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 in a cloud environment, the reducer phase is responsible for computing an output key-value pair, given a sequence of input values associated with a particular key. Because of non-determinism in transmitting key-value pairs over the network, the inputs may not arrive at a reducer in a fixed order. This gives rise to the *reducer commutativity* problem, that is, is the reducer computation is independent of the order of its inputs? Commutativity of reducers is a desirable property, absence of which may lead to correctness violations and hard-to-find bugs.
In this paper, we study the reducer commutativity problem formally. To model real-world reducers, we introduce the notion of an *integer reducer*, a syntactic subset of integer programs. We show that, in spite of syntactic restrictions, deciding commutativity of integer reducers over unbounded sequences of integer values is undecidable. It remains undecidable even with input sequences of a fixed length. The problem, however, is decidable for reducers over unbounded input sequences if the integer values are bounded. We also propose an efficient reduction of commutativity checking to conventional assertion checking and report experimental results from checking commutativity using various off-the-shelf program analyzers.

1 Introduction

Map-Reduce is a widely adopted programming model for data-parallel computation, for example, in a cloud computing environment. The input data is partitioned into sets of key-value pairs for parallel computation, for instance, $(word, count)$ for the word counting problem. The computation consists of two key phases: *map* and *reduce*, each of which processes a series of key-value pairs. Several map and reduce instances may be deployed for a large Map-Reduce computation. The map phase takes a key-value pair as input and produces zero or more output key-value pairs. The output pairs produced by multiple concurrent mappers are *shuffled* by a load-balancing algorithm and delivered to appropriate reducers. The reduce phase iterates through the input values associated with a particular key and produces an output key-value pair.

Due to the variations in the number 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, that is, is *not commutative*, the Map-Reduce program may yield different results on different runs. This makes such programs hard to debug and even cause correctness violations. The commutativity problem for a reducer program R is to check if the computation of R is commutative over its, possibly unbounded, sequence of inputs. A recent study [19] found that a large proportion ($> 50\%$) of the analyzed real-life reducers are, in fact, non-commutative. However, somewhat surprisingly, the problem of checking the commutativity of reducers has received little attention formally.

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 sequence 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 skeleton, perhaps manual inspection is enough to decide if a reducer is commutative? Consider the two reducers `dis` and `rangesum` shown below (in C syntax, simplified by omitting the *key* input), which reflect the central computation in many real-world reducers. 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 for averaging. This shows that it is tricky to check the commutativity of such reducers manually; automated tool support is definitely 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 the notion of a *integer reducer*, a syntactically restricted class of loopy programs over integer variables. In addition to assignments and conditional branches, the reducer contains a single iterator variable to loop over inputs. Two operations are allowed on the iterator i: *next*, which moves i to the subsequent element in the input sequence and *initialize*, which moves i to the beginning of input sequence. 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:

– We first show, via a reduction from solving Diophantine equations, that checking the commutativity of integer reducers, over exact integers and unbounded input size, is *undecidable*. The problem remains undecidable with a bounded number of input values. This is surprising, given the restricted syntax and control flow of these reducers.
– Most reducer programs do not use exact integers in practice. We then investigate the problem of checking reducer commutativity over input sequences with bounded integer values (but unbounded length). This problem turns out to be *decidable*: using automata- and group-theoretic constructions, we reduce the commutativity checking problem to the regular language equivalence problem over two-way deterministic finite automata.
– Finally, we provide a reduction from reducer commutativity problem to program assertion checking; the reduction applies to arbitrary reducers instances with input sequences of bounded length. This enables checking the commutativity of real-world reducers automatically using off-the-shelf program analysis engines. We present an evaluation of multiple different program analysis techniques for checking reducer commutativity.

The paper is organized as follows. We review basic notions in Sec. 2. Sec. 3 presents a formal model for reducers and 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, Sec. 8 explains the related work and we conclude in Sec. 9.

