Commutativity of Reducers*

Yu-Fang Chen[1], Chih-Duo Hong[1], Nishant Sinha[2], and Bow-Yaw Wang[1]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] IBM Research, India

Abstract. In the Map-Reduce programming model for data parallel computation, a reducer computes an output from a list of input values associated with a key. The inputs however may not arrive at a reducer in a fixed order due to nondeterminism in transmitting key-value pairs over the network. This gives rise to the *reducer commutativity* problem, that is, is the reducer computation independent of the order of its inputs? In this paper, we study the reducer commutativity problem formally. We introduce a syntactic subset of integer programs termed *integer reducers* to model real-world reducers. In spite of syntactic restrictions, we show that checking commutativity of integer reducers over unbounded lists of exact integers is undecidable. It remains undecidable even with input lists of a fixed length. The problem however becomes decidable for reducers over unbounded input lists of bounded integers. We propose an efficient reduction of commutativity checking to conventional assertion checking and report experimental results using various off-the-shelf program analyzers.

1 Introduction

Map-Reduce is a widely adopted programming model for data-parallel computation such as those in a cloud computing environment. The computation consists of two key phases: *map* and *reduce*. Each phase is carried out by a number of map and reduce instances called mappers and reducers respectively. A mapper takes a key-value pair as input and produces zero or more output key-value pairs. The output pairs produced by all mappers are *shuffled* by a load-balancing algorithm and delivered to appropriate reducers. A reducer iterates through the input values associated with a particular key and produces an output key-value pair. Consider the example which counts frequencies of each word in a distributed file system. A mapper takes an input pair (*filename*, *content*) and produces an output pair $(w, 1)$ for each word w in *content*. A reducer then receives an input pair $(w, [1; 1; \cdots ; 1])$ and returns an output pair (w, n) where n is the sum of values associated with the word w, equivalently, the frequency of the word w.

Due to the deployment of mappers/reducers, load-balancing algorithm and network latency, the order of values received by a reducer is not fixed. If a reducer computes different outputs for different input orders (namely, it is *not commutative*), the Map-Reduce program may yield different results on different runs. This makes such programs hard to debug and even cause errors. The commutativity problem for a reducer

* This work was partially supported by the Ministry of Science and Technology of Taiwan (102-2221-E-001 -016 -MY3, 103-2221-E-001 -019 -MY3, and 103-2221-E-001 -020 -MY3).

© Springer-Verlag Berlin Heidelberg 2015
C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 131–146, 2015.
DOI: 10.1007/978-3-662-46681-0_9

program R is to check if the computation of R is commutative over its (possibly unbounded) list of inputs. A recent study [19] found that the majority of analyzed real-life reducers are in fact non-commutative. Somewhat surprisingly, the problem of formally checking commutativity of reducers however has attracted little attention.

At a first glance, the commutativity problem for arbitrary reducers appears to be undecidable by the Rice's theorem. Yet reducers are seldom Turing machines in practice. Most real-world reducers simply iterate through their input list and compute their outputs; they do not have complicated control or data flows. Therefore, one wonders if the commutativity problem for such reducers can be decided for practical purposes.

On the other hand, because real-world reducers have a simple structure, perhaps manual inspection is enough to decide if a reducer is commutative? Consider the two sample reducers `dis` and `rangesum` shown below (in C syntax, simplified by omitting the *key* input). Both reducers compute the average of a selected set of elements from the input array `x` of length N and are very similar structurally. However, note that `dis` is commutative while `rangesum` is not: `dis` selects elements from `x` which are greater than 1000, while `rangesum` selects elements at index more than 1000. Checking commutativity of such reducers manually can be tricky. Automated tool support is required.

```
int dis (int x[N]) {
  int i = 0, ret = 0, cnt = 0;
  for (i = 0; i < N; i++) {
   if (x[i] > 1000){
     ret = ret + x[i];
     cnt = cnt + 1;
   }
  }
  if (cnt !=0) return ret / cnt;
  else return 0;
}
```

```
int rangesum (int x[N]) {
  int i, ret = 0, cnt = 0;
  for (i = 0; i < N; i++) {
   if (i > 1000){
     ret = ret + x[i];
     cnt = cnt + 1;
   }
  }
  if (cnt !=0) return ret / cnt;
  else return 0;
}
```

In this paper, we investigate the problem of reducer commutativity checking formally. To model real-world reducers, we introduce *integer reducers*, a syntactically restricted class of loopy programs over integer variables. In addition to assignments and conditional branches, a reducer contains an iterator to loop over inputs. Two operations are allowed on the iterator: *next*, which moves the iterator to the subsequent element in the input list; and *initialize*, which moves the iterator to the beginning of input list. Integer reducers do not allocate memory and are assumed to always terminate. In spite of these restrictions, we believe that integer reducers can capture the core computation of real-world reducers faithfully. The paper makes the following contributions:

- Via a reduction from solving Diophantine equations, we first show that checking the commutativity of integer reducers over exact integers with unbounded lengths of input lists is *undecidable*. The problem remains undecidable even with a bounded number of input values.
- Most reducer programs do not use exact integers in practice. We investigate the problem of checking reducer commutativity over bounded integers but with unbounded lengths of input lists. This problem turns out to be *decidable*. Using automata- and group-theoretic constructions, we reduce the commutativity checking problem to the language equivalence problem over two-way deterministic finite automata.
- Finally, we reduce the reducer commutativity problem to program assertion checking. The reduction applies to arbitrary reducers instances with input lists of a bounded

length. It enables checking the commutativity of real-world reducers automatically using off-the-shelf program analyzers. We present an evaluation of different program analysis techniques for checking reducer commutativity.

Related Work. Previous work on commutativity [17,15,6] has focused on checking if interface operations on a shared data structure commute, often to enable better parallelization. Their approach is *event*-centric, that is, it checks for independence of operations on data with arbitrary shapes. In contrast, our approach is *data*-centric: we use group-theoretic reductions on ordered data collections for efficient checking.

A recent survey [19] points out the abundance of non-commutative reducers in industrial Map-Reduce deployments. Previous approaches to checking reducer commutativity use black-box testing [20] and symbolic execution [4]. They generate large number of tests using permutations of the input and verify that the output is same. This does not scale even for small input sizes. Checking commutativity of reducers may be seen as a specific form of regression checking [10,7] where the two versions are identical except permuting the input order. The work in [11] proposes a static analysis technique to check re-orderings in the data-flow architecture consisting of multiple map and reduce phases using read or write conflicts between different phases. It does not consider the data commutativity problem.

The paper is organized as follows. We review basic notions in Sec. 2. Sec. 3 presents a formal model for reducers and a definition of the commutativity problem. It is followed by the undecidability result (Sec. 4). We then consider reducers with only bounded integers in Sec. 5. Sec. 6 shows the commutativity problem for bounded integer reducers is decidable. Sec. 7 gives the experimental results. We conclude in Sec. 8.

