x, y, and z in X:

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

Also recall that f is *commutative* if and only if, for all x and y in X, $f(x, y) = f(y, x)$.

These definitions assume f is defined on all of $X \times X$; that is, the function subprogram defining f halts on all inputs. That is the situation of practical interest in the context of scans and reductions. Of course, determining whether an arbitrary function subprogram halts on all inputs is an undecidable problem. We avoid nonhalting computations by focusing on bounded-loop programs, also called loop programs [Meyer and Ritchie 1967b]. An obvious algorithm determines whether a function subprogram is in this set, since the definition of the set is purely syntactical.

We assume in this paper that all programs are written in Pascal. Our results apply to Pascal-like restrictions of other languages. For example, the syntax of the **for** loop in C and the **DO** loop in Fortran must be restricted so that, for instance, it is not possible to change the value of a loop control variable within such a loop.

Definition 1. A *bounded-loop* program is a function subprogram that makes no calls on subprograms and whose only sequence control is via **for** loops, **begin end** blocks, and **if** and **if then else** statements. An *input-free* program is a program containing no input statements. To simplify our treatment we also assume that an input-free program contains no subprograms.

Our focus on bounded-loop programs leaves out no function of practical interest. This follows from two facts. First, a function from tuples of natural numbers to natural numbers can be defined by a bounded-loop program if and only if it is primitive recursive [Meyer and Ritchie 1967b]. Of course, natural numbers can be used to code any data type. Second, the set of primitive recursive functions contains all the recursive functions that halt on all inputs "that we could conceivably want to compute for any practical purpose" [Phillips 1992, p. 111].

Roughly, this is because if f is a recursive function that is not primitive recursive, the running time of any program to compute f grows at an enormous rate.

The standard approach to proving the undecidability of a decision problem concerning properties of a function is to use Rice's Theorem [Hopcroft and Ullman 1979; Rogers 1987]. Rice's Theorem concerns recursive functions in general, including those that fail to halt on some inputs. Interpreted in terms of unrestricted programs, Rice's Theorem states that each nontrivial extensional property is undecidable. (An "extensional" property of a program is a property that depends on the input/output functional behavior of the program rather than on the syntactical form of the program.) For unrestricted programs, it follows that the property of being associative is undecidable; we need the stronger result that this undecidability also holds for bounded-loop programs. For unrestricted programs, Rice's Theorem implies that the property of being both primitive recursive and associative is undecidable, but it also implies that the property of being primitive recursive alone is undecidable. Since the functions of practical interest in the context of this paper arise from bounded-loop programs, and thus are known to be primitive recursive, the framework of Rice's Theorem is inappropriate.

We assume all numbers are accurately represented within the computer, so addition and multiplication are associative. Although this assumption cannot be satisfied on a physical computer, it provides a standard, useful level of abstraction. (If this assumption is not made and X has finite size n, then for bounded-loop programs in the worst case: associativity is decidable and requires n^3 equality checks and commutativity is decidable and requires $n(n-1)/2$ equality checks.)

3. COMPILE-TIME LIMITATIONS

This section shows that, in general, compilers cannot check for either associativity or commutativity, because determining whether or not a bounded-loop program is associative (or commutative) is an undecidable problem. We actually prove the following stronger results: the set of associative bounded-loop programs and the set of commutative bounded-loop programs both fail to be recursively enumerable.

LEMMA 3.1 (UNDECIDABILITY OF INPUT-FREE HALTING PROBLEM). *The following problem is undecidable: Given an input-free program P, will P halt?*

PROOF. Follows from the undecidability of the blank-tape halting problem [Phillips 1992]. □

THEOREM 3.2. *Let X be the set of integers. The set S of bounded-loop programs from $X \times X$ to X that are associative is not recursively enumerable.*