2 Preliminaries

Let $\underline{n} = \{1, 2, \ldots, n\}$. 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 $q_0 w$. For any $q_f \in F$, $a \in \Sigma$, and $w \in \Sigma^*$, $wq_f a$ 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 $L(M) = \{w : M \text{ accepts } w\}$. It is known that 2DFA can be algorithmically translated to language equivalent classical deterministic finite automata (DFA) and checking language equivalence between two DFA is decidable [16].

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 $\{x_1, x_2, \ldots, x_k\} \subseteq \underline{n}$. 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 permutation $\sigma \in S_n$, 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, $\tau_3([3; 2; 1]) = [2; 1; 3]$. Next proposition is useful to checking commutativity of reducers.

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 sequence of values associated with k as input; it returns a key-value pair (k, v) as the output. We are interested in checking whether the output is independent of the order of input sequence. Since both input and output keys are not essential, they are ignored in our model. Because most data parallel computation deals with numerical values [19], we assume that both input and output values are integers. To access input sequence values, we adopt the abstract notion of iterators from modern programming languages; a reducer performs its core computation by iterating over the input sequence.

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

$v \in \texttt{Var} \triangleq \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots$	
$e \in \texttt{Exp} \triangleq e = e \mid e > e \mid !\, e \mid e \,\&\&\, e \mid$	Boolean expressions
$\ldots, -1, 0, 1, 2, \ldots \mid v \mid e{+}e \mid e{\times}e \mid$	integer expressions
$\texttt{current}() \mid$	current element
$\texttt{end}()$	end of iterator
$c \in \texttt{Cmd} \triangleq v := e \mid$	assignment
$\texttt{init\_iter}() \mid$	initialize iterator
$\texttt{next}() \mid$	next element
$\texttt{assume}\ e \mid$	assumption
$\texttt{return}\ e$	return

The command $\texttt{init\_iter}()$ initializes the iterator by pointing to the first input value in the list. The expression $\texttt{current}()$ returns the current input value pointed to by the iterator. The $\texttt{next}()$ command updates the iterator by pointing to the next input value. The expression $\texttt{end}()$ returns 1 if the iterator is at the end of the list; it returns 0 otherwise.

A *control flow graph* $G = \langle N, E, \mathrm{cmd}, n_s, n_e \rangle$ consists of a finite set of *nodes* N, a set of *edges* $E \subseteq N \times N$, a *command labeling function* $\mathrm{cmd} : E \to \texttt{Cmd}$, 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 $\texttt{init\_iter}()$ 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.

Figure 1 shows the control flow graph 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 $\texttt{next}()$ command and examines the next input value. Otherwise, `m` is returned. The reducer thus computes the maximum value from the input sequence.

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 implement 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.

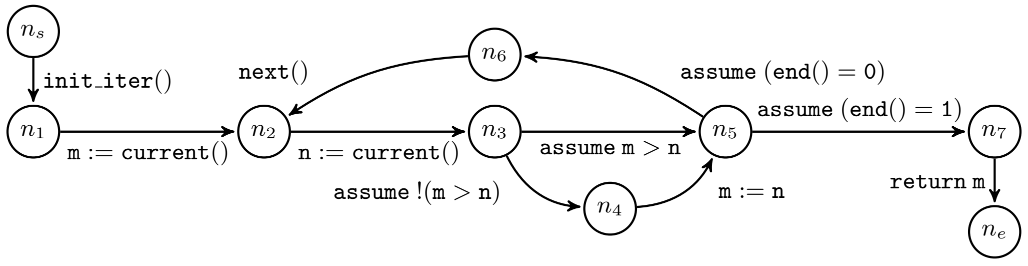


Fig. 1: A max Reducer

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 $\llbracket e \rrbracket_{\rho,\eta}$ as follows.