2 Preliminaries

Let $\mathbb{Z}, \mathbb{Z}^+, \mathbb{N}$ denote the set of integers, positive integers, and non-negative integers respectively. Define $\underline{n} = \{1, 2, \ldots, n\}$ when $n \in \mathbb{Z}^+$. A *permutation* on $\underline{n}$ is a one-to-one and onto mapping from $\underline{n}$ to $\underline{n}$. The set of permutations on $\underline{n}$ is denoted by S_n. It can be shown that S_n is a group (called the *symmetric group on n letters*) under the functional composition. Let $l_1, l_2, \ldots, l_m \in \mathbb{Z}$. We write $[l_1; l_2; \cdots; l_m]$ to denote the integer list consisting of the elements $l_1, l_2, \ldots, l_m$. For an integer list ℓ, the notations $|\ell|$, $\mathsf{hd}(\ell)$, and $\mathsf{tl}(\ell)$ denote the length, head, and tail of ℓ respectively. The function $\mathsf{empty}(\ell)$ returns 1 if ℓ is empty; otherwise, it returns 0. For instance, $\mathsf{hd}([0; 1; 2]) = 0$, $\mathsf{tl}([0; 1; 2]) = [1; 2]$, and $\mathsf{empty}(\mathsf{tl}([0; 1; 2])) = 0$.

We define the semantics of reducer programs using transition systems. A *transition system* $\mathcal{T} = \langle S, \longrightarrow \rangle$ consists of a (possibly infinite) set S of *states* and a *transition relation* $\longrightarrow \subseteq S \times S$. For $s, t \in S$, we write $s \rightarrow t$ for $(s, t) \in \rightarrow$.

A *two-way deterministic finite automaton (2DFA)* $M = \langle Q, \Sigma, \Delta, q_0, F \rangle$ consists of a finite *state set* Q, a finite *alphabet* Σ, a *transition function* $\Delta : Q \times \Sigma \rightarrow Q \times \{L, R, -\}$, an *initial state* $q_0 \in Q$, and an *accepting* set $F \subseteq Q$. A 2DFA has a read-only *tape* and a *read head* to indicate the current symbol on the tape. If $\Delta(q, a) = (q', \gamma)$, M at the state q reading the symbol a transits to the state q'. It then moves its read head to the left, right, or same position when γ is L, R, or $-$ respectively. A *configuration of M* is of the form wqv where $w \in \Sigma^*$, $v \in \Sigma^+$, and $q \in Q$; it indicates that M is

at the state q and reading the first symbol of v. The *initial configuration of* M *on input* w is q_0w. For any $q_f \in F$, $a \in \Sigma$, and $w \in \Sigma^*$, wq_fa is an *accepting configuration*. M *accepts* a string $w \in \Sigma^*$ if M starts from the initial configuration on input w and reaches an accepting configuration. Define $L(M) = \{w : M \text{ accepts } w\}$. A 2DFA can be algorithmically translated to a classical deterministic finite automata accepting the same language [16]. It hence recognizes a regular language.

Theorem 1. *Let* $M = \langle Q, \Sigma, \Delta, q_0, F\rangle$ *be a 2DFA.* $L(M)$ *is regular.*

2.1 Facts about Symmetric Groups

We will need notations and facts from basic group theory. Let $x_1, x_2, \ldots, x_k \in \underline{n}$ be distinct. The notation $(x_1\ x_2\ \cdots\ x_k)$ denotes a permutation function on $\underline{n}$ such that $x_1 \mapsto x_2, x_2 \mapsto x_3, \ldots, x_{k-1} \mapsto x_k$, and $x_k \mapsto x_1$. Define $\tau_k = (1\ 2\ \cdots\ k)$.

Theorem 2 ([12]). *For every* $\sigma \in S_n$, σ *is equal to a composition of* τ_2 *and* τ_n.

For $\ell = [l_1; l_2; \cdots; l_m]$ and $\sigma \in S_m$, define $\sigma(\ell) = [l_{\sigma(1)}; l_{\sigma(2)}; \cdots; l_{\sigma(m)}]$. For example, $\tau_3([3; 2; 1]) = [2; 1; 3]$. The following proposition will be useful.

Proposition 1. *Let* A *be a set of lists. The following are equivalent:*

1. *for every* $\ell \in A$ *with* $|\ell| > 1$, *both* $\tau_2(\ell)$ *and* $\tau_{|\ell|}(\ell)$ *are in* A*;*
2. *for every* $\ell \in A$ *and* $\sigma \in S_{|\ell|}$, $\sigma(\ell)$ *is in* A.

In other words, to check whether all permutations of a list belong to a set, it suffices to check two specific permutations by Proposition 1.

3 Integer Reducers

Map-Reduce is a programming model for data parallel computation. Programmers can choose to implement map and reduce phases in a programming language of their choice. In order to analyze real-world reducers, we give a formal model to characterize the essence of reducers. Our model allows to describe the computation of reducers and investigate their commutativity.

A reducer receives a key k and a non-empty list of values associated with k as input; it returns a key-value pair as an output. We are interested in checking whether the output is independent of the order of input list. Since both input and output keys are not essential, they are ignored in our model. Most data parallel computation moreover deals with numerical values [19] We assume that both input and output values are integers. To access values in a input list, our model has iterators adopted from modern programming languages. A reducer performs its core computation by iterating over the input list.

Reducers are represented by control flow graphs. Let `Var` denote the set of integer variables. Define the syntax of commands `Cmd` as follows.

$$
\begin{aligned}
v \in \texttt{Var} &\triangleq \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots \\
e \in \texttt{Exp} &\triangleq e = e \mid e > e \mid !\, e \mid e \,\&\&\, e \mid \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \mid v \mid e{+}e \mid e{\times}e \mid \\
&\quad\ \texttt{cur}() \mid \texttt{end}() \\
c \in \texttt{Cmd} &\triangleq v := e \mid \texttt{init}() \mid \texttt{next}() \mid \texttt{assume}\ e \mid \texttt{return}\ e
\end{aligned}
$$

In addition to standard expressions and commands, the command assume e blocks the computation when e evaluates to false. The command init() initializes the iterator by pointing to the first input value in the list. The expression cur() returns the current input value pointed to by the iterator. The next() command updates the iterator by pointing to the next input value. The expression end() returns 1 if the iterator is at the end of the list; it returns 0 otherwise.

A *control flow graph (CFG)* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$ consists of a finite set of *nodes* N, a set of *edges* $E \subseteq N \times N$, a *command labeling function* $\text{cmd} : E \rightarrow \texttt{Cmd}$, a *start* node $n_s \in N$, and an *end* node $n_e \in N$. The start node has no incoming edges. The end node has no outgoing edges and exactly one incoming edge. The only incoming edge of the end node is the only edge labeled with a return command. Without loss of generality, we assume that the first command is always init() and all variables are initialized to 0. Moreover, edges with the same source must all be labeled assume commands; the Boolean expressions in these assume commands must be exhaustive and exclusive. In other words, we only consider deterministic reducers.