PROOF. First, we prove that S is not recursive, by showing that the recursiveness of S would imply a decision method for solving the halting problem for an arbitrary input-free program P. We show that P fails to halt if and only if f_P is associative, where f_P is the commutative, bounded-loop program given

```
function f(x, y: Integer): Integer;
  var HaltFound: Boolean;
      i: Integer;
begin
  HaltFound := False;
  ... Prepare to begin tracing the program P.
  for i := 1 to (x + y) do
      if not HaltFound then
        begin
          ... Trace the execution of next step of the program P.
          ... If that step causes P to halt, assign True to HaltFound.
        end;
  if HaltFound then f := x * y else f := x + y
end;
```

Fig. 1. The function f_P used in the proof of Theorem 3.2.

by the schema in Figure 1. (That f_P is a bounded-loop program follows from the fact that tracing any single step of P can be determined by a bounded-loop program [Meyer and Ritchie 1967b]; to avoid subprograms we can assume this is done via an inline expansion.) If P fails to halt, then f_P is associative, since f_P is identically the addition function. If P halts, then let n be the number of steps in the execution of P, where $n > 0$, and note that $f_P(-1, f_P(1, n+1)) = -(n+1)$ whereas $f_P(f_P(-1, 1), n+1) = 0$, so f_P is not associative.

To complete the proof, it suffices to show that the set of bounded-loop programs from $X \times X$ to X that fail to be associative is recursively enumerable, since if a set and its complement are both recursively enumerable, the set is recursive. Let $F = \{f_{\alpha(1)}, f_{\alpha(2)}, \ldots\}$ be a recursive enumeration of all bounded-loop programs from $X \times X$ to X, using the purely syntactical definition of such subprograms. Let $T = \{t_{\beta(1)}, t_{\beta(2)}, \ldots\}$ be a recursive enumeration of all triples of integers. The enumerations α and β induce a recursive enumeration γ of the set $F \times T$. Now let the output list O be initially empty, denote $\gamma(i)$ as $(\gamma(i)_1, \gamma(i)_2)$ and, iterating for $i = 1, 2, \ldots$, if $\gamma(i)_1$ is not yet in O, check the associativity equation for function subprogram $\gamma(i)_1$ and triple $\gamma(i)_2$, inserting $\gamma(i)_1$ on O if the equation does not hold. The resulting list O is a recursive enumeration of the set of bounded-loop programs from $X \times X$ to X that fail to be associative. □

THEOREM 3.3. *Let X be the set of integers. The set S of bounded-loop programs from $X \times X$ to X that are commutative is not recursively enumerable.*

PROOF. Modify the proof of Theorem 3.2 by changing the limit on the loop in Figure 1 to $(x - y)$. Note that if P halts in n steps, with $n > 0$, then $f_P(n, 0) = 0$ whereas $f_P(0, n) = n$. In the rest of the proof, use integer doubles, instead of integer triples. □

The compile-time limitations do not require the full power of the theorems, since they follow from just the nonrecursiveness of the two decision problems. Do any important limitations follow from applying the full power of the theorems? We explore that question in the following two remarks.

Remark 3.4. It follows from Theorem 3.2 that a software tool cannot provide access to all associative bounded-loop programs through a mechanism that lets the programmer select the desired function subprogram through interactively answering yes/no questions about the nature of the program. We assume this model of interaction: the programmer understands the nature of the desired function subprogram well enough to use such a mechanism, might not be sure the resulting function is associative, but is sure it can be defined as a bounded-loop program; if the resulting function is indeed associative, given sufficient time the mechanism would ultimately allow the programmer to learn that fact. (On the other hand, if the resulting function is not associative, the programmer could use the mechanism forever without learning that fact.) A breadth-first traversal of the search tree provided by such a mechanism would yield a recursive enumeration of the possible associative, bounded-loop programs from which to select, contradicting Theorem 3.2. A similar remark applies to the commutative, bounded-loop programs.

Remark 3.5. One should not conclude too much from the preceding remark. The programmer might not want the software tool to provide access to *all* associative bounded-loop programs, regardless of whether they satisfy certain additional properties, such as good run-time efficiency and a program text the programmer can easily comprehend. The limitation described in Remark 3.4 ignores such properties.

