Commutativity Analysis: A New Technique for Automatically Parallelizing Serial Programs

Martin C. Rinard (martin@cs.ucsb.edu)
Pedro Diniz (pedro@cs.ucsb.edu)
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

POPL95 Topic: Program Analysis, Parallelization

Abstract

This paper introduces a new analysis technique, commutativity analysis, for automatically parallelizing programs written in sequential, imperative programming languages. Commutativity analysis aggregates both data and computation into larger grain units. It then analyzes the computation at this granularity to discover when pieces of the computation commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. This approach differs from standard approaches based on data dependence analysis in that it generates parallel programs that may violate the data dependences of the original serial program. Commutativity analysis can therefore automatically parallelize programs that are inherently beyond the reach of any compiler that preserves the data dependences. In this paper we formally define a set of conditions that the compiler can use to automatically detect commuting operations, prove that if operations meet these conditions then their executions commute and show how to exploit the commutativity information to automatically generate parallel code.

1 Introduction

Current parallelizing compilers preserve the semantics of the original serial program by preserving the data dependences[1, 8]. These compilers attempt to identify independent pieces of computation (two pieces of computation are independent if neither writes a piece of data that the other accesses), then generate code that executes independent pieces concurrently. A significant limitation of this approach is the difficulty of performing dependence analysis that is precise enough to expose substantial amounts of concurrency. While researchers have been able to develop reasonably effective algorithms for loop nests that manipulate dense matrices using affine access functions[9], there has been little progress towards the successful automatic analysis of programs that manipulate pointer-based data structures. Researchers have attempted to build totally automatic systems [5], but the most promising approaches require the programmer to provide annotations that specify information about the global topology of the manipulated data structures[6].

A second, more fundamental limitation of data-dependence based approaches is an inability to parallelize computations that manipulate graph-like data structures. The aliases inherently present in these data structures preclude the static discovery of independent pieces of code, forcing the compiler to generate serial code. Finally, preserving the data dependences for programs that periodically update shared data structures can artificially limit the amount of exposed concurrency, since tasks must delay updates until they are sure that each update will not change the relative order of reads and writes to the shared data structure[2, 11].

This paper presents a new analysis technique, commutativity analysis, that eliminates many of the limitations of existing data-dependence based approaches. Instead of preserving the relative order of individual reads and writes to single words of memory, commutativity analysis aggregates both data and computation into larger grain units. It then analyzes the computation at this granularity to discover when pieces of the computation commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. While the resulting parallel program may violate the data dependences of the original serial program, it is still guaranteed to generate the same result. We expect this approach to be most useful for computations whose parallel tasks periodically generate commuting updates to shared data structures and/or traverse recursive data structures such as lists, trees and graphs. It is impossible to effectively parallelize many of these computations without violating the data dependences of the original serial program. Commutativity analysis therefore enables the automatic parallelization of programs that are inherently beyond the reach of any currently proposed technique.

The rest of the paper is structured as follows. In Section 2 we present an example that illustrates how commutativity analysis can automatically parallelize complex graph traversals. In Section 3 we describe the basic approach and informally state the conditions that the compiler uses to recognize commuting operations. In Section 4 we survey related work. In Section 5 we provide a formal treatment of commutativity analysis. In Section 5.1 we present an operational semantics of parallel and serial execution. We then present a formal definition of commutativity in this model, and state a theorem establishing that the parallel and serial executions generate the same result if all of the invoked operations commute. In Section 5.3 we formally state a set of statically checkable conditions that a compiler can use to recognize commuting operations and prove that operations commute if they satisfy the conditions. In Section 5.4 we show how to generate a conservative approximation to the set of all operations invoked as a result of executing a given operation. In Section 5.5 we present an algorithm that checks if all of these operations commute, and outline how to generate parallel code if they do.

2 A Programming Example

We show how commutativity analysis can parallelize complex traversals by discussing the automatic parallelization of the marked graph traversal algorithm in Figure 1. The algorithm is written in C++.

```
class graph { int mark; int val; graph *left; graph *right; }

graph::traverse(int f, int m) {
  graph *l; graph *r;
  val = (~val) & f;
  if (mark != m) {
    mark = m;
    if (left == right) left = NULL;
    l = left; r = right;
    if (l != NULL) l->traverse(f, m);
    if (r != NULL) r->traverse(f, m);
  }
}
```

Figure 1: Serial Graph Traversal

The traversal may visit each node of the graph several times. At each visit the method updates the receiver's `val` instance variable, then uses the `mark` instance variable to see if it should recursively visit the nodes that the current node points to. If so, the method first ensures that the `left` and `right` instance variables do not point to the same node by nulling out the `left` instance variable if they do. It then recursively invokes the `traverse` operation on the nodes that the `left` and `right` instance variables point to.

The way to parallelize this program is to execute the two recursive `traverse` operations concurrently. But this parallelization may violate the data dependences. The serial program executes all accesses generated by the left traversal before all accesses generated by the right traversal. If the two traversals visit the same node, in the parallel execution the right traversal may visit the node before the left traversal, changing the order of reads and writes to that node. This violation of the data dependences may generate cascading changes in the overall execution of the computation. Because of the marking algorithm, a node only generates recursive calls the first time it is visited. So, if the right traversal reaches a node before the left traversal, the parallel execution may also change the order in which the overall traversal is generated.

In fact, none of these changes affect the overall result of the computation. It is possible to automatically parallelize the computation even though the resulting parallel program may generate computations that differ substantially from the original serial computation. The key property that enables the parallelization is that the parallel computation generates the same set of `traverse` operations as the serial computation and the generated `traverse` operations can execute in any order without affecting the aggregate behavior of the traversal. Given this commutativity information, the compiler can automatically generate the parallel `traverse` operation in Figure 2. The top level `traverse` operation calls the `parallel_traverse` operation, then invokes the `wait` routine, which blocks until all parallel tasks created by the current task or its descendant tasks finishes. The `parallel_traverse` operation executes the recursive calls concurrently using the `spawn` construct, which creates a new task for the execution of the new traversal. The compiler also augments the `graph` data structure with a mutual exclusion lock. The `parallel_traverse` uses this lock to ensure the atomic execution of its accesses to the receiver.

This graph traversal algorithm illustrates several aspects of the computations that commutativity analysis is designed to handle. First, the traversal manipulates a recursive, pointer-based data structure. The global topology of the data structure is irrelevant - it may be a list, a tree or a DAG, contain cycles or even change during the course of the computation. Second, it is possible to statically analyze the program and discover that all of the generated operations commute. This commutativity information enables the automatic generation of parallel code. Third, both the parallel and serial executions generate the same set of operations. Commutativity analysis is not designed for computations (such as branch and bound searches) that may generate different sets of operations in different parallel executions. Fourth, both the parallel and serial computations generate identical data structures at the end of the traversal. Computations such as fully parallel tree insertions that may generate different data structures in different parallel executions are beyond the reach of this and all other published automatic techniques.