$$\begin{aligned}
\llbracket \mathtt{n} \rrbracket_{\rho,\eta} &\triangleq n & \llbracket \mathtt{x} \rrbracket_{\rho,\eta} &\triangleq \eta(\mathtt{x})\\
\llbracket e_0 + e_1 \rrbracket_{\rho,\eta} &\triangleq \llbracket e_0 \rrbracket_{\rho,\eta} + \llbracket e_1 \rrbracket_{\rho,\eta} & \llbracket e_0 \times e_1 \rrbracket_{\rho,\eta} &\triangleq \llbracket e_0 \rrbracket_{\rho,\eta} \times \llbracket e_1 \rrbracket_{\rho,\eta}\\
\llbracket !e \rrbracket_{\rho,\eta} &\triangleq \neg \llbracket e \rrbracket_{\rho,\eta} & \llbracket e_0 \,\&\&\, e_1 \rrbracket_{\rho,\eta} &\triangleq \llbracket e_0 \rrbracket_{\rho,\eta} \wedge \llbracket e_1 \rrbracket_{\rho,\eta}\\
\llbracket e_0 = e_1 \rrbracket_{\rho,\eta} &\triangleq \llbracket e_0 \rrbracket_{\rho,\eta} = \llbracket e_1 \rrbracket_{\rho,\eta} & \llbracket e_0 > e_1 \rrbracket_{\rho,\eta} &\triangleq \llbracket e_0 \rrbracket_{\rho,\eta} > \llbracket e_1 \rrbracket_{\rho,\eta}\\
\llbracket \mathtt{current}() \rrbracket_{\rho,\eta} &\triangleq \mathsf{hd}(\rho(\mathtt{iter})) & \llbracket \mathtt{end}() \rrbracket_{\rho,\eta} &\triangleq \mathsf{empty}(\rho(\mathtt{iter}))
\end{aligned}$$

Let $G = \langle N, E, \mathrm{cmd}, n_s, n_e \rangle$ be a control flow graph . Define $\mathtt{Cmd}_G = \{\mathrm{cmd}(m,n) : (m,n) \in E\}$. We first define the exact integer semantics of G. $\mathtt{IntReducer}_G$ is a transition system $\langle Q, \longrightarrow \rangle$ where $Q = N \times \mathit{Val}[\mathbf{r}] \times \mathit{Val}[\mathbf{x}]$ and $\longrightarrow$ is the following transition relation:

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

On an init_iter() command, $\mathtt{IntReducer}_G$ re-initializes the reserved variable iter with the input values in inputs. The reserved variable result records the output value on the return command. The reserved variable iter implements the iterator for input values. 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.

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 $1 \leq i \leq k$, $(n_i,\rho_i,\eta_i) \to (n_{i+1},\rho_{i+1},\eta_{i+1})$. 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, $\mathtt{IntReducer}_G$ *returns* r on ℓ if $(n_s, \rho_0[\mathtt{vals} \mapsto \ell], \eta_0) \xrightarrow{*} (n_e, \rho', \eta')$ and $\rho'(\mathtt{result}) = r$. The elements in ℓ are the *input values*. The returned value r is an *output value*. We will also write $\mathtt{IntReducer}_G(\ell)$ for the output value on ℓ.

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

4 Undecidability of Commutativity for Integer Reducers

By Rice's theorem, it is easy to see that the commutativity problem for Turing machines is undecidable. In practice, reducers must terminate and are often simple processes running on commod-