Also, Theorems 3.2 and 3.3 pertain to function *subprograms*, not to functions. Although that is the appropriate context for a compile-time limitation, it may not be the appropriate context for a limitation on an interactive software tool. The user of such a software tool might want a function subprogram having a particular desired functional (i.e., input/output) behavior, regardless of the form of the program. We can easily see that Remark 3.4 does not apply in that situation by considering the case of commutative functions. A software tool could use a recursive enumeration of all bounded-loop programs from $X \times X$ to X (which exists, since the definition of such subprograms is purely syntactical) and for each such function subprogram h, it could list the function subprogram whose signature is

function $f(x, y$: Integer): Integer;

and whose body is (the result of replacing calls by inline expansions in) the return statement

$f := h(\min(x, y), \max(x, y))$

The mathematical functions corresponding to the resulting function subprograms range over all the commutative, primitive recursive functions.

4. LIMITATIONS ON CONSTRUCTION FROM A BASE SET OF FUNCTIONS

A programmer-defined function f is commutative whenever it is constructed so it accesses its two parameters via commutative functions. That is, if g_1 and g_2 are commutative function subprograms and h is any binary function

subprogram, then the function

$$f(x, y) = h(g_1(x, y), g_2(x, y))$$

is commutative.

However, in the context of this article, when the programmer wants a commutative function, the programmer also wants the function to be associative. Associativity is much less well-behaved than commutativity. Ensuring the associativity of h, as well as that of g_1 and g_2, does not guarantee that the resulting f is associative, even if f is restricted to a straight-line function subprogram. To see this, let $x = 1$, $y = 2$, and $z = 3$, let g_1 and g_2 be addition of integers, let h be multiplication of integers, and note that $f(x, f(y, z))$ fails to equal $f(f(x, y), z)$, since

$$f(1, f(2, 3)) = f(1, 25) = 26^2 \neq 12^2 = f(9, 3) = f(f(1, 2), 3).$$

Moreover, the same x, y, and z show that neither the sum of the associative functions g_1 and g_2 nor their product is an associative function.

5. LIMITATIONS ON COMPILER-GENERATED RUN-TIME CHECKS

In view of the limitations on compile-time checks and construction of functions from a base set, we consider run-time checking of associativity and commutativity when a reduction or scan is executed. In this situation, there is just one particular sequence $\langle x_1, \ldots, x_n \rangle$ to consider, with $n > 2$. The concern in this situation is not necessarily with the general properties of f, it is with what happens when f is applied $n - 1$ times, starting with the values in $\{x_1, \ldots, x_n\}$. That is, the concern is no more than whether f is associative and commutative on the set $f^{n-2}\{x_1, \ldots, x_n\}$, where that notation is defined as follows.

Definition 2. Let $A \subseteq X$ and let i be an integer. We define $f^0 A$ to be A and, for $i > 0$, we define $f^i A$ to be the set

$$A \cup \{f(x, y) \mid x \in f^j A \text{ and } y \in f^k A, \text{ where } j \geq 0,\ k \geq 0, \text{ and } j + k < i\}$$

so that $f^i A$ consists of the values that result from applying f at most i times, using members of A. We say that A is *closed* under f iff $f^1 A = A$.

Until further notice, we consider just the simplest situation, in which $\{x_1, \ldots, x_n\}$ is closed under f. When f is indeed commutative, exhaustively checking that

$$f(x_i, x_j) = f(x_j, x_i)$$

holds for all distinct x_i and x_j requires carrying out $n(n-1)/2$ equality checks. Even when that is accomplished using n processors with an efficiency of 1, the number of equality checks performed per processor is at least $(n-1)/2$, so linear time is required. That approach is unacceptable: programmers expect the entire work of a reduction or scan, when implemented using n processors, to be accomplished in less than linear time on typical parallel computers; for instance, in $O(log(n))$ time on a hypercube. (Remark 5.1 gives related results.) Using an exhaustive approach for checking the associativity of f is even worse: when f is indeed associative, there would be n^3 checks of the associativity equation,

so quadratic time is required, assuming an algorithm using n processors with efficiency 1.