3 The Basic Approach

Commutativity analysis is designed for programs written using a pure object-based paradigm. Such programs structure the computation as a sequence of operations on objects. Each operation consists of a receiver object, an operation name and several parameters. Each operation name identifies a method that defines the behavior of the operation; when the operation executes it invokes that method. Each object implements its state using a set of instance variables. When a method executes it can recursively invoke other operations and/or use primitive operators (such as addition and multiplication) to perform computations involving the instance variables of the receiver. We require that each method be *separable* - i.e., that every control-flow path through the method

```
class graph {
  lock _l; int mark; int val;
  graph *left; graph *right;
}

graph::traverse(int f, int m) {
  this->parallel_traverse(f, m);
  wait();
}

graph::parallel_traverse(int f, int m) {
  graph *l; graph *r;
  acquire(_l);
  val = (~val) & f;
  if (mark != m) {
    mark = m;
    if (left == right) left = NULL;
    l = left; r = right;
    release(_l);
    if (l != NULL) spawn(l->traverse(f, m));
    if (r != NULL) spawn(r->traverse(f, m));
  } else {
    release(_l);
  }
}
```

Figure 2: Parallel Graph Traversal

consist of an object section and an invocation section. The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. It is of course possible for local variables to carry values computed in the object section into the invocation section.

The foundation of the commutativity analysis algorithm is an algorithm that determines if two operations commute. Section 5.3 contains a formal definition of the conditions that the algorithm uses to check for commutativity and a proof that if the two operations and the methods that they execute meet these conditions then they commute. We next provide an informal, intuitive description of these conditions.

The first set of conditions deals with the effect on the receiver. If the two operations access different receivers, then they access disjoint pieces of data and trivially generate the same result regardless of the execution order. If the two operations access the same receiver, they must have the same combined effect on the receiver regardless of the order in which they are executed. When each operation executes, it starts with the receiver in a given state. It then updates the instance variables of the receiver, leaving the receiver in a new state. It may be the case that the two operations always leave the receiver in identical new states if they execute in the same starting receiver state. In this case their effects on the receiver commute. The compiler checks this condition by first verifying that the two operations have the same operation name and therefore execute the same method. It then analyzes the method to verify that all of the parameters involved in the calculation of the new instance variable values have the same value in both operations. If this is true then the two operations are guaranteed to leave the receiver in the same new state regardless of the execution order. In Figure 1, for example, the new values of the `val`, `mark` and `left` instance variables depend on the starting state of the receiver and on the parameters `f` and `m`. Because `f` and `m` have the same value in all of the invoked `traverse` operations, the effects of all of these operations on the receiver commute.

It is possible to extend these conditions to take advantage of the algebraic properties of the individual operators used to compute the new values of the instance variables. For example, the compiler may decide that two operations with different parameter values commute if they only add the parameter to one of the instance variables.[1] Section 5.3 provides a formal description of how these algebraic properties are integrated into the commutativity analysis algorithm.

The second set of conditions deals with the set of invoked operations. The compiler must check that the two operations together invoke the same set of operations regardless of the order in which they execute. The compiler checks this condition by considering each operation invocation site in turn, verifying that each site is independent of the execution order - in other words, generates the same set of operations no matter which one of the two original operations executes first. If the invocation site only has control or data dependences on the parameters of the operation and does not depend on the instance variables, then it is independent of the execution order because the state of the receiver does not affect the operations it generates. If the invocation site depends on the instance variables of the receiver, then the algorithm checks that all of the parameters on which it depends have the same value in both invocations. In this case the invocation site is oblivious to the difference between the two original operations and generates the same operations in both execution orders. In Figure 1, for example, both invocation sites in the `traverse` method depend on several instance variables and on the parameters `f` and `m`. Because `f` and `m` have the same value for all of the invoked operations, the invocation sites are independent of the execution order. Once the compiler can check if two operations commute, it can generate a conservative approximation to the set of operations invoked as a result of executing a given operation and see if they all commute. If so, it can generate parallel code using the strategy outlined in Section 5.5.

[1]Given the vagaries of finite precision arithmetic as implemented in modern computers, the additions of two different numbers into a running sum may not, in fact, commute. It is therefore important for the programmer be able to control whether the compiler treats adding numbers into a running sum as commuting operations.

4 Related Work

Other researchers have recognized the value of including support for commuting operations in parallel computing systems[15, 2, 14]. These systems, however, focus on exploiting commuting operations and rely on some external mechanism, typically the programmer, to specify when the operations actually commute. The techniques presented in this paper, on the other hand, automatically detect commuting operations.

Several existing compilers can recognize when a loop performs a reduction of many values into a single value[4, 10, 3]. These compilers recognize when the reduction primitive (typically addition) is associative. They then exploit this algebraic property to eliminate the data dependence associated with the serial accumulation of values into the result. The generated program computes the reduction in parallel. Commutativity analysis can also recognize when operations commute because of the algebraic properties of the employed primitives. But unlike reduction analysis, commutativity analysis does not rely solely on these properties. We present a set of general conditions that allows the compiler to recognize commuting operations even when the operations employ no primitives with special algebraic properties. Knowledge of the algebraic properties therefore enhances the effectiveness of, rather than enables, the presented techniques.

Research performed in the context of database concurrency control has shown that it is possible to take advantage of commuting operations to increase the amount of concurrency available in transaction processing systems[16]. The presented approach is based on using abstract specifications of the behavior of the operations to expose the commutativity. It is the responsibility of the programmer to ensure that the actual implementation preserves the semantics of the specification. The techniques presented in this paper, on the other hand, automatically detect commuting operations given the implementation of the operations and do not rely on higher-level specifications of their behavior.

Programmers who parallelize computations by hand have always used commuting operations. For example, four (Water, MP3D, LocusRoute and Cholesky) of the six parallel applications in the SPLASH benchmark suite [13] and three of the four parallel applications described in [12] violate the data dependences of the original serial program and rely on commuting operations for their correct execution. While the techniques presented in this paper fail to recognize many of the commuting operations in these programs, this applications experience does show how important it is to support commuting operations.

5 Formal Treatment

We next provide a formal treatment of how commutativity enables parallel execution. We first present a formal model of computation for separable operations. We define parallel and serial execution in this model, and define what it means for two operations to commute. The key theorem establishes that if all of the operations in the parallel executions commute, then all parallel executions are equivalent to the serial execution.

Section 5.3 presents commutativity analysis. This analysis uses two kinds of information: data and control dependence information and information about the parameter and receiver values of invoked operations. We show how to extract this information using a form of symbolic execution. The key theorems establish the consistency of the analysis and establish that certain conditions on methods guarantee that the corresponding operations commute. We finally show how to combine the information gathered during the analysis to test if it is possible to parallelize the execution.

5.1 The Model of Computation

We next describe a formal model of computation for separable programs. We first assume a set of objects O, with $r, o \in O$. O contains an error object `err` and the true object `true`. We convert O into a flat lattice by adding a top object $\top$ and a bottom object $\bot$, with least upper bound $\wedge$ defined by $\top \wedge o = o \wedge \top = \top$,

$\perp \wedge o = o \wedge \perp = o$, $o \wedge o = o$ and $o \wedge r = r \wedge o = \top$ if $o \neq r, o \neq \perp$ and $r \neq \perp$. For O, as for all other lattices in this paper, the $\wedge$ operator induces a partial order $\sqsubseteq$ defined by $o \sqsubseteq r$ if $o \wedge r = r$. The lattice structure, of course, does not affect actual computations because no computation generates $\top$ or $\perp$. We impose the additional structure to simplify the proof of correspondence with the symbolic analysis in Section 5.3.