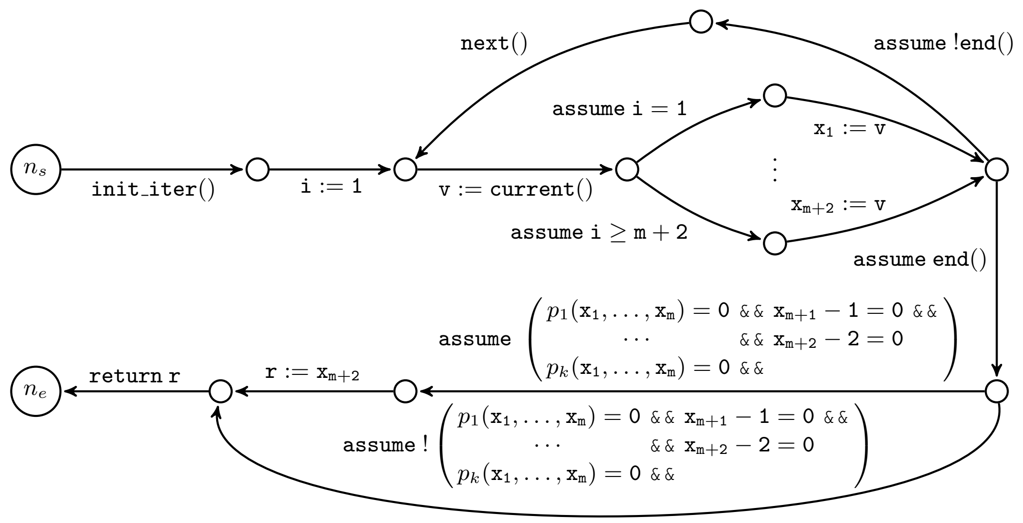

next()
assume !end()
assume i = 1
x1 := v
⋮
xm+2 := v
assume i ≥ m + 2
assume end()
n_s
init_iter()
i := 1
v := current()
assume (p1(x1, ..., xm) = 0 && xm+1 − 1 = 0 && ··· && xm+2 − 2 = 0 pk(x1, ..., xm) = 0 &&)
n_e
return r
r := xm+2
assume ! (p1(x1, ..., xm) = 0 && xm+1 − 1 = 0 && ··· && xm+2 − 2 = 0 pk(x1, ..., xm) = 0 &&)

Fig. 2: A Single-Pass Integer Reducer

ity machines. In this section, we show that the commutativity problem is undecidable even for a very restricted class of integer reducers which only allows to iterate through each input value at most once. We call such class of reducers *single-pass integer reducers*.

The undecidability result 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 $p_1(x_1, x_2, \ldots, x_m) = 0$, $p_2(x_1, x_2, \ldots, x_m) = 0$, $\ldots$, $p_k(x_1, x_2, \ldots, x_m) = 0$. 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.*

We will reduce the Diophantine problem to the commutativity problem for single-pass integer reducers. Given a system of Diophantine equations $S(x_1, x_2, \ldots, x_m) = \{p_j(x_1, x_2, \ldots, x_m) = 0 : 1 \leq j \leq k\}$, define $\hat{S}(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2}) = S(x_1, x_2, \ldots, x_m) \cup \{x_{m+1} - 1 = 0, x_{m+2} - 2 = 0\}$.

Lemma 1. *Consider any system of Diophantine equations $S(x_1, x_2, \ldots, x_m)$. For every integers $i_1, i_2, \ldots, i_m \in \mathbb{Z}$, $i_1, i_2, \ldots, i_m$ is a solution to $S(x_1, x_2, \ldots, x_m)$ if and only if $i_1, i_2, \ldots, i_m, 1, 2$ is a solution to $\hat{S}(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2})$.*

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

Proof (Sketch). Let $S(x_1, x_2, \ldots, x_m) = \{p_j(x_1, x_2, \ldots, x_m) = 0 : 1 \leq j \leq k\}$ be a system of Diophantine equations. Consider the single-pass integer reducer $\texttt{IntReducer}_G$ for the control flow graph G in Figure 2. $\texttt{IntReducer}_G$ stores $m + 2$ input values in the program variables $\mathtt{x_1}, \mathtt{x_2}, \ldots, \mathtt{x_{m+2}}$. It then checks if the Diophantine system $\hat{S}(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2})$ is solved by the input values. If $\hat{S}(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2})$ has no solution, $\texttt{IntReducer}_G$ returns 0 for all input value list (r is initialized to 0) and hence $\texttt{IntReducer}_G$ is commutative. On the other hand, if $\hat{S}(x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2})$ has a solution $i_1, i_2, \ldots, i_m, 1, 2$, then we should have $\texttt{IntReducer}_G([i_1; i_2; \cdots; i_m; 1; 2]) = 2$, but $\texttt{IntReducer}_G([i_1; i_2; \cdots; i_m; 2; 1]) = 0$. Hence $\texttt{IntReducer}_G$ is not commutative. □

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 sub-problem, 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 $m \geq 11$.*

Corollary 1. *The m-commutativity problem of integer reducers is undecidable when $m \geq 11$.*