Since exhaustive checking is unacceptable, we consider the adequacy of more efficient run-time checking. We examine reductions, but the results also apply to scans. We assume a processor participating in calculating the f-reduction of $\langle x_1, \ldots, x_n \rangle$ performs successive computations of the form $f(x', x'')$, where $x', x'' \in \{x_1, \ldots, x_n\}$. If during a particular reduction all $n - 1$ equations of the form

$$f(x', x'') = f(x'', x') \tag{1}$$

hold, does it follow that f is commutative on $\{x_1, \ldots, x_n\}$? The answer is "no," even if f is associative. For example, let f be matrix multiplication and let

$$x_1 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}, \quad \text{and} \quad y = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}.$$

Then (1) is satisfied throughout the left-to-right reduction of $\langle x_1, x_2, x_3 \rangle$, since each of $f(x_1, x_2)$, $f(x_2, x_1)$, $f(f(x_1, x_2), x_3)$, and $f(x_3, f(x_1, x_2))$ is the zero matrix. In contrast, $f(x_1, f(x_3, x_2))$ is the non-zero matrix y, since $f(x_3, x_2) = y$.

A similar negative result holds for associativity. Suppose during a particular reduction all $n - 2$ equations of the form

$$f(x', f(x'', x''')) = f(f(x', x''), x''') \tag{2}$$

hold. Then f is not necessarily associative on $\{x_1, \ldots, x_n\}$ even if f is a commutative, straight-line function, such as

$$f(x, y) = (2(x + y) - 1) \; div \; 3,$$

where *div* denotes integer division. The left-to-right reduction of $\langle 1, 1, 2, 3 \rangle$ using that particular f satisfies (2) at each step and yields the final answer 2, whereas the right-to-left reduction yields the final answer 1.

Now we no longer assume that $\{x_1, \ldots, x_n\}$ is closed under f. In contrast to the preceding negative results, we could obtain an automatic run-time verification that the associativity properties of f are adequate for obtaining reductions using f, if the run-time system could check that f satisfies the following property for each pair of parameter values (x, y) passed to f at run-time:

$$\forall w \in X : f(w, f(x, y)) = f(f(w, x), y). \tag{3}$$

This is because any given f-reduction of $\langle x_1, \ldots, x_n \rangle$ equals the left-to-right f-reduction. To see this, consider successive applications of the part of the equality in (3) that goes from left to right, using recursion and the invariant that each (x, y) considered is indeed a pair of parameter values passed to f in the given f-reduction.

An efficient decision method for (3), one that avoids such inefficiencies as exhaustive checking, would yield an efficient, parallel, run-time test of associativity. But, in general, (3) is undecidable. That is, no algorithm can input (f, x, y), where f is any bounded-loop program and (x, y) is any member of the domain of f, and correctly decide (3). To see this, let $(x, y) = (-1, 1)$ and use the construction in the first paragraph of the proof of Theorem 3.2.

One can similarly obtain an analogous limitative result for commutativity, in which one of the two variables in the commutativity equation is left free.

Remark 5.1. Related computational complexity results are known, based on the additional assumption that the bounded-loop program for f is a multiplication table. If a *P-complete* decision problem is also in NC, then all problems solvable on a single processor in polynomial time are in NC. (A decision problem is in NC if it is solvable in polylogarithmic time on a polynomial number of processors.) The decision problem GEN is defined as follows: given a set X, a subset $A \subseteq X$, and an element $x \in X$; determine whether or not x is contained in the smallest subset of X that includes A and is closed under f. Here are three known results: The GEN problem is P-complete. The GEN problem is P-complete even if f is commutative and A is a singleton set. If f is associative, the GEN problem is in NC; in fact it is NLOG complete. For further details and precise definitions, see Jones and Laaser [1977] and Greenlaw et al. [1995].

6. LIMITATIONS ON PROGRAMMER-CONSTRUCTED RUN-TIME CHECKS

Our final theorem shows that, in general, it is even impossible for a programmer to construct a boolean function subprogram that the programmer-defined f can use, at the beginning of its work, to carry out run-time checks of (3).