We also assume a set of constants $\mathtt{c} \in \mathtt{C}$, a set of variables $\mathtt{v} \in \mathtt{V}$ and a set of operation names $\mathtt{OP}$ with $\mathtt{op} \in \mathtt{OP}$. Each object o has a set of instance variables $iv(o) \subseteq \mathtt{V}$. We assume that the program is strongly typed and extend iv to $\mathtt{OP}$, with $iv(\mathtt{op}) \subseteq \mathtt{V}$ the set of instance variables of the receiver objects of $\mathtt{op}$ operations.

The operational semantics also uses several functions. The first function is the object memory $m : O \times \mathtt{V} \rightarrow O, m \in M$, where $m(o, \mathtt{v})$ is the value of instance variable $\mathtt{v}$ of object o. The object $\wedge$ operator induces a $\wedge$ operator on M: $m = m_1 \wedge m_2$ if $\forall o, \mathtt{v} : m(o, \mathtt{v}) = m_1(o, \mathtt{v}) \wedge m_2(o, \mathtt{v})$. We require that $\wedge$ distribute through m: $m(r, \mathtt{v}) \wedge m(o, \mathtt{v}) = m(r \wedge o, \mathtt{v})$. We also have applied operations $a \in A = O \times \mathtt{OP} \times O$, and write an element a in the form $r\texttt{->}\mathtt{op}(o)$. Each applied operation has a receiver r, an operation name $\mathtt{op}$ and a parameter o. To simplify the presentation we assume that each operation takes only one parameter, but the framework generalizes in a straightforward way to include operations with multiple parameters. The object $\wedge$ operator also induces a $\wedge$ operator on subsets of A with the same operation name $\mathtt{op}$: $r_1\texttt{->}\mathtt{op}(o_1) \wedge r_2\texttt{->}\mathtt{op}(o_2) = r_1 \wedge r_2\texttt{->}\mathtt{op}(o_1 \wedge o_2)$.

We next describe how we model computation. Each invoked operation first updates the receiver, then invokes several other operations. We model the effect on the receiver with a function $f : M \times A \times \mathtt{V} \rightarrow O, f \in F$. $f(m, r\texttt{->}\mathtt{op}(o), \mathtt{v})$ is the new value of the instance variable $\mathtt{v}$ of the receiver r when the operation $r\texttt{->}\mathtt{op}(o)$ executes in the memory m. f must be monotone in both m and $r\texttt{->}\mathtt{op}(o)$ (Equation 1) and the new value of each variable should depend on the state of no object except the receiver (Equation 2).

$$m_1 \sqsubseteq m_2 \text{ and } a_1 \sqsubseteq a_2 \text{ implies } \forall \mathtt{v} : f(m_1, a_1, \mathtt{v}) \sqsubseteq f(m_2, a_2, \mathtt{v}) \tag{1}$$

$$\forall \mathtt{v} \in \mathtt{V} : m_1(r, \mathtt{v}) = m_2(r, \mathtt{v}) \text{ implies } \forall \mathtt{v} \in \mathtt{V} : f(m_1, r\texttt{->}\mathtt{op}(o), \mathtt{v}) = f(m_2, r\texttt{->}\mathtt{op}(o), \mathtt{v}) \tag{2}$$

We model the invoked operations using sequences $s \in seq(A)$ of the form $a_1 \circ \cdots \circ a_k$. ϵ is the empty sequence, and $\epsilon \circ s = s \circ \epsilon = s$. We use the applied object partial order $\sqsubseteq$ to induce a corresponding partial order $\sqsubseteq$ on $seq(A)$: $a_1 \circ \cdots \circ a_k \sqsubseteq a'_1 \circ \cdots \circ a'_{k'}$ if there exists a strictly increasing, one to one function $h : [1, k] \rightarrow [1, k']$ such that $\forall i \in [1, k] : a_i \sqsubseteq a'_{h(i)}$. We use a function $g : M \times A \rightarrow seq(A), g \in G$ to model the invoked operations. $g(m, r\texttt{->}\mathtt{op}(o))$ is the sequence of operations invoked when $r\texttt{->}\mathtt{op}(o)$ executes in memory m. As for f, g must be monotone in both m and $r\texttt{->}\mathtt{op}(o)$ (Equation 3) and the sequence of invoked operations should depend on the state of no object except the receiver (Equation 4).

$$m_1 \sqsubseteq m_2 \text{ and } a_1 \sqsubseteq a_2 \text{ implies } g(m_1, a_1) \sqsubseteq g(m_2, a_2) \tag{3}$$

$$\forall \mathtt{v} \in \mathtt{V} : m_1(r, \mathtt{v}) = m_2(r, \mathtt{v}) \text{ implies } g(m_1, r\texttt{->}\mathtt{op}(o)) = g(m_2, r\texttt{->}\mathtt{op}(o)) \tag{4}$$

The serial operational semantics of the program uses a transition function $\rightarrow$ on serial states $\langle m, s \rangle \in M \times seq(A)$, where m is the current memory and s is the sequence of operations left to invoke. $\rightarrow$ models the execution of the next operation in s. As Definition 1 illustrates, $\rightarrow$ updates the memory to reflect the effect of the executed operation on the receiver. It also removes the executed operation from the current sequence of operations left to invoke and prepends the set of operations that the executed operation invokes. The operations therefore execute in the standard depth-first execution order. Strictly speaking, $\rightarrow$ depends on f and g, but we do not represent this dependence explicitly as the correct f and g are always obvious from context.

Definition 1

$$\frac{s' = g(m, r\texttt{->}\mathtt{op}(o)) \circ s, m'(o', \mathtt{v}) = \begin{cases} f(m, r\texttt{->}\mathtt{op}(o), \mathtt{v}) & \textit{if } \mathtt{v} \in iv(r) \textit{ and } o' = r \\ m(o', \mathtt{v}) & \textit{otherwise} \end{cases}}{\langle m, r\texttt{->}\mathtt{op}(o) \circ s \rangle \rightarrow \langle m', s' \rangle}$$

The states $\langle m, p \rangle$ in the parallel operational semantics are similar to the states in the serial operational semantics, but have a multiset p, rather than a sequence s, of operations left to invoke. Such multisets $p \in \mathit{mst}(A)$ are of the form $\{a_1\} \uplus \cdots \uplus \{a_k\}$. We use the sequence to multiset function $\mathit{sp} : \mathit{seq}(A) \rightarrow \mathit{mst}(A)$ defined by $\mathit{sp}(a_1 \circ \cdots \circ a_k) = \{a_1\} \uplus \cdots \uplus \{a_k\}$ to map sequences of applied operations into the corresponding multisets.

We model parallel execution by generating all possible interleavings of the execution of the applied operations. In the parallel execution any of the current operations may execute next, with each operation taking the current state to a potentially different state than the other operations. The parallel operational semantics therefore models execution using a transition relation $\Rightarrow$ rather than a transition function. As the following definition illustrates, $\Rightarrow$ relates a state to all of the states reachable via the execution of any of the current set of operations left to invoke. The definition of $\Rightarrow$ completes the execution framework required to precisely state when two operations commute.