4.2 From m-Commutativity to Program Analysis

```
l_1 := *; l_2 := *; ... l_m := *;

x_1 := l_1; x_2 := l_2; ... x_m := l_m;
ret :=IntReducer_G([x_1; x_2; ...; x_m]);

x_1 := l_2; x_2 := l_1; x_3 := l_3; ... x_m := l_m;
ret_2 :=IntReducer_G([x_1; x_2; ...; x_m]);
assert (ret = ret_2);

x_1 := l_2; x_2 := l_3; ... x_{m-1} := l_m; x_m := l_1;
ret_m:=IntReducer_G([x_1; x_2; ...; x_m]);
assert (ret = ret_m);
```

Fig. 3: 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.

Our idea is to apply the group-theoretic reduction from Proposition 1. Rather than checking all permutations of input values, it suffices to verify the output values on two particular permutations. Figure 3 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 then stores the output value $\texttt{IntReducer}_G([\texttt{l}_1; \texttt{l}_2; \ldots; \texttt{l}_m])$ in $\texttt{ret}$. The program then computes the output values $\texttt{IntReducer}_G(\tau_2([\texttt{l}_1; \texttt{l}_2; \ldots; \texttt{l}_m]))$ and $\texttt{IntReducer}_G(\tau_m([\texttt{l}_1; \texttt{l}_2; \ldots; \texttt{l}_m]))$. If both output values are equal to $\texttt{ret}$ for every m input values, we conclude that $\texttt{IntReducer}_G$ is m-commutative.

Theorem 6. *If assertions in Figure 3 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 3 always hold for all computation. If program analyzers establish both assertions, we conclude that $\texttt{IntReducer}_G$ is m-commutativity. Program analyzers however may fail to prove the assertions; they may also disprove the assertions incorrectly. The m-commutativity problem is inconclusive in both cases.

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 a positive integer $d > 0$. Recall that $\mathbf{r} = \{\texttt{vals}, \texttt{iter}, \texttt{result}\}$ are reserved variables. Define $\mathbb{Z}_d = \{0, 1, \ldots, d-1\}$. A *bounded reserved valuation* assigns the reserved variables $\texttt{vals}$, $\texttt{iter}$ lists of elements in $\mathbb{Z}_d$, and $\texttt{result}$ an element 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 set 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 $\llbracket e \rrbracket_{\rho,\eta}$ as follows.

$$\begin{array}{rclrcl}
\llbracket \texttt{n} \rrbracket_{\rho,\eta} & \triangleq & n \bmod d & \llbracket \texttt{x} \rrbracket_{\rho,\eta} & \triangleq & \eta(x) \\
\multicolumn{6}{c}{\llbracket e_0 \texttt{+} e_1 \rrbracket_{\rho,\eta} \triangleq \llbracket e_0 \rrbracket_{\rho,\eta} + \llbracket e_1 \rrbracket_{\rho,\eta} \bmod d} \\
\multicolumn{6}{c}{\llbracket e_0 \times e_1 \rrbracket_{\rho,\eta} \triangleq \llbracket e_0 \rrbracket_{\rho,\eta} \times \llbracket e_1 \rrbracket_{\rho,\eta} \bmod d} \\
\llbracket !e \rrbracket_{\rho,\eta} & \triangleq & \neg \llbracket e \rrbracket_{\rho,\eta} & \llbracket e_0 \ \texttt{\&\&}\ e_1 \rrbracket_{\rho,\eta} & \triangleq & \llbracket e_0 \rrbracket_{\rho,\eta} \wedge \llbracket e_1 \rrbracket_{\rho,\eta} \\
\llbracket e_0 \texttt{=} e_1 \rrbracket_{\rho,\eta} & \triangleq & \llbracket e_0 \rrbracket_{\rho,\eta} = \llbracket e_1 \rrbracket_{\rho,\eta} & \llbracket e_0 \texttt{>} e_1 \rrbracket_{\rho,\eta} & \triangleq & \llbracket e_0 \rrbracket_{\rho,\eta} > \llbracket e_1 \rrbracket_{\rho,\eta} \\
\llbracket \texttt{current}() \rrbracket_{\rho,\eta} & \triangleq & \mathsf{hd}(\rho(\texttt{iter})) & \llbracket \texttt{end}() \rrbracket_{\rho,\eta} & \triangleq & \mathsf{empty}(\rho(\texttt{iter}))
\end{array}$$