This is a particularly strong limitation, since *one is permitted to use knowledge of the specific properties of f in programming such a boolean function subprogram.* (In contrast, there is no analogous strong limitation related to the halting problem. For any given input-free program P there is indeed a boolean function subprogram that indicates whether or not P halts: it is either the function that always returns true or the function that always returns false. The unsolvability of the halting problem only implies that no *algorithm* can take any given P and determine which of the two boolean function subprograms is appropriate for P.)

THEOREM 6.1. *There exists a bounded-loop program f such that the following problem is undecidable: Given a pair of parameter values x and y, does the following hold?*

$$\forall w \in X : f(w, f(x, y)) = f(f(w, x), y).$$

PROOF. We use a coding of all input-free programs having the property that (i) each natural number n is the code of an input-free program P_n and (ii) there is a bounded-loop program for obtaining the program P_n from its code n. Define the code n of a program P to be the position of P in a list of all input-free programs, in which shorter programs precede longer programs, using the fact that input-free is a syntactical concept and the observation that $max(n, 62)$ is an upper bound on the length of P in characters. (The number of input-free programs of length n exceeds n, for programs having 62 or more characters. For instance, the order of declaration of the variables in

program P; **var** x1, x2, x3, x4, x5: Integer; **begin** x1 := 0 **end**.

```
function f(x, y: TripleType): TripleType;
  var HaltFound: Boolean;
      i, Count, Max: Integer;
      z: TripleType;
begin
  HaltFound := False; Count := 0;
  if x.TraceMe then
    begin z := x; Max := y.Max; Count := Count + 1 end;
  if y.TraceMe then
    begin z := y; Max := x.Max; Count := Count + 1 end;
  if Count = 1 then
    {Exactly one of x and y request a trace, z equals the value of the parameter
     requesting a trace, and Max equals the other parameter's Max value.}
    begin
      ... Prepare to begin tracing the program having code z.Code.
      for i := 1 to Max do
          if not HaltFound then
            begin
              ... Trace the execution of next step of the program having code z.Code.
              ... If that step causes that program to halt, assign True to HaltFound.
            end
    end;
  if HaltFound then
    begin z.TraceMe := False; z.Max := x.Max + y.Max; f := z end
  else
    f := Zero
end;
```

Fig. 2. The function f used in the proof of Theorem 6.1.

can be permuted to obtain 120 input-free programs, each of length 62; the given program contains 62 characters and we declare 5 variables to ensure that the number of such permutations exceeds the program length.)

Let $\#P$ denote the code of P. Define a type *TripleType* that consists of records of the form (*Code*, *TraceMe*, *Max*), where the integer field *Code* can store the code of a program, the boolean field *TraceMe* can indicate whether a trace of that program is requested, and the integer field *Max* can store a suggested number of steps for tracing such a program. Let P_1 be an input-free program that halts in its first step and let *Zero* be the constant with value $(\#P_1, \mathit{False}, 0)$.

Let f be the bounded-loop program in Figure 2 and suppose there is a decision method for determining whether or not

$$\forall w \in X : f(w, f(x, y)) = f(f(w, x), y).$$

We show this implies a decision method for the input-free halting problem. Let P be any input-free program and let x_P have the value $(\#P, \mathit{True}, 0)$. We claim that P fails to halt if and only if

$$\forall w \in X : f(w, f(x_P, x_P)) = f(f(w, x_P), x_P). \tag{4}$$

To see this, note that if P fails to halt, then both $f(x_P, x_P)$ and $f(w, x_P)$ equal *Zero* so both sides of the equation equal *Zero* and (4) holds. On the other hand,

suppose P halts, let n be the number of steps in the execution of P, where $n > 0$, and let $w_P = (\#P, False, n)$. Then (4) does not hold, since

$$\begin{aligned} f(w_P,\, f(x_P, x_P)) = f(w_P, Zero) &= Zero \\ &\neq (\#P, False, n) \\ &= f((\#P, False, n), (\#P, True, 0)) \\ &= f((\#P, False, n), x_P) \\ &= f(f((\#P, False, n), (\#P, True, 0)), x_P) \\ &= f(f(w_P, x_P), x_P). \end{aligned}$$

□

We note the following. The function constructed in this proof is commutative. The predicate given in the statement of this theorem is not even recursively enumerable, by an argument similar to the second half of the proof of Theorem 3.2. Theorem 3.2 is not a corollary of Theorem 6.1: it is possible for a compiler to show that the f in Figure 2 is not associative by performing four calculations using f. (It could let an input-free program P_1 that halts in its first step play the role of P in the part of the above proof showing that (4) fails to hold when P halts.)