Definition 2

$$\frac{\langle m, r\text{->}\mathtt{op}(o) \rangle \rightarrow \langle m', s' \rangle, p' = \mathit{sp}(s') \uplus p}{\langle m, \{r\text{->}\mathtt{op}(o)\} \uplus p \rangle \Rightarrow \langle m', p' \rangle}$$

Definition 3 $r_1\text{->}\mathtt{op}_1(o_1)$ *and* $r_2\text{->}\mathtt{op}_2(o_2)$ *commute if*

$$\begin{array}{rcl} \forall m: \quad \langle m, \{r_1\text{->}\mathtt{op}_1(o_1)\}\rangle \Rightarrow \langle m_1, p_1 \rangle & \textit{and} & \langle m, \{r_2\text{->}\mathtt{op}_2(o_2)\}\rangle \Rightarrow \langle m_2, p_2 \rangle \textit{ and} \\ \langle m_1, \{r_2\text{->}\mathtt{op}_2(o_2)\}\rangle \Rightarrow \langle m_{12}, p_{12} \rangle & \textit{and} & \langle m_2, \{r_1\text{->}\mathtt{op}_1(o_1)\}\rangle \Rightarrow \langle m_{21}, p_{21} \rangle \textit{ implies} \\ m_{12} = m_{21} & \textit{and} & p_1 \uplus p_{12} = p_2 \uplus p_{21} \end{array}$$

We next present several lemmas that characterize the relationship between the parallel and serial operational semantics. To establish this relationship we must first define a notion of equivalence between different executions. Our notion of equivalence is that two computations are equivalent if they produce the same memory at the end of the computation. Lemma 1 says that one of the possible parallel executions is equivalent to the serial execution. Lemma 2 says that if all of the operations invoked in the parallel executions commute, then all of the parallel executions are equivalent. This lemma uses the function $\mathit{gen} : M \times A \rightarrow 2^A$, which tells which operations can be invoked as a result of invoking a given operation. An immediate corollary of these two lemmas is that if all of the operations invoked in the parallel executions commute, then all parallel executions are equivalent to the serial execution.

Lemma 1 *If* $\langle m, r\text{->}\mathtt{op}(o)\rangle \rightarrow \cdots \rightarrow \langle m', \epsilon \rangle$ *then* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', \emptyset \rangle$

$$\mathit{gen}(m, r\text{->}\mathtt{op}(o)) = \{r'\text{->}\mathtt{op}'(o') : \exists p' : \langle m, \{r\text{->}\mathtt{op}(o)\}\} \Rightarrow \cdots \Rightarrow \langle m', \{r'\text{->}\mathtt{op}'(o')\} \uplus p' \rangle \qquad (5)$$

Lemma 2 *If* $\forall r_1\text{->}\mathtt{op}_1(o_1), r_2\text{->}\mathtt{op}_2(o_2) \in \mathit{gen}(m, r\text{->}\mathtt{op}(o)) : r_1\text{->}\mathtt{op}_1(o_1)$ *and* $r_2\text{->}\mathtt{op}_2(o_2)$ *commute, then* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_1', \emptyset \rangle$ *and* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_2', \emptyset \rangle$ *implies* $m_1' = m_2'$

Proof Sketch: If all of the invoked operations commute then the transition system is confluent, which guarantees deterministic execution[7].

5.2 Program Semantics

We next show, given a program that defines a set of operations, how to extract the functions f and g that determine the behavior of the operations. We first define the abstract syntax for a simple object-based language:

$$
\begin{array}{lcl}
\texttt{PR} & ::= & \texttt{op(v)\{ST\}PR} \\
\texttt{PR} & ::= & \epsilon \\
\texttt{ST} & ::= & \texttt{ST;ST} \\
\texttt{ST} & ::= & \texttt{v = EX;} \\
\texttt{ST} & ::= & \epsilon \\
\texttt{ST} & ::= & \texttt{EX->op(EX);} \\
\texttt{ST} & ::= & \texttt{if(EX)\{ST\}else\{ST\}} \\
\texttt{EX} & ::= & \texttt{EX} \oplus \texttt{EX} \\
\texttt{EX} & ::= & \texttt{v} \\
\texttt{EX} & ::= & \texttt{c}
\end{array}
$$

This grammar defines the set of programs PR, with $\texttt{pr} \in \texttt{PR}$, the set of statements ST, with $\texttt{st} \in \texttt{ST}$, and the set of expressions EX with $\texttt{ex} \in \texttt{EX}$. As for O, we convert EX into a flat lattice by adding a top element $\top$ and bottom element $\bot$. EX also contains the error expression err. Following the standard convention in object-based languages, each method omits the declaration of the receiver. The method refers to the receiver using the this variable. The $\oplus$ symbol stands for an arbitrary binary function $\oplus : O \times O \to O$. For example, the object model might include the integers as (immutable) objects, with $\oplus$ implementing the integer addition function. We require that $\top \oplus o = o \oplus \top = \top$ and $\texttt{err} \oplus o = o \oplus \texttt{err} = \texttt{err}$ if $o \neq \top$. The framework generalizes in a straightforward way to include arbitrary numbers of arbitrary n-ary operators.

We next define the f and g for a given program using standard techniques from denotational semantics. The semantics uses a local variable function $l : \texttt{V} \to O, l \in L$ which gives the values of the local variables for the evaluation of expressions. We use the object $\wedge$ operator to define a $\wedge$ operator on L as follows: $l = l_1 \wedge l_2$ if $\forall \texttt{v} \in \texttt{V} : l(\texttt{v}) = l_1(\texttt{v}) \wedge l_2(\texttt{v})$. $ev(m, l, \texttt{ex})$ is the object o which is the value of the expression ex in the context of the memory m and local variable function l. $exms(\texttt{st}, m, l, s)$ is the tuple $\langle m', l', s'\rangle$ that tells how the execution of the statement st in memory m and local variable function l generates a new memory m', local variable function l' and operation sequence s'. $exfg(\texttt{pr}, f, g)$ is the tuple $\langle f', g'\rangle$ that tells how the program pr generates updated f' and g' functions.