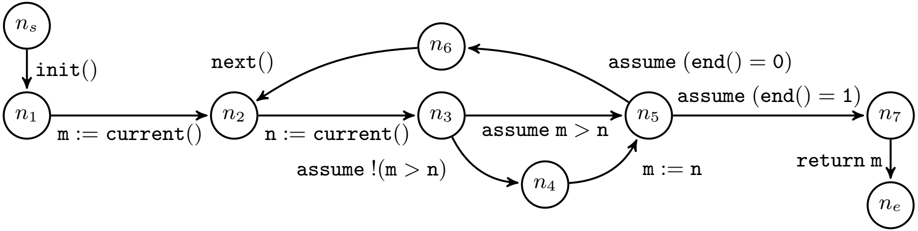


Fig. 1. A max Reducer

Figure 1 shows the CFG of a reducer. After the iterator is initialized, the reducer stores the first input value in the variable m. For each input value, it stores the value in n. If m is not greater than n, the reducer updates the variable m. It then checks if there are more input values. If so, the reducer performs a next() command and examines the next input value. Otherwise, m is returned. The reducer thus computes the maximum value of the input list.

In order to define the semantics of reducers, we assume a set of *reserved variables* $\mathbf{r} = \{\texttt{vals}, \texttt{iter}, \texttt{result}\}$. The reserved variable vals contains the list of input values; result contains the output value. The reserved variable iter is a list; it is used to model the iterator for input values. A *reserved valuation* maps each reserved variable to a value. $\mathit{Val}[\mathbf{r}]$ denotes the set of reserved valuations.

In addition to reserved variables, a reducer has a finite set of program variables $\mathbf{x}$. A *program valuation* assigns integers to program variables. $\mathit{Val}[\mathbf{x}]$ is the set of program valuations. For $\rho \in \mathit{Val}[\mathbf{r}]$, $\eta \in \mathit{Val}[\mathbf{x}]$, and $e \in \texttt{Exp}$, define $[\![e]\!]_{\rho,\eta}$ as follows.

$$
\begin{array}{rlrl}
[\![\texttt{n}]\!]_{\rho,\eta} & \triangleq n & [\![\texttt{x}]\!]_{\rho,\eta} & \triangleq \eta(\texttt{x}) \\
[\![e_0 + e_1]\!]_{\rho,\eta} & \triangleq [\![e_0]\!]_{\rho,\eta} + [\![e_1]\!]_{\rho,\eta} & [\![e_0 \times e_1]\!]_{\rho,\eta} & \triangleq [\![e_0]\!]_{\rho,\eta} \times [\![e_1]\!]_{\rho,\eta} \\
[\![!e]\!]_{\rho,\eta} & \triangleq \neg [\![e]\!]_{\rho,\eta} & [\![e_0 \;\&\&\; e_1]\!]_{\rho,\eta} & \triangleq [\![e_0]\!]_{\rho,\eta} \wedge [\![e_1]\!]_{\rho,\eta} \\
[\![e_0 = e_1]\!]_{\rho,\eta} & \triangleq [\![e_0]\!]_{\rho,\eta} = [\![e_1]\!]_{\rho,\eta} & [\![e_0 > e_1]\!]_{\rho,\eta} & \triangleq [\![e_0]\!]_{\rho,\eta} > [\![e_1]\!]_{\rho,\eta} \\
[\![\texttt{cur}()]\!]_{\rho,\eta} & \triangleq \mathsf{hd}(\rho(\texttt{iter})) & [\![\texttt{end}()]\!]_{\rho,\eta} & \triangleq \mathsf{empty}(\mathsf{tl}(\rho(\texttt{iter})))
\end{array}
$$

Let $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$ be a CFG. Define $\texttt{Cmd}_G = \{\text{cmd}(m, n) : (m, n) \in E\}$. We first define the exact integer semantics of G. $\texttt{IntReducer}_G$ is a transition system $\langle Q, \longrightarrow \rangle$ where $Q = N \times \mathit{Val}[\mathbf{r}] \times \mathit{Val}[\mathbf{x}]$ and $\longrightarrow$ is defined as follows.

$$
\begin{array}{ll}
(m, \rho, \eta) \rightarrow (n, \rho, \eta[x \mapsto \llbracket e \rrbracket_{\rho,\eta}]) & \text{if } \text{cmd}(m, n) \text{ is } x := e \\
(m, \rho, \eta) \rightarrow (n, \rho[\texttt{iter} \mapsto \rho(\texttt{vals})], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{init}() \\
(m, \rho, \eta) \rightarrow (n, \rho[\texttt{iter} \mapsto \mathsf{tl}(\rho(\texttt{iter}))], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{next}() \\
(m, \rho, \eta) \rightarrow (n, \rho[\texttt{result} \mapsto \llbracket e \rrbracket_{\rho,\eta}], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{return}\ e \\
(m, \rho, \eta) \rightarrow (n, \rho, \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{assume}\ e \text{ and } \llbracket e \rrbracket_{\rho,\eta} = \texttt{tt}
\end{array}
$$

On an `init()` command, $\texttt{IntReducer}_G$ re-initializes the reserved variable `iter` with the input values in `inputs`. The head of `iter` is the current input value of the iterator. On a `next()` command, `iter` discards the head and hence moves to the next input value. If `iter` is the empty list, no more input values remain to be read. Finally, he reserved variable `result` records the output value on the `return` command.

For $(n, \rho, \eta), (n', \rho', \eta') \in Q$, we write $(n, \rho, \eta) \xrightarrow{*} (n', \rho', \eta')$ if there are states (n_i, ρ_i, η_i) such that $(n, \rho, \eta) = (n_1, \rho_1, \eta_1)$, $(n', \rho', \eta') = (n_{k+1}, \rho_{k+1}, \eta_{k+1})$, and for every $1 \leq i \leq k$, $(n_i, \rho_i, \eta_i) \rightarrow (n_{i+1}, \rho_{i+1}, \eta_{i+1})$. Since variables are initialized to 0, let $\rho_0 \in \mathit{Val}[\mathbf{r}]$ and $\eta_0 \in \mathit{Val}[\mathbf{x}]$ be constant 0 valuations. For any non-empty list ℓ of integers, $\texttt{IntReducer}_G$ *returns* r on ℓ if $(n_s, \rho_0[\texttt{vals} \mapsto \ell], \eta_0) \xrightarrow{*} (n_e, \rho', \eta')$ and $\rho'(\texttt{result}) = r$. The elements in ℓ are the *input values*. The returned value r is an *output value*. We will also write $\texttt{IntReducer}_G(\ell)$ for the output value on ℓ.

The *commutativity problem for integer reducers* is the following: given an integer reducer $\texttt{IntReducer}_G$, decide whether $\texttt{IntReducer}_G(\ell)$ is equal to $\texttt{IntReducer}_G(\sigma(\ell))$ for every non-empty list ℓ of input values and every permutation $\sigma \in S_{|\ell|}$.

4 Undecidability of Commutativity for Integer Reducers

By Rice's theorem, the commutativity problem for Turing machines is undecidable. In practice, reducers must terminate and are often simple processes running on commodity machines. In this section, we show that the commutativity problem is undecidable even for a very restricted class of integer reducers which can iterate through each input value at most once. Such reducers are called *single-pass integer reducers*.