The undecidability result analogous to Theorem 6.1 for commutativity (in which one variable is left free) also holds. To see this, define f so it uses subtraction rather than addition in calculating the value of $z.Max$ to return when *HaltFound* holds.

7. CONCLUSIONS

Previous research demonstrates that when reductions and scans are used in conjunction with selection and broadcasting, the result is an elegant, yet powerful model of parallel computation. The full power of such a model employs the use of programmer-defined function subprograms. Such function subprograms must satisfy an associativity property, and in some situations, a commutativity property as well.

Our limitative results show that, in general, assurance of such properties cannot be provided by fully automatic tools, such as compilers. In addition, neither straightforward construction from a base set of functions nor certain compiler-generated run-time checks provide a satisfactory, general approach. Even certain programmer-constructed run-time checks do not suffice.

Our undecidability results suggest the need for developing software tools to assist programmers with creating associative functions and/or proofs of associativity, even if those tools lack full generality.

ACKNOWLEDGMENTS

Dennis Ritchie helped clarify his work with A. R. Meyer on bounded-loop programs. Each of the three anonymous referees provided helpful suggestions.

REFERENCES

AKL, S. G. AND STOJMENOVIC, I. 1996. Broadcasting with selective reduction: A powerful model of parallel computation. In *Parallel and Distributed Computing Handbook*, E. Y. H. Zomaya, Ed. McGraw-Hill, New York, 192–222.

BALA, V., BRUCK, J., CYPHER, R., ELUSTONDO, P., HO, A., HO, C., KIPNIS, S., AND SNIR, M. 1995. CCL: A portable and tunable collective communication for scalable parallel computers. *IEEE Trans. Paral. Syst. 6*, 2, 154–164.

GREENLAW, R., HOOVER, H. J., AND RUZZO, W. L. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York.

HILLIS, W. D. AND STEELE JR., G. L. 1986. Data parallel algorithms. *Commun. ACM 29*, 12 (Dec.), 1170–1183.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass.

IBM. 1994. *IBM AIX Parallel Environment Parallel Subroutine Reference*, Release 2.1 ed. IBM, Poughkeepsie, New York.

INTEL. 1994. *Paragon System C System Calls Reference Manual*. Intel, Beaverton, Ore.

JONES, N. D. AND LAASER, W. T. 1977. Complete problems for deterministic polynomial time. *Theoreti. Comput. Sci. 3*, 1, 105–117.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. 1985. The power of parallel prefix. *IEEE Trans. Comput. C-34*, 10 (Oct.), 965–968.

MEYER, A. R. AND RITCHIE, D. M. 1967a. The complexity of loop programs. In *Proceedings of the 22nd ACM National Meeting*.

MEYER, A. R. AND RITCHIE, D. M. 1967b. Computational complexity and program structure. IBM Research Report RC-1817, Yorktown Heights, NY.

PARASOFT. 1990. *Express C User's Guide, Version 3.0*. Parasoft, Pasadena, Calif.

PHILLIPS, I. C. C. 1992. Recursion theory. In *Handbook of Logic in Computer Science*. Volume 1 *Background: Mathematical Structures*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, New York, 79–187.

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5, 1467–1471.

RAMALINGAM, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22*, 2, 416–430.

REPS, T. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst. 22*, 1, 162–186.

ROGERS, H. 1987. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, Mass.

SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. 1996. *MPI: The Complete Reference*. MIT Press, Cambridge, Mass.

Received July 2001; revised February 2002; accepted April 2002