We define *ev* inductively using the following set of axioms. *ev* is defined for expressions of the form ex.v, which stands for the instance variable v of object ex and $\texttt{if}(\texttt{ex}, \texttt{ex}_1, \texttt{ex}_2)$, which stands for the expression whose value is $\texttt{ex}_1$ if ex is true and $\texttt{ex}_2$ if ex is not true. These expressions do not appear in programs - the symbolic execution described in Section 5.3 generates them. We then use *ev* to define the statement execution function $exms : \texttt{ST} \times M \times L \times seq(A) \to M \times L \times seq(A)$.

$$ev(m, l, \texttt{err}) = \texttt{err} \quad ev(m, l, \top) = \top \quad ev(m, l, \bot) = \bot \quad ev(m, l, \texttt{c}) = \texttt{c}$$

$$ev(m, l, \texttt{ex.v}) = m(ev(m, l, \texttt{ex}), \texttt{v}) \quad ev(m, l, \texttt{ex}_1 \oplus \texttt{ex}_2) = ev(m, l, \texttt{ex}_1) \oplus ev(m, l, \texttt{ex}_2)$$

$$ev(m, l, \texttt{v}) = \begin{cases} m(l(\texttt{this}), \texttt{v}) & \text{if } \texttt{v} \in iv(l(\texttt{this})) \\ l(\texttt{v}) & \text{if } \texttt{v} \notin iv(l(\texttt{this})) \end{cases}$$

$$ev(m, l, \texttt{if}(\texttt{ex}, \texttt{ex}_1, \texttt{ex}_2)) = \begin{cases} ev(m, l, \texttt{ex}_1) & \text{if } ev(m, l, \texttt{ex}) = \texttt{true} \\ ev(m, l, \texttt{ex}_1) \wedge ev(m, l, \texttt{ex}_2) & \text{if } ev(m, l, \texttt{ex}) = \top \\ ev(m, l, \texttt{ex}_2) & \text{otherwise} \end{cases}$$

$$\frac{exms(\texttt{st}_1, m, l, s) = \langle m_1, l_1, s_1\rangle, exms(\texttt{st}_2, m_1, l_1, s_1) = \langle m', l', s'\rangle}{exms(\texttt{st}_1;\texttt{st}_2, m, l, s) = \langle m', l', s'\rangle}$$

$$\frac{\mathtt{v} \in \mathit{iv}(l(\mathtt{this})), m' = m[\mathtt{v} \mapsto \mathit{ev}(m, l, \mathtt{ex})]}{\mathit{exms}(\mathtt{v\ =\ ex;}, m, l, s) = \langle m', l, s\rangle} \qquad \frac{\mathtt{v} \notin \mathit{iv}(l(\mathtt{this})), l' = l[\mathtt{v} \mapsto \mathit{ev}(m, l, \mathtt{ex})]}{\mathit{exms}(\mathtt{v\ =\ ex;}, m, l, s) = \langle m, l', s\rangle}$$

$$\frac{\begin{array}{c}\mathit{exms}(\mathtt{st}_1, m, l, s) = \langle m_1, l_1, s_1\rangle, \mathit{exms}(\mathtt{st}_2, m, l, s) = \langle m_2, l_2, s_2\rangle, \\ \langle m', l', s'\rangle = \begin{cases} \langle m_1, l_1, s_1\rangle & \text{if } \mathit{ev}(m, l, \mathtt{ex}) = \mathtt{true} \\ \langle m_1 \wedge m_2, l_1 \wedge l_2, s_1 \circ s_2\rangle & \text{if } \mathit{ev}(m, l, \mathtt{ex}) = \top \\ \langle m_2, l_2, s_2\rangle & \text{otherwise}\end{cases}\end{array}}{\mathit{exms}(\mathtt{if(ex)\{st_1\}else\{st_2\}}, m, l, s) = \langle m', l', s'\rangle}$$

$$\frac{s' = s \circ \mathit{ev}(m, l, \mathtt{ex}_1)\texttt{->}\mathtt{op}(\mathit{ev}(m, l, \mathtt{ex}_2))}{\mathit{exms}(\mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2), m, l, s) = \langle m, l, s'\rangle} \qquad \mathit{exms}(\epsilon, m, l, s) = \langle m, l, s\rangle$$

We next define the function *exfg* : $\mathtt{PR} \times F \times G \to F \times G$, which actually extracts the f and g functions from the program. We can verify, by induction on the structure of statements, that the extracted f and g satisfy the properties of Equations 1, 2, 3 and 4.

$$\frac{\begin{array}{rcl} \multicolumn{3}{l}{\forall m \in M, r, o \in O, \mathtt{v} \in V, \mathtt{op}' \in \mathtt{OP} :} \\ \langle m', l', s'\rangle & = & \mathit{exms}(\mathtt{st}, m, l_{\mathtt{err}}[\mathtt{this} \mapsto r, \mathtt{v} \mapsto o], \epsilon) \\ f'(m, r\texttt{->}\mathtt{op}'(o), \mathtt{v}) & = & \begin{cases} m'(r, \mathtt{v}) & \text{if } \mathtt{op}' = \mathtt{op} \\ f(m, r\texttt{->}\mathtt{op}'(o), \mathtt{v}, \mathtt{v}) & \text{otherwise}\end{cases} \\ g'(m, r\texttt{->}\mathtt{op}'(o)) & = & \begin{cases} s' & \text{if } \mathtt{op}' = \mathtt{op} \\ g(m, r\texttt{->}\mathtt{op}'(o)) & \text{otherwise}\end{cases}\end{array}}{\mathit{exfg}(\mathtt{op(v)\{st\}pr}, f, g) = \mathit{exfg}(\mathtt{pr}, f', g')}$$

$$\mathit{exfg}(\epsilon, f, g) = \langle f, g\rangle$$

5.3 Symbolic Analysis

In this section we present the symbolic analysis required to determine when it is possible to parallelize the execution of a given operation. The analysis consists of three phases. The first phase extracts the dependence information required to determine when two invocations of the same operation commute. The second phase extracts a conservative, tractably computable approximation to the set of operations invoked as a result of invoking the given operation. The third phase combines the information from the first two phases to determine if all of the potential invoked operations commute. If so, it is possible to parallelize the invocation of the given operation. All of the analysis required to extract this information is based on symbolic execution of the individual methods.

For each operation and receiver instance variable the symbolic execution generates an expression that gives the new value of the instance variable after the operation executes. The free variables in this expression are `this` (the receiver), `parm` (the parameter) and `this.v` (the value of the instance variable `v` of the receiver before the operation executes).

The symbolic execution also yields a sequence $t \in \mathit{seq}(\mathtt{OX})$ of invoked operation expressions $t = \mathtt{ox}_1 \circ \cdots \circ \mathtt{ox}_k$, where the $\mathtt{ox}_i \in \mathtt{OX}$. An `ox` of the form $\mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2)$ represents an invocation of operation `op` with receiver $\mathtt{ex}_1$ and parameter $\mathtt{ex}_2$. Because an operation may be invoked along one control-flow path but not another, `OX` also includes conditional invoked operation expressions of the form $\mathtt{if(ex, ox)}$, which represents an operation invoked only if `ex` is `true`, and $\mathtt{ifnot(ex, ox)}$, which represents an operation invoked only if `ex` is not `true`. We extend *ev* to $\mathtt{OX} \cup \mathit{seq}(\mathtt{OX})$ as follows:

$$\mathit{ev}(m, l, \mathtt{if(ex, ox)})) = \begin{cases} \mathit{ev}(m, l, \mathtt{ox}) & \text{if } \mathtt{true} \sqsubseteq \mathit{ev}(m, l, \mathtt{ex}) \\ \epsilon & \text{otherwise}\end{cases}$$

$$ev(m, l, \texttt{ifnot}(\texttt{ex}, \texttt{ox}))) = \begin{cases} \epsilon & \text{if } \texttt{true} = ev(m, l, \texttt{ex}) \\ ev(m, l, \texttt{ox}) & \text{otherwise} \end{cases}$$

$$ev(m, l, \texttt{ex}_1\texttt{->op}(\texttt{ex}_2) = ev(m, l, \texttt{ex}_1)\texttt{->op}(ev(m, l, \texttt{ex}_2))$$

$$ev(m, l, \texttt{ox}_1 \circ \texttt{ox}_2) = ev(m, l, \texttt{ox}_1) \circ ev(m, l, \texttt{ox}_2)$$