Undecidability is obtained by a reduction from the Diophantine problem. Let $x_1, x_2, \ldots, x_m$ be variables. A *Diophantine equation over* $x_1, x_2, \ldots, x_m$ is of the form

$$p(x_1, x_2, \ldots, x_m) = \sum_{\delta=0}^{D} \sum_{\delta_1+\delta_2+\cdots+\delta_m=\delta} c_{\delta_1,\delta_2,\ldots,\delta_m} x_1^{\delta_1} x_2^{\delta_2} \cdots x_m^{\delta_m} = 0$$

where $\delta_i \in \mathbb{N}$ for every $1 \leq i \leq m$ and D is a constant. A *system of* k *Diophantine equations* $S(x_1, x_2, \ldots, x_m)$ *over* $x_1, x_2, \ldots, x_m$ consists of k Diophantine equations $p_j(x_1, x_2, \ldots, x_m) = 0$ where $1 \leq j \leq k$. A *solution* to a system of k Diophantine equations $S(x_1, x_2, \ldots, x_m)$ is a tuple of integers $i_1, i_2, \ldots, i_m$ such that $p_j(i_1, i_2, \ldots, i_m) = 0$ for every $1 \leq j \leq k$. The *Diophantine problem* is to determine whether a given system of Diophantine equations has a solution.

Theorem 3 ([13]). *The Diophantine problem is undecidable.*

Given a system of Diophantine equations, it is straightforward to construct a single-pass integer reducer to check whether the input list of integers is a solution to the system. If the input list is indeed a solution, the reducer returns 1; otherwise, it returns 0. Hence if the given system has no solution, the reducer always returns 0 on any permutation of an input list. Note that the reducer is also commutative when the given system is trivially solved. Our construction introduces two additional variables to make the reducer not commutative on any solvable systems of Diophantine equations.

Theorem 4. *Commutativity problem for single-pass integer reducers is undecidable.*

4.1 Single-Pass Reducers over Fixed-Length Inputs

The commutativity problem for single-pass integer reducers is undecidable. It is therefore impossible to verify whether an arbitrary integer reducer produces the same output on the same input values in different orders. In the hope of identifying a decidable subproblem, we consider the commutativity problem with a fixed number of input values. The *m-commutativity problem* for integer reducers is the following: given an integer reducer $\texttt{IntReducer}_G$, determine whether $\texttt{IntReducer}_G(\ell) = \texttt{IntReducer}_G(\sigma(\ell))$ for every list of input values ℓ of length m and $\sigma \in S_m$. Because solving Diophantine equations with 9 non-negative variables is undecidable [12], the m-commutativity problem is undecidable when $m \geq 11$.

Theorem 5. *The m-commutativity problem of single-pass integer reducers is undecidable when $m \geq 11$.*

4.2 From m-Commutativity to Program Analysis

Since it is impossible to solve the m-commutativity problem completely, we propose a sound but incomplete solution to the problem. For any m input values, the naïve solution is to check whether an integer reducer returns the same output value on all permutations of the m input values. Since the number of permutations grows exponentially, the solution clearly is impractical. A more effective technique is needed.

```
l1 := *; l2 := *; ... lm := *;

x1 := l1; x2 := l2; ... xm := lm;
ret :=IntReducerG([x1; x2; ...; xm]);

x1 := l2; x2 := l1; x3 := l3; ... xm := lm;
ret2 :=IntReducerG([x1; x2; ...; xm]);
assert (ret = ret2);

x1 := l2; x2 := l3; ... xm−1 := lm; xm := l1;
retm:=IntReducerG([x1; x2; ...; xm]);
assert (ret = retm);
```

Fig. 2. Checking m-Commutativity

Our idea is to apply the group-theoretic reduction from Proposition 1. Figure 2 shows a program that realizes the idea. In the program, the expression $*$ denotes a non-deterministic value. The program starts with m non-deterministic integer values in $\texttt{l}_1, \texttt{l}_2, \ldots, \texttt{l}_m$. It stores the result of $\texttt{IntReducer}_G([\texttt{l}_1;\texttt{l}_2;\ldots;\texttt{l}_m])$ in $\texttt{ret}$. The program then computes the results

of $\texttt{IntReducer}_G(\tau_2([\texttt{l}_1; \texttt{l}_2; \ldots; \texttt{l}_m]))$ and $\texttt{IntReducer}_G(\tau_m([\texttt{l}_1; \texttt{l}_2; \ldots; \texttt{l}_m]))$. If both results are equal to `ret` for every input values, $\texttt{IntReducer}_G$ is m-commutative.

Theorem 6. *If assertions in Figure 2 hold for all computation,* $\texttt{IntReducer}_G$ *is* m*-commutative.*

Theorem 6 gives a sound but incomplete technique for the m-commutativity problem. Using off-the-shelf program analyzers, we can verify whether the assertions in Figure 2 always hold for all computation. If program analyzers establish both assertions, we conclude that $\texttt{IntReducer}_G$ is m-commutativity.

5 Bounded Integer Reducers

The commutativity problem for integer reducers is undecidable (Theorem 4). Undecidability persists even if the number of input values is fixed (Theorem 5). One may conjecture that the number of input values is irrelevant to undecidability of the commutativity problem. What induces undecidability of the problem then?

Exact integers induce undecidability in computational problems such as the Diophantine problem. However, in most programming languages, exact integers are not supported natively. Consequently, real-world reducers seldom use exact integers. It is thus more faithful to consider reducers with only bounded integers.

Fix $d \in \mathbb{Z}^+$. Define $\mathbb{Z}_d = \{0, 1, \ldots, d-1\}$. Recall that $\mathbf{r} = \{\texttt{vals}, \texttt{iter}, \texttt{result}\}$ are reserved variables. A *bounded reserved valuation* assigns the reserved variables `vals`, `iter` lists of values in $\mathbb{Z}_d$, and `result` a value in $\mathbb{Z}_d$; a *bounded program valuation* maps $\mathbf{x}$ to $\mathbb{Z}_d$. We write $BVal[\mathbf{r}]$ and $BVal[\mathbf{x}]$ for the sets of bounded reserved valuations and bounded program valuations respectively. For every $\rho \in BVal[\mathbf{r}]$, $\eta \in BVal[\mathbf{x}]$, and $e \in \texttt{Exp}$, define $[\![e]\!]_{\rho,\eta}$ as follows.