Let $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$ be a control flow graph over program variables $\mathbf{x}$. In this section we use 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 $\hookrightarrow$:

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

Except that expressions are evaluated in modular arithmetic, $\texttt{BoundedReducer}_G$ behaves exactly the same as the integer reducer $\texttt{IntReducer}_G$. Similarly, 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 elements 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 $\rho'(\texttt{result}) = r$. We use $\texttt{BoundedReducer}_G(\ell)$ to denote 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 because the reserved variables `vals` and `iter` have infinitely many possible values. 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 obtain our decidability result.

6 Deciding Commutativity of Bounded Integer Reducers

We present an automata-theoretic technique to establish decidability of the commutativity problem for bounded integer reducers. Although bounded integer reducers receive input sequences of arbitrary lengths, we first show that 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, it follows that checking bounded integer reducer commutativity is decidable.

More precisely, let G be a control flow graph, $m > 0$, 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 $\lhd l_1 l_2 \cdots l_m \rhd r$[1] 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 `current` and `end` are introduced for the `current`() 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 $\mathsf{s} = \{\texttt{current}, \texttt{end}, \texttt{result}\}$ be reserved variables and $G = \langle N, E, \mathrm{cmd}, n_s, n_e \rangle$ a control flow graph over program variables $\mathbf{x}$. A *finite reserved valuation* maps s to $\mathbb{Z}_d$; a *finite program valuation* maps $\mathbf{x}$ to $\mathbb{Z}_d$. We write $FVal[\mathsf{s}]$ and $FVal[\mathbf{x}]$ for the sets of finite reserved valuations and finite program valuations respectively. Note that $FVal[\mathsf{s}]$ and $FVal[\mathbf{x}]$ are finite sets since s, $\mathbf{x}$, $\mathbb{Z}_d$ are finite. For every $\rho \in FVal[\mathsf{s}]$, $\eta \in FVal[\mathbf{x}]$, and $e \in \texttt{Exp}$, define $\{\!|e|\!\}_{\rho,\eta}$ as follows.

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

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

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

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

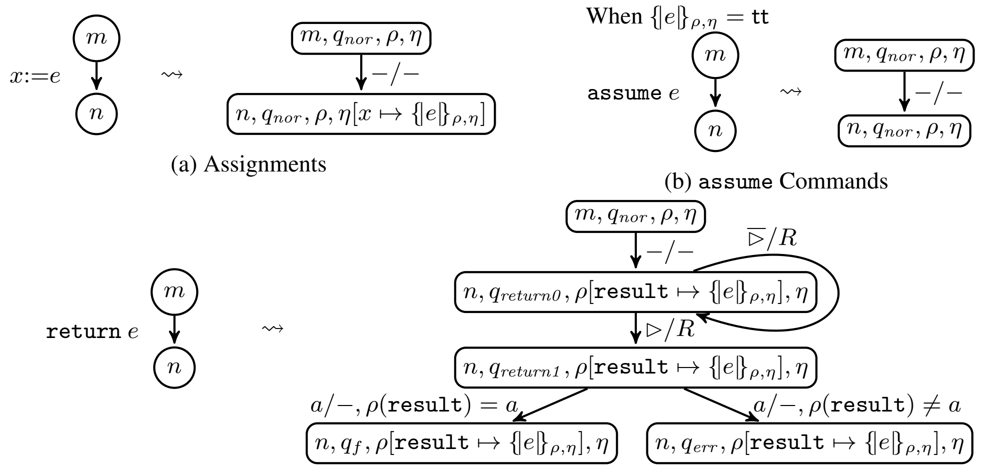
$$\{\!|e_0 = e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} = \{\!|e_1|\!\}_{\rho,\eta} \qquad \{\!|e_0 > e_1|\!\}_{\rho,\eta} \triangleq \{\!|e_0|\!\}_{\rho,\eta} > \{\!|e_1|\!\}_{\rho,\eta}$$