The symbolic execution uses several functions. There is a binding function $b : \texttt{V} \to \texttt{EX}, b \in B$ to implement the bindings of variables to expressions during the symbolic execution. There is also a symbolic expression evaluation function $eb : B \times \texttt{EX} \to \texttt{EX}, eb \in EB$ which takes a set of variable bindings and an expression and symbolically evaluates the expression using the bindings, generating a new expression.

$$eb(b, \texttt{c}) = \texttt{c} \quad eb(b, \texttt{err}) = \texttt{err} \quad eb(b, \top) = \top \quad eb(b, \bot) = \bot$$

$$eb(b, \texttt{v}) = b(\texttt{v}) \quad eb(b, \texttt{ex}_1 \oplus \texttt{ex}_2) = eb(b, \texttt{ex}_1) \oplus eb(b, \texttt{ex}_2)$$

We next define the *exst* function, which symbolically executes a statement, yielding a new binding function and sequence of invoked operation expressions. We define *exst* inductively on the structure of statements. To execute each statement, these rules typically execute the components of the statement, then combine the results to generate the final symbolic execution. The rule for an `if` construct, for example, symbolically executes both branches of the `if`. It combines the two binding functions by generating a conditional expression when the functions do not agree on a binding. It combines the sequences of invoked operations by generating the appropriate conditional operation expressions and concatenating them. So, the invoked operations from both branches of the `if` appear.

$$\frac{exst(\texttt{st}_1, b, t) = \langle b_1, t_1 \rangle}{exst(\texttt{st}_1\texttt{;st}_2, b, t) = exst(\texttt{st}_2, b_1, t_1)} \qquad \frac{b' = b[\texttt{v} \mapsto eb(b, \texttt{ex})]}{exst(\texttt{v = ex;}, b, t) = \langle b', t \rangle}$$

$$\frac{t' = t \circ eb(b, \texttt{ex}_1)\texttt{->op}(eb(b, \texttt{ex}_2))}{exst(\texttt{ex}_1\texttt{->op}(\texttt{ex}_2)\texttt{;}, b, t) = \langle t', b \rangle} \qquad exst(\epsilon, b, t) = \langle b, t \rangle$$

$$\frac{\begin{array}{c} \langle b_1, \texttt{ox}_1^1 \circ \cdots \circ \texttt{ox}_1^j \rangle = exst(\texttt{st}_1, b, \epsilon), \langle b_2, \texttt{ox}_2^1 \circ \cdots \circ \texttt{ox}_2^k \rangle = exst(\texttt{st}_2, b, \epsilon), \\ t' = t \circ \texttt{if}(\texttt{ex}', \texttt{ox}_1^1) \circ \cdots \circ \texttt{if}(\texttt{ex}', \texttt{ox}_1^j) \circ \texttt{ifnot}(\texttt{ex}', \texttt{ox}_2^1) \circ \cdots \circ \texttt{ifnot}(\texttt{ex}', \texttt{ox}_2^k), \\ \texttt{ex}' = eb(b, \texttt{ex}), b'(\texttt{v}) = \begin{cases} b_1(\texttt{v}) & \text{if } b_1(\texttt{v}) = b_2(\texttt{v}) \\ \texttt{if}(eb(b, \texttt{ex}), eb(b_1, \texttt{v}), eb(b_2, \texttt{v})) & \text{if } b_1(\texttt{v}) \neq b_2(\texttt{v}) \end{cases} \end{array}}{exst(\texttt{if}(\texttt{ex})\{\texttt{st}_1\}\texttt{else}\{\texttt{st}_2\}, b, t) = \langle b', t' \rangle}$$

We next present the *expr* function, which uses the *exst* function to symbolically execute each method. *expr* returns the results of the execution in two functions. The first function is $opt : \texttt{OP} \to seq(\texttt{OX})$, which records, for each operation, the generated sequence of operation expressions. The second function is $opb : \texttt{OP} \to \texttt{V} \to \texttt{EX}$. $opb(\texttt{op})(\texttt{v})$ is the symbolic expression that gives the new value of instance variable `v` of the receiver when `op` is executed. Our definition of *expr* uses the following auxiliary function: $b_{\texttt{op}}(\texttt{v}) = \texttt{this.v}$ if $\texttt{v} \in iv(\texttt{op})$, $b_{\texttt{op}}(\texttt{v}) = \texttt{err}$ otherwise.

$$\frac{\langle b, s \rangle = exst(\texttt{st}, b_{\texttt{op}}[\texttt{this} \mapsto \texttt{this}, \texttt{v} \mapsto \texttt{parm}], \epsilon), opt' = opt[\texttt{op} \mapsto s], opb' = opb[\texttt{op} \mapsto b]}{expr(\texttt{op}(\texttt{v})\{\texttt{st}\}\texttt{pr}, opb, opt) = expr(\texttt{pr}, opb', opt')}$$

$$expr(\epsilon, opb, opt) = \langle opb, opt \rangle$$

We next establish the connection between the symbolic analysis and the operational semantics. We start with several lemmas, which are readily proved via structural induction. We then state Theorem 1, which establishes that the symbolic execution accurately reflects the actual execution.

Lemma 3 *If* $\forall \mathtt{v} : ev(m, l, \mathtt{v}) = ev(m_0, l_0, b(\mathtt{v})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2])$ *then* $\forall \mathtt{ex} : ev(m, l, \mathtt{ex}) = ev(m_0, l_0, eb(b, \mathtt{ex})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2])$.

Lemma 4

$$\begin{array}{lrcl}
\textit{If} & s & = & ev(m_0, l_0, t[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2]) \textit{ and} \\
& \forall \mathtt{v} : ev(m, l, \mathtt{v}) & = & ev(m_0, l_0, b(\mathtt{v})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2]) \textit{ and} \\
& \langle m', l', s' \rangle & = & exms(\mathtt{st}, m, l, s) \textit{ and } \langle b', t' \rangle = exst(\mathtt{st}, b, t) \\
\textit{Then} & s' & = & ev(m_0, l_0, t'[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2]) \textit{ and} \\
& \forall \mathtt{v} : ev(m', l', \mathtt{v}) & = & ev(m_0, l_0, b'(\mathtt{v})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2])
\end{array}$$

Theorem 1 *If* $\langle f, g \rangle = exfg(\mathtt{pr}, \emptyset, \emptyset)$, $\langle opb, opt \rangle = expr(pr, \emptyset, \emptyset)$, *and* $ev(m, l_0, \mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2)) = r\texttt{->}\mathtt{op}(o)$ *then* $g(m, r\texttt{->}\mathtt{op}(o)) = ev(m, l_0, opt(\mathtt{op})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2])$ *and* $\forall \mathtt{v} : f(m, r\texttt{->}\mathtt{op}(o), \mathtt{v}) = ev(m, l_0, opb(\mathtt{op})(\mathtt{v})[\mathtt{this}/\mathtt{ex}_1, \mathtt{parm}/\mathtt{ex}_2])$