$$[\![\texttt{n}]\!]_{\rho,\eta} \triangleq n \bmod d \qquad [\![\texttt{x}]\!]_{\rho,\eta} \triangleq \eta(x)$$
$$[\![e_0 + e_1]\!]_{\rho,\eta} \triangleq [\![e_0]\!]_{\rho,\eta} + [\![e_1]\!]_{\rho,\eta} \bmod d$$
$$[\![e_0 \times e_1]\!]_{\rho,\eta} \triangleq [\![e_0]\!]_{\rho,\eta} \times [\![e_1]\!]_{\rho,\eta} \bmod d$$
$$[\![!e]\!]_{\rho,\eta} \triangleq \neg [\![e]\!]_{\rho,\eta} \qquad [\![e_0 \ \&\& \ e_1]\!]_{\rho,\eta} \triangleq [\![e_0]\!]_{\rho,\eta} \wedge [\![e_1]\!]_{\rho,\eta}$$
$$[\![e_0 = e_1]\!]_{\rho,\eta} \triangleq [\![e_0]\!]_{\rho,\eta} = [\![e_1]\!]_{\rho,\eta} \qquad [\![e_0 > e_1]\!]_{\rho,\eta} \triangleq [\![e_0]\!]_{\rho,\eta} > [\![e_1]\!]_{\rho,\eta}$$
$$[\![\texttt{cur}()]\!]_{\rho,\eta} \triangleq \mathsf{hd}(\rho(\texttt{iter})) \qquad [\![\texttt{end}()]\!]_{\rho,\eta} \triangleq \mathsf{empty}(\mathsf{tl}(\rho(\texttt{iter})))$$

Let $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$ be a CFG over program variables $\mathbf{x}$. We now define the bounded integer semantics of G. $\texttt{BoundedReducer}_G$ is a transition system $\langle Q, \hookrightarrow \rangle$ where $Q = N \times BVal[\mathbf{r}] \times BVal[\mathbf{x}]$ and the following transition relation $\hookrightarrow$:

$$\begin{array}{ll}
(m, \rho, \eta) \hookrightarrow (n, \rho, \eta[x \mapsto [\![e]\!]_{\rho,\eta}]) & \text{if } \text{cmd}(m, n) \text{ is } x := e \\
(m, \rho, \eta) \hookrightarrow (n, \rho[\texttt{iter} \mapsto \rho(\texttt{vals})], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{init}() \\
(m, \rho, \eta) \hookrightarrow (n, \rho[\texttt{iter} \mapsto \mathsf{tl}(\rho(\texttt{iter}))], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{next}() \\
(m, \rho, \eta) \hookrightarrow (n, \rho[\texttt{result} \mapsto [\![e]\!]_{\rho,\eta}], \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{return}\ e \\
(m, \rho, \eta) \hookrightarrow (n, \rho, \eta) & \text{if } \text{cmd}(m, n) \text{ is } \texttt{assume}\ e \text{ and } [\![e]\!]_{\rho,\eta} = \mathsf{tt}
\end{array}$$

Except that expressions are evaluated in modular arithmetic, $\texttt{BoundedReducer}_G$ behaves exactly the same as the integer reducer $\texttt{IntReducer}_G$. We write $(n, \rho, \eta) \overset{*}{\hookrightarrow} (n', \rho', \eta')$ if there are $(n_1, \rho_1, \eta_1) = (n, \rho, \eta)$ and $(n_{k+1}, \rho_{k+1}, \eta_{k+1}) = (n', \rho', \eta')$ such that $(n_i, \rho_i, \eta_i) \hookrightarrow (n_{i+1}, \rho_{i+1}, \eta_{i+1})$ for every $1 \leq i \leq k$. For any non-empty list ℓ of values in $\mathbb{Z}_d$, the bounded integer reducer $\texttt{BoundedReducer}_G$ *returns* r *on* ℓ if $(n_s, \rho_0[\texttt{vals} \mapsto \ell], \eta_0) \overset{*}{\hookrightarrow} (n_e, \rho', \eta')$ and $\rho'(\texttt{result}) = r$. $\texttt{BoundedReducer}_G(\ell)$ denotes the output value r returned by $\texttt{BoundedReducer}_G$ on the list ℓ of input values.

Note that the number of input values is unbounded. $\texttt{BoundedReducer}_G$ is an infinite-state transition system due to the reserved variables `vals` and `iter`. On the other hand, all program variables and the reserved variable `result` can only have finitely many different values. We will exploit this fact to attain our decidability result.

6 Deciding Commutativity of Bounded Integer Reducers

We present an automata-theoretic technique to solve the commutativity problem for bounded integer reducers. Although bounded integer reducers receive input lists of arbitrary lengths, their computation can be summarized by 2DFA exactly. Based on the 2DFA characterizing the computation of a bounded integer reducer, we construct another 2DFA to summarize the computation of the reducer on permuted input values. Using Proposition 1, we reduce the commutativity problem for bounded integer reducers to the language equivalence problem for 2DFA. Since language equivalence problem of 2DFA is decidable, checking bounded integer reducer commutativity is decidable.

More precisely, let G be a CFG, $m > 0$, and $l_1, l_2, \ldots, l_m, r \in \mathbb{Z}_d$. We construct a 2DFA A_G such that it accepts the string $\lhd l_1 l_2 \cdots l_m \rhd r$ exactly when the bounded integer reducer $\texttt{BoundedReducer}_G$ returns r on the list $[l_1; l_2; \ldots; l_m]$. For clarity, we say l_i is the i-th input value of A_G, which is in fact the i-th input value of $\texttt{BoundedReducer}_G$. We use the read-only tape as the reserved `vals` variable. Two additional reserved variables `cur` and `end` are introduced for the `cur()` and `end()` expressions. On a `return` command, A_G stores the returned value in the reserved `result` variable. If the last symbol r of the input string is equal to `result`, A_G accepts the input. Otherwise, it rejects the input. More concretely, let $\mathbf{s} = \{\texttt{cur}, \texttt{end}, \texttt{result}\}$ be reserved variables and $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$ a CFG over program variables $\mathbf{x}$. A *finite reserved valuation* maps $\mathbf{s}$ to $\mathbb{Z}_d$; a *finite program valuation* maps $\mathbf{x}$ to $\mathbb{Z}_d$. We write $FVal[\mathbf{s}]$ and $FVal[\mathbf{x}]$ for the sets of finite reserved valuations and finite program valuations respectively. Note that $FVal[\mathbf{s}]$ and $FVal[\mathbf{x}]$ are finite sets since $\mathbf{s}$, $\mathbf{x}$, $\mathbb{Z}_d$ are finite. For every $\rho \in FVal[\mathbf{s}]$, $\eta \in FVal[\mathbf{x}]$, and $e \in \texttt{Exp}$, define $\{\!|e|\!\}_{\rho,\eta}$ as follows.

$$\{\!|\texttt{n}|\!\}_{\rho,\eta} \triangleq n \bmod d \qquad \{\!|\texttt{x}|\!\}_{\rho,\eta} \triangleq \eta(x)$$

$$\{\!|e_0 \texttt{+} e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} + \{\!|e_1|\!\}_{\rho,\eta} \bmod d$$

$$\{\!|e_0 \times e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} \times \{\!|e_1|\!\}_{\rho,\eta} \bmod d$$

$$\{\!|\texttt{!}e|\!\}_{\rho,\eta} \triangleq \neg\{\!|e|\!\}_{\rho,\eta} \qquad \{\!|e_0 \texttt{ \&\& } e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} \wedge \{\!|e_1|\!\}_{\rho,\eta}$$

$$\{\!|e_0 \texttt{=} e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} = \{\!|e_1|\!\}_{\rho,\eta} \qquad \{\!|e_0 \texttt{>} e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} > \{\!|e_1|\!\}_{\rho,\eta}$$

$$\{\!|\texttt{cur()}|\!\}_{\rho,\eta} \triangleq \rho(\texttt{cur}) \qquad \{\!|\texttt{end()}|\!\}_{\rho,\eta} \triangleq \rho(\texttt{end})$$