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

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 4a). For an `assume` command, A_G has a transition exactly when the assumed expression evaluated to `tt` (Figure 4b). For a `return` command, A_G stores the returned value in `result` and enters the control state $q_{return0}$. In $q_{return0}$, A_G moves its read head to the right until it sees the $\rhd$ symbol (Figure 4c)[2]. On the $\rhd$ symbol, A_G enters the control state $q_{return1}$

[1] $\lhd$ and $\rhd$ are also symbols of A_G

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



$x{:=}e$
m
n
$\rightsquigarrow$
m, q_{nor}, ρ, η
$-/-$
$n, q_{nor}, \rho, \eta[x \mapsto \{\!|e|\!\}_{\rho,\eta}]$
When $\{\!|e|\!\}_{\rho,\eta} = \mathtt{tt}$
assume e
m, q_{nor}, ρ, η
$-/-$
n, q_{nor}, ρ, η
return e
m, q_{nor}, ρ, η
$-/-$
$\overline{\rhd}/R$
$n, q_{return0}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta$
$\rhd/R$
$n, q_{return1}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta$
$a/-, \rho(\mathtt{result}) = a$
$a/-, \rho(\mathtt{result}) \neq a$
$n, q_f, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta$
$n, q_{err}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta$

(a) Assignments

(b) `assume` Commands

(c) `return` Commands

Fig. 4: Construction of A_G

and compares the last symbol a with the returned value. It enters the accepting control state q_f if a and `result` are equal.

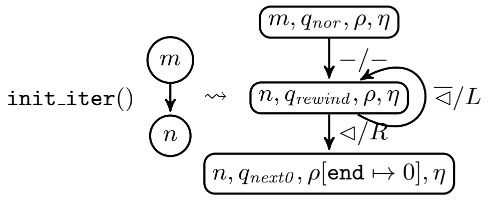
For an `init_iter`() command, A_G simulates it by moving its read head to the leftmost input value. It enters the control state q_{rewind} and stays until the $\lhd$ symbol is read. A_G then moves its read head to the first input value, sets `end` to 0 and enters the control state q_{next0} to update the reserved variable `current` (Figure 5a). For the `next`() command, A_G similarly enters q_{next0} to update the value of `current` (Figure 5b). At the control state q_{next0}, A_G assumes that 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 `current`, 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 5c). Details are in Appendix A.

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

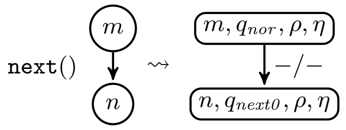
$$L(A_G) = \{\lhd l_1 l_2 \cdots l_m \rhd r : \mathtt{BoundedReducer}_G([l_1; l_2; \cdots; l_m]) = r\}.$$

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 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.

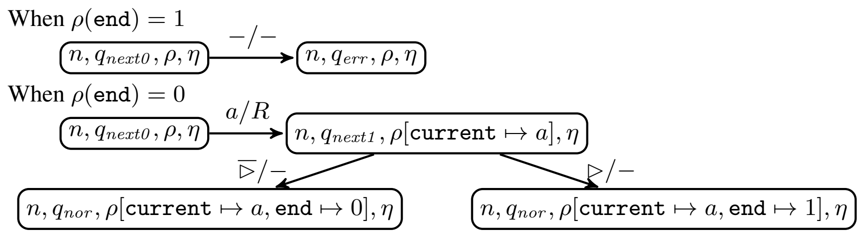
We will generalize the construction of A_G to define another 2DFA named $A_G^{\tau_2}$ for the computation on permuted input values. Consider a non-empty list of input values $\ell = [l_1; l_2; \cdots; l_m]$ with $m > 1$. The 2DFA $A_G^{\tau_2}$ will accept the string $\lhd l_1 l_2 \cdots l_