We next state the conditions that the method must satisfy for two invocations of the corresponding operation to commute. These conditions are stated in terms of the data and control dependences that affect the new values of the receiver's instance variables and the operations that the execution invokes. We use the function $dep : \mathtt{EX} \cup \mathtt{OX} \rightarrow 2^{\mathtt{V}}$ to extract the dependence information. If $\mathtt{v} \in dep(\mathtt{ex})$ then the value of `ex` depends on the initial value of the instance variable `v` of the receiver. If $\mathtt{parm} \in dep(\mathtt{ex})$ then the value of `ex` depends on the parameter. The definition of *dep* allows us to state the key theorem that establishes the dependence conditions that the method must satisfy for the operations to commute.

$$dep(\mathtt{if}(\mathtt{ex}_1, \mathtt{ex}_2, \mathtt{ex}_3)) = dep(\mathtt{ex}_1) \cup dep(\mathtt{ex}_2) \cup dep(\mathtt{ex}_3)$$

$$dep(\mathtt{parm}) = \{\mathtt{parm}\} \quad dep(\mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2)) = dep(\mathtt{ex}_1) \cup dep(\mathtt{ex}_2)$$

$$dep(\mathtt{this.v}) = \{\mathtt{v}\} \quad dep(\mathtt{ex}_1 \oplus \mathtt{ex}_2) = dep(\mathtt{ex}_1) \cup dep(\mathtt{ex}_2)$$

$$dep(\mathtt{if}(\mathtt{ex}, \mathtt{ox})) = dep(\mathtt{ex}) \cup dep(\mathtt{ox}) \quad dep(\mathtt{ifnot}(\mathtt{ex}, \mathtt{ox})) = dep(\mathtt{ex}) \cup dep(\mathtt{ox})$$

Theorem 2 *If* $\langle f, g \rangle = exfg(\mathtt{pr}, \emptyset, \emptyset)$ *and* $\langle opb, opt \rangle = expr(pr, \emptyset, \emptyset)$ *then* $r_1\texttt{->}\mathtt{op}_1(o_1)$ *and* $r_2\texttt{->}\mathtt{op}_1(o_2)$ *commute if either* $r_1 \neq r_2$, *or* $\mathtt{op}_1 = \mathtt{op}_2 = \mathtt{op}$, $\forall \mathtt{v} \in iv(\mathtt{op}) : \mathtt{parm} \in dep(opb(\mathtt{op})(\mathtt{v}))$ *implies* $o_1 = o_2$ *and* $\forall \mathtt{ox} \in sp(opt(\mathtt{op})) : \{\mathtt{parm}\} = dep(\mathtt{ox})$ *or* $\mathtt{parm} \in dep(\mathtt{ox})$ *implies* $o_1 = o_2$.

It is possible to integrate the exploitation of algebraic operator properties into this commutativity analysis framework. For example, if the operator $\oplus$ is commutative and associative, we can further relax the commutativity conditions above to allow the operations to commute if each variable `v` either satisfies the conditions in Theorem 2 or if $opb(\mathtt{op})(\mathtt{v}) = \mathtt{this.v} \oplus \mathtt{ex}$ and $\forall \mathtt{v}' \in iv(\mathtt{op}) : \mathtt{v}' \neq \mathtt{v}$ implies $\mathtt{v} \notin dep(opb(\mathtt{op})(\mathtt{v}'))$.

5.4 Invoked Operations

We next show how to compute a conservative approximation to the set of operations that can be invoked as a result of executing a given operation. Because the conditions in Theorem 2 depend on the values of the parameters, the analysis keeps track of the receiver and parameter values. The analysis starts with the invocation of one operation and traces out all the possible operation invocations in a breadth-first manner, recording the symbolic receiver and parameter values. We render the analysis tractable by recording one receiver value and one parameter value for each operation. If the operation is invoked more than once with different receiver or parameter values, the recorded value is $\top$, which represents the possibility that the operation may be invoked with multiple different receiver or parameter values. Eventually the analysis traces out all possible invocations and the symbolic execution process reaches a fixed point. The algorithm has then computed a conservative approximation of the set of invoked operations and their receiver and parameter values.

We represent the set of symbolic invoked operations using an invocation function $n : \mathtt{OP} \rightarrow \mathtt{EX} \times \mathtt{OP} \times \mathtt{EX}$. $n(\mathtt{op})$ tells the symbolic receiver and parameter values for op. If op has not been invoked, $n(\mathtt{op}) = \bot\texttt{->}\mathtt{op}(\bot)$. The initial invocation function $n_\bot$ is defined by $n_\bot(\mathtt{op}) = \bot\texttt{->}\mathtt{op}(\bot)$. The transition function $\triangleright$ on invocation functions computes the next set of symbolic invoked operations given a current set of symbolic invocations.

$$\frac{n'(\mathtt{op}) = \bigwedge\{\mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2) \in \mathit{sp}(\mathit{exop}(\mathit{opt}(\mathtt{op}')))[\mathtt{this}/\mathtt{ex}_1', \mathtt{parm}/\mathtt{ex}_2']) : n(\mathtt{op}') \neq \bot\texttt{->}\mathtt{op}(\bot)\}}{n \ \triangleright n'}$$

$\triangleright$ uses the function *exop*, which extracts a symbolic invocation from a conditional symbolic invocation expression. *exop* also propagates $\top$ values up expressions, generating a symbolic value of $\top$ for any expression with a component whose symbolic value is $\top$. *exop* also converts all instance variable accesses to $\top$. Because instance variables can be updated during the course of the computation, an expression that contains an instance variable reference can represent different values at different points in the computation. Converting all expressions containing instance variable references to $\top$ eliminates the resulting potential problem of the same symbolic expression representing different values. As Lemma 6 illustrates, converting all instance variable references to $\top$ is overly conservative, and it is possible to use more precise analysis techniques without going outside the basic paradigm.

$$\mathit{exop}(\mathtt{if}(\mathtt{ex},\mathtt{ox})) = \mathit{exop}(\mathtt{ox}) \quad \mathit{exop}(\mathtt{ifnot}(\mathtt{ex},\mathtt{ox})) = \mathit{exop}(\mathtt{ox})$$

$$\mathit{exop}(\mathtt{ex}_1\texttt{->}\mathtt{op}(\mathtt{ex}_2)) = \mathit{exop}(\mathtt{ex}_1)\texttt{->}\mathtt{op}(\mathit{exop}(\mathtt{ex}_2)) \quad \mathit{exop}(\mathtt{ox}_1 \circ \mathtt{ox}_2) = \mathit{exop}(\mathtt{ox}_1) \circ \mathit{exop}(\mathtt{ox}_2)$$

$$\mathit{exop}(\mathtt{v}) = \mathtt{v} \quad \mathit{exop}(\mathtt{c}) = \mathtt{c} \quad \mathit{exop}(\mathtt{ex}_1 \oplus \mathtt{ex}_2) = \begin{cases} \top \text{ if } \mathit{exop}(\mathtt{ex}_1) = \top \text{ or } \mathit{exop}(\mathtt{ex}_2) = \top \\ \mathit{exop}(\mathtt{ex}_1) \oplus \mathit{exop}(\mathtt{ex}_2) \text{ otherwise} \end{cases}$$

$$\mathit{exop}(\mathtt{ex.v}) = \top \quad \mathit{exop}(\mathtt{if}(\mathtt{ex},\mathtt{ex}_1,\mathtt{ex}_2) = \begin{cases} \top \text{ if } \mathit{exop}(\mathtt{ex}) = \top, \mathit{exop}(\mathtt{ex}_1) = \top \text{ or } \mathit{exop}(\mathtt{ex}_2) = \top \\ \mathtt{if}(\mathit{exop}(\mathtt{ex}), \mathit{exop}(\mathtt{ex}_1), \mathit{exop}(\mathtt{ex}_2')) \text{ otherwise} \end{cases}$$