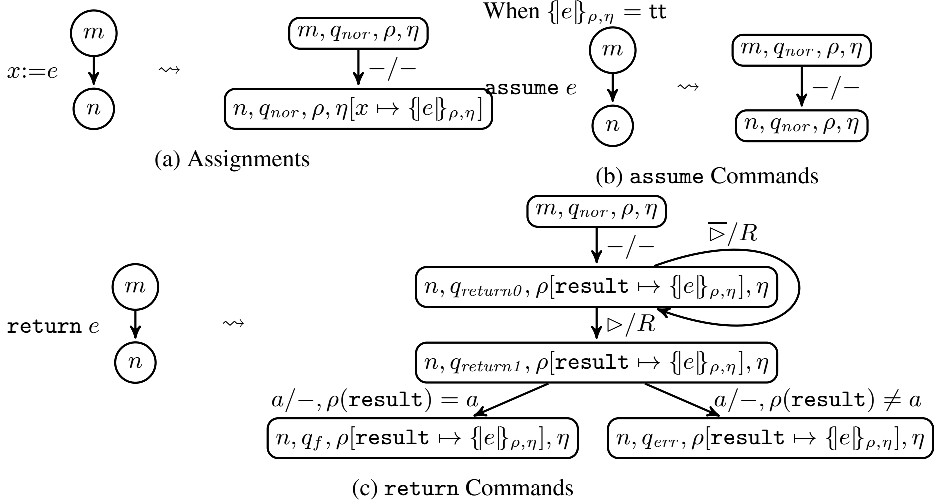


Fig. 3. Construction of A_G

A state of A_G is a quadruple (n, q, ρ, η) where n is a node in G, q is a control state, ρ is a finite reserved valuation, and η is a finite program valuation. The control state q_{nor} means the "normal" operation mode. For an assignment command in G, A_G simulates the assignment in its finite states (Figure 3a). For an `assume` command, A_G has a transition exactly when the assumed expression evaluated to `tt` (Figure 3b). For a `return` command, A_G stores the returned value in `result` and enters the control state $q_{return0}$. A_G then moves its read head to the right until it sees the $\rhd$ symbol (Figure 3c)[2]. On the $\rhd$ symbol, A_G enters the control state $q_{return1}$ and compares the last symbol a with the returned value. It enters the accepting state q_f if they are equal.

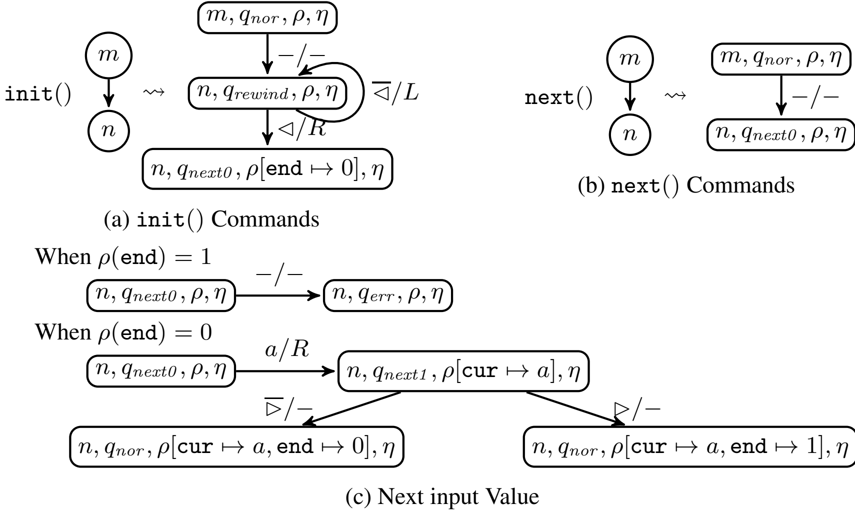
For an `init()` command, A_G initializes the iterator at the control state q_{rewind} by moving its read head to the left until the $\lhd$ symbol is read. A_G then moves its read head to the first input value, sets `end` to 0 and enters the control state q_{next0} to update the reserved variable `current` (Figure 4a). For the `next()` command, A_G enters q_{next0} to update the value of `current` (Figure 4b). At the control state q_{next0}, the symbol under its read head is the next input value. If `end` is 1, A_G enters the error control state q_{err} immediately. Otherwise, it updates the reserved variable `cur`, moves its read head to the right, and checks if there are more input values at the control state q_{next1}. If the symbol is $\rhd$, A_G sets `end` to 1 and enters the normal operation mode (Figure 4c).

Lemma 1. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$, $m > 0$, *and* $l_1, l_2, \ldots, l_m, r \in \mathbb{Z}_d$. *Then*

$$L(A_G) = \{\lhd l_1 l_2 \cdots l_m \rhd r : \texttt{BoundedReducer}_G([l_1; l_2; \cdots; l_m]) = r\}.$$

The commutativity problem for bounded integer reducers asks us to check whether a given bounded integer reducer returns the same output value on any permutation of

[2] $\overline{\alpha}$ denotes any symbol other than α.



(a) init() Commands

(b) next() Commands

(c) Next input Value

Fig. 4. Construction of A_G (continued)

input values. Applying Proposition 1, it suffices to consider two particular permutations. We have shown that the computation of a bounded integer reducer can be summarized by a 2DFA. Our proof strategy hence is to summarize the computation of the given bounded integer reducer on permuted input values by two 2DFA. We compare the computation of a bounded integer reducer on original and permuted input values by checking if the two 2DFA accept the same language.

We will generalize the construction of A_G to define another 2DFA named $A_G^{\tau_2}$ for the computation on permuted input values. Consider a non-empty list of input values $\ell = [l_1; l_2; \cdots; l_m]$ with $m > 1$. The 2DFA $A_G^{\tau_2}$ will accept the string $\lhd l_1 l_2 \cdots l_m \rhd r$ where r is $\texttt{BoundedReducer}_G(\tau_2(\ell))$ and $\texttt{BoundedReducer}_G$ is the bounded integer reducer for the CFG G. Our construction uses additional reserved variables to store the first two input values. $A_G^{\tau_2}$ also has two new control states to indicate whether the first two input values are to be read. Since the construction of $A_G^{\tau_2}$ is more complicated, we skip its description due to page limit.

Lemma 2. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$, $m > 0$, *and* $l_1, l_2, \ldots, l_m, r \in \mathbb{Z}_d$. *Then*

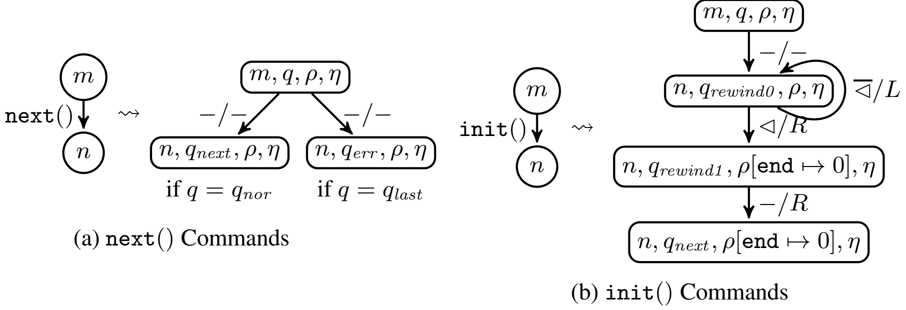
$$L(A_G^{\tau_2}) = \{\lhd l_1 l_2 \cdots l_m \rhd r : \texttt{BoundedReducer}_G(\tau_2([l_1; l_2; \cdots; l_m])) = r\}.$$

Lemma 3. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$. *The languages* $L(A_G^{\tau_2}) = L(A_G)$ *if and only if* $\texttt{BoundedReducer}_G(\ell) = \texttt{BoundedReducer}_G(\tau_2(\ell))$ *for every non-empty list* ℓ *of values in* $\mathbb{Z}_d$.

Based on the construction of A_G, we construct another 2DFA named $A_G^{\tau_*}$ which characterizes the computation of the given bounded integer reducer $\texttt{BoundedReducer}_G$ on input values in a different permutation. More precisely, for any non-empty list of input values $\ell = [l_1; l_2; \cdots; l_{|\ell|}]$, $A_G^{\tau_*}$ accepts the string $\lhd l_1 l_2 \cdots l_{|\ell|} \rhd r$ where r is

$\texttt{BoundedReducer}_G(\tau_{|\ell|}(\ell))$. For the string $\lhd l_1 l_2 \cdots l_{|\ell|} \rhd r$ on $A_G^{\tau_*}$'s tape, we want to summarize the computation of $\texttt{BoundedReducer}_G$ on $[l_2; l_3; \cdots; l_{|\ell|}; l_1]$. Observe that l_2 is the $2nd$ *input value* of $A_G^{\tau_*}$ and the $1st$ *input value* of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$.

A state of $A_G^{\tau_*}$ is a quadruple (n, q, ρ, η) where n is a node in G, q is a control state, ρ is a finite reserved valuation, and η is a finite program valuation. In addition to s, $A_G^{\tau_*}$ has the reserved variable fst to memorize the first input value of $A_G^{\tau_*}$. It also has three new control states: q_0 for initialization, q_{nor} for the normal operation mode, and q_{last} for the case where the last input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$ has been read.



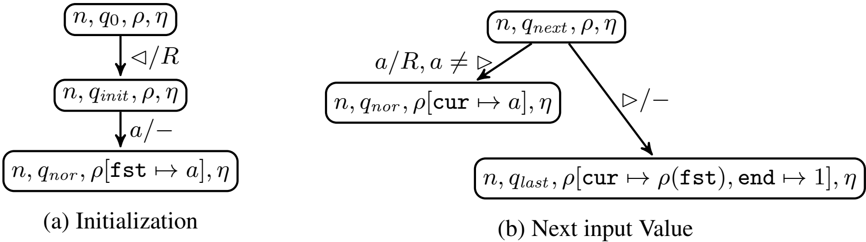
(a) next() Commands

(b) init() Commands

Fig. 5. Construction of $A_G^{\tau_*}$

$A_G^{\tau_*}$ starts by storing its first input value in the reserved variable fst and moving to the normal operation mode q_{nor}. To initialize the iterator, $A_G^{\tau_*}$ moves its read head and stores the first input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$ in the reserved variable cur. Retrieving the next input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$ is slightly different. If there are more input values, $A_G^{\tau_*}$ moves its read head to the right and updates cur accordingly. Otherwise, the first input value of $A_G^{\tau_*}$ is the last input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$. $A_G^{\tau_*}$ sets cur to the value of fst and enters q_{last}.

More concretely, $A_G^{\tau_*}$ transits to the control state q_{next} if it is in the normal operation mode q_{nor} for a next() command. It enters the error state q_{err} when the last input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$ has been read (Figure 5a). For an init() command, $A_G^{\tau_*}$ moves its read head to the second input value of $A_G^{\tau_*}$. Since the second input value of $A_G^{\tau_*}$ is the first input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$, $A_G^{\tau_*}$ sets end to 0 and enters the control state q_{next} to update the reserved variable cur (Figure 5b).



(a) Initialization

(b) Next input Value

Fig. 6. Construction of $A_G^{\tau_*}$ (continued)

Figure 6a shows the initialization step. $A_G^{\tau_*}$ simply stores its first input value in the reserved variable `fst` and transits to the normal operation model q_{nor}. The auxiliary control state q_{next} retrieves the next input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$ (Figure 6b). If there are more input values of $A_G^{\tau_*}$, $A_G^{\tau_*}$ updates `cur`, moves its read head to the right, and transits to the normal operation mode q_{nor}. If $A_G^{\tau_*}$ reaches the end of its input values, the first input value of $A_G^{\tau_*}$ is the last input value of $\texttt{BoundedReducer}_G$ on $\tau_{|\ell|}(\ell)$. $A_G^{\tau_*}$ hence updates `cur` to the value of `fst`, sets `end` to 1, and transits to q_{last}.

Lemma 4. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e\rangle$, $m > 0$, *and* $l_1, l_2, \ldots, l_m, r \in \mathbb{Z}_d$. *Then*

$$L(A_G^{\tau_*}) = \{\lhd l_1 l_2 \cdots l_m \rhd r : \texttt{BoundedReducer}_G(\tau_m([l_1; l_2; \cdots ; l_m])) = r\}.$$

Lemma 5. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e\rangle$. *The languages* $L(A_G) = L(A_G^{\tau_*})$ *if and only if* $\texttt{BoundedReducer}_G(\ell) = \texttt{BoundedReducer}_G(\tau_{|\ell|}(\ell))$ *for every non-empty list* ℓ *of values in* $\mathbb{Z}_d$.

By Proposition 1, Lemma 3 and 5, we have the following theorem:

Theorem 7. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a CFG* $G = \langle N, E, \text{cmd}, n_s, n_e\rangle$. $L(A_G) = L(A_G^{\tau_2}) = L(A_G^{\tau_*})$ *if and only if* $\texttt{BoundedReducer}_G(\ell) = \texttt{BoundedReducer}_G(\sigma(\ell))$ *for every non-empty list* ℓ *of values in* $\mathbb{Z}_d$ *and* $\sigma \in S_{|\ell|}$.

The next result follows from decidability of 2DFA language equivalence problem.

Theorem 8. *The commutativity problem for bounded integer reducers is decidable.*

7 Experiments

The reduction in Sec. 4.2 allows us to use any off-the-shelf program analyzer to check commutativity of reducers. Given a reducer, we construct a program by the reduction and verify its assertions by program analyzers. This section evaluates the performance of state-of-the-art program analyzers for checking commutativity.