We next state the theorem that establishes that the symbolic execution generates a superset of the operations that are actually invoked in any parallel computation. We first state several lemmas, then state the theorem. In the following lemmas, $n_0 = n_\bot[\mathtt{op} \mapsto \mathtt{this}\texttt{->}\mathtt{op}(\mathtt{parm})]$ and $l_0 = l_{\mathtt{err}}[\mathtt{this} \mapsto r, \mathtt{parm} \mapsto o]$.

Lemma 5 *If* $\langle m, \{r\texttt{->}\mathrm{op}(o)\}\rangle \Rightarrow \langle m', p'\rangle$, $n \ \triangleright n'$ *and* $r\texttt{->}\mathrm{op}(o) \sqsubseteq \mathit{ev}(m, l_0, n(\mathrm{op}))$ *then* $\forall r'\texttt{->}\mathrm{op}'(o') \in p' : r'\texttt{->}\mathrm{op}'(o') \sqsubseteq \mathit{ev}(m, l_0, n'(\mathrm{op}'))$.

Proof Sketch: Let $\langle m, r\texttt{->}\mathtt{op}(o)\rangle \rightarrow \langle m_1, s_1\rangle$ and $\langle m, \mathit{ev}(m, l_0, n(\mathtt{op}))\rangle \rightarrow \langle m_2, s_2\rangle$. By monotonicity of f and g, $s_1 \sqsubseteq s_2$. By definition of $\triangleright$, $\forall r_1\texttt{->}\mathtt{op}_1(o_1) \in \mathit{sp}(s_1)\exists r_2\texttt{->}\mathtt{op}_2(o_2) \in \mathit{sp}(s_2) : r_1\texttt{->}\mathtt{op}_1(o_1) \sqsubseteq r_2\texttt{->}\mathtt{op}_2(o_2) \sqsubseteq n'(op')$.

Lemma 6 *If* $\langle m_0, r\texttt{->}\mathrm{op}(o)\rangle \Rightarrow \cdots \Rightarrow \langle m_k, p_k\rangle \Rightarrow \langle m_{k+1}, p_{k+1}\rangle$ *and* $n_0 \ \triangleright \cdots \triangleright n_k \ \triangleright n_{k+1}$ *then* $\forall \mathrm{op} : \mathit{ev}(m_k, l_0, n_{k+1}(\mathrm{op})) = \mathit{ev}(m_{k+1}, l_0, n_{k+1}(\mathrm{op}))$.

Lemma 6 is much more general than it needs to be. Because $n_{k+1}(\mathtt{op})$ contains no subexpressions of the form $\mathtt{ex.v}$, $\mathit{ev}(m, l_0, n_{k+1}(\mathtt{op}))$ is the same for all m. We state the lemma in this general form, however, because it suggests ways to improve the precision of the analysis without compromising the basic framework. It is possible, for example, to maintain a precise representation of expressions that reference instance variables whose values are not affected by the invoked operations.

Theorem 3 $\exists n : n_0 \ \triangleright \cdots \triangleright n \ \triangleright n$.

Theorem 4 *If* $n_0 \ \triangleright \cdots \triangleright n \ \triangleright n$ *then* $\forall r'\texttt{->}\mathrm{op}'(o') \in \mathit{gen}(m, r\texttt{->}\mathrm{op}(o)) : r'\texttt{->}\mathrm{op}'(o') \sqsubseteq \mathit{ev}(m, l_0, n(\mathrm{op}'))$.

5.5 Parallelization

To determine if it is possible to parallelize the execution of a given operation op, we start with $n_0 = n_\perp[\texttt{op} \mapsto \texttt{this->op(parm)}]$ and compute the next n_k in the sequence $n_0 \rhd n_1 \cdots \rhd n_k \rhd \cdots$ until reaching a fixed point n_f. We then compare all of the different pairs of potentially invoked operations to see if they commute according to the following algorithm. In this algorithm $type(\texttt{op})$ refers to the type of the receivers of op operations.

let $s = \{n_f(\texttt{op}) : n_f(\texttt{op}) \neq \perp\texttt{->op}(\perp)\}$
parallelize = true
for all $\texttt{ex}_1\texttt{->op}(\texttt{ex}_2) \in s$ and $\texttt{ex}'_1\texttt{->op}'(\texttt{ex}'_2) \in s$
  if (($type(\texttt{op}) = type(\texttt{op}')$) and ($\texttt{op} \neq \texttt{op}'$)) then parallelize = false
  else if ($\texttt{ex}_2 = \top$) then
    if ($\exists \texttt{v} \in iv(\texttt{op}) : \texttt{parm} \in dep(opb(\texttt{op})(\texttt{v}))$) then parallelize = false
    if ($\exists \texttt{ox} \in sp(opt(\texttt{op})) : \texttt{parm} \in dep(\texttt{ox})$ and $iv(\texttt{op}) \cap dep(\texttt{ox}) \neq \emptyset$) then parallelize = false

We next address code generation issues. If it is possible to parallelize an operation we generate two methods: a parallel method and a wrapper method that calls the parallel method then waits until the computation has finished. The parallel method directly invokes the parallel versions of all of the invoked methods. At each invocation site the algorithm generates code that executes the invoked operation concurrently with all other invoked operations. The compiler augments the data structure declaration of the receiver object with a lock. The parallel method acquires the lock before the first access to the receiver, then releases the lock before the first operation invocation site. The compiler may generate multiple lock release sites; only one will be executed per invocation. Figure 2 contains an example of the generated code.

6 Conclusion

Existing parallelizing compilers all preserve the data dependences of the original serial execution. We believe that this strategy is too conservative: compilers must recognize and exploit commuting operations if they are to effectively parallelize a range of applications. This paper presents the first general algorithm that can statically recognize and exploit commuting operations. We state a general set of conditions that is sufficient to guarantee commutativity, prove that if operations meet these conditions then their executions commute, show how to to statically verify that all of the operations invoked by the execution of a given operation commute, and describe an algorithm that automatically generates parallel code for such operations.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[2] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, August 1991.

[3] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[4] A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[5] W.L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[6] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[7] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[8] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[9] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[10] S. Pinter and R. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[11] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[12] D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[13] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[14] J. Solworth and B. Reagan. Arbitrary order operations on trees. In *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Portland, OR, August 1993.

[15] G. Steele. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, San Francisco, CA, January 1990.

[16] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.