We compare CBMC [3], KLEE [2], CPACHECKER [1], and our prototype tool, SYMRED. Two configurations of CPACHECKER are used: predicate abstraction automated with interpolation and abstract interpretation using octagon domain. CBMC is a bounded model checker for C programs over bounded machine integers. The tools KLEE and SYMRED implement symbolic execution techniques: KLEE symbolically executes one path at-a-time while SYMRED constructs multi-path reducer summaries using symbolic execution and precise data-flow merging [18]. The tool KLEE uses STP [8] while SYMRED uses Z3 [5] as the underlying solver.

All experiments were conducted on a Xeon 3.07GHz Linux Ubuntu workstation with 16GB memory (Table 1). The symbol (TO) denotes timeout (5 minutes). The symbol (F) denotes that an incorrect result is reported. We found that KLEE cannot handle programs with division on some benchmarks; such cases are shown with the symbol -.

Our benchmarks consist of a set of 5 reducer programs in C, parameterized over the length of the input list (from 5 to 100). All the benchmark reducers but `rangesum`

are commutative. The first three sets of benchmarks compute respectively the sum, average, and max value of the list. The benchmark `sep` computes the difference of the occurrences of even and odd numbers in the list. The example `dis` computes the average of input values greater than 100000. The example `rangesum` computes the average of input values of index greater than a half of the list length. We model input lists as bounded arrays and the iteration as a while loop with an index variable.

Table 1. Experimental Results

	CBMC	CPA-Pred.	CPA-Oct.	SYMRED	KLEE
sum5.c	43	64	3(F)	0.2	0.02
sum10.c	TO	TO	3(F)	0.4	0.02
sum20.c	TO	TO	3(F)	1	0.03
sum40.c	TO	TO	3(F)	1	0.04
sum60.c	TO	TO	4(F)	2	0.1
avg5.c	TO	TO	3(F)	0.3	-
avg10.c	TO	TO	3(F)	0.4	-
avg20.c	TO	TO	3(F)	0.8	-
avg40.c	TO	TO	3(F)	1	-
avg60.c	TO	TO	3(F)	2	-
max5.c	3	TO	3(F)	0.5	6
max10.c	215	TO	5(F)	7	102
max20.c	TO	TO	6(F)	103	TO
max40.c	TO	TO	7(F)	288	TO
max60.c	TO	TO	9(F)	TO	TO
sep5.c	0.2	21	4(F)	0.5	0.1
sep10.c	0.3	TO	8(F)	2	5
sep20.c	2	TO	202(F)	22	TO
sep40.c	26	TO	TO	21	TO
sep60.c	TO	TO	TO	22	TO
dis5.c	TO	3	4(F)	1	-
dis10.c	TO	TO	5(F)	3	-
dis20.c	TO	TO	9(F)	TO	-
dis40.c	TO	TO	24(F)	TO	-
dis60.c	TO	TO	67(F)	TO	-
rangesum5.c	0.1	5	3	0.3	-
rangesum10.c	0.1	8	3	0.5	-
rangesum20.c	2	18	3	0.9	-
rangesum40.c	4	25	4	2	-
rangesum60.c	5	TO	4	2	-

CPACHECKER with predicate abstraction generates predicates by interpolating incorrect error traces to separate reachable states and bad states. Benchmark sets such as `sum` and `avg` contain no branch conditions and has only one symbolic trace. Here, it suffices to check the satisfiability and compute interpolant of the single trace formula. Still, the verifier cannot scale to large input lists for these examples.

CPACHECKER with abstract interpretation over octagon domain finishes in seconds on all benchmarks but reports false positives on all commutative ones. We observe that a suitable abstract domain for checking commutativity should simultaneously support (a) permutations of the input list (b) numerical properties such as the sum of the input list, and (c) equivalence between numerical values. Although individual domains for numerical properties of lists [9] and program equivalence [14] exist, we are not aware of any domain combining both simultaneously.

Reducers with addition and division operations in general are difficult for CBMC. The `avg` and `div` benchmarks use divisions and the tool cannot handle cases with input lists of length more than 5. The `sep` benchmark does not use divisions. CBMC scales better on this benchmark. For `rangesum`, CBMC catches the bug in seconds.

The two symbolic execution based approaches, KLEE and SYMRED, seem to be more effective for commutativity checking. SYMRED performs better than KLEE on `sep` and `max`, both containing branches. We believe this is because SYMRED avoids KLEE-like path enumeration using precise symbolic merges with ite (if-then-else) expressions at join locations. Loop iterations produce nested ite expressions. Although simplification of such expressions reduces the actual solver time on most benchmarks, it fails to curb the blowup for the `dis` benchmark. Therefore, better heuristics are needed to check reducer commutativity for unbounded input sizes.

8 Conclusions

We present tractability results on the commutativity problem for reducers by analyzing a syntactically restricted class of integer reducers. We show that deciding commutativity of single-pass reducer over exact integers is undecidable via a reduction from solving Diophantine equation. Undecidability holds even if reducers receive only a bounded number of input values. We further show that the problem is decidable for reducers over unbounded input list over bounded integers via a reduction to language equivalence checking of 2DFA. A practical solution to commutativity checking is provided via a reduction to assertion checking using group-theoretic reduction. We evaluate the performance of multiple program analyzers on parameterized problem instances. In future, we plan to investigate better heuristics and exploit more structural properties of real-world reducers for solving the problem for unbounded inputs over exact integers.

References

1. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
2. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. ACM (2008)
3. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
4. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing MapReduce-style programs. In: FSE, pp. 504–507 (2011)
5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. In: PLDI, p. 33. ACM (2014)
7. Felsing, D., Grebing, S., Klebanov, V., Rummer, P., Ulbrich, M.: Automating regression verification. In: ASE, pp. 349–360 (2014)
8. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
9. Halbwachs, N., Peron, M.: Discovering properties about arrays in simple programs. In: PLDI (2008)
10. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 282–299. Springer, Heidelberg (2013)
11. Hueske, F., Peters, M., Sax, M.J., Rheinländer, A., Bergmann, R., Krettek, A., Tzoumas, K.: Opening the black boxes in data flow optimization. VLDB Endowment 5(11) (2012)
12. Hungerford, T.W.: Algebra. Graduate Texts in Mathematics, vol. 73. Springer (2003)
13. Jones, J.P.: Universal diophantine equation. Journal of Symbolic Logic 47(3) (1982)
14. Kovacs, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: CCS, pp. 211–222. ACM (2013)
15. Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., Pingali, K.: Exploiting the commutativity lattice. ACM SIGPLAN Notices 46(6) (2011)

16. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal Res. Dev. 3(2) (1959)
17. Rinard, M., Diniz, P.C.: Commutativity analysis: A new analysis technique for parallelizing compilers. TOPLAS 19(6), 942–991 (1997)
18. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and learn: Finding witnesses without looking all over. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 599–615. Springer, Heidelberg (2012)
19. Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., Zhou, L.: Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In: Companion Proceedings of ICSE, pp. 44–53 (2014)
20. Xu, Z., Hirzel, M., Rothermel, G.: Semantic characterization of MapReduce workloads. In: IISWC, pp. 87–97 (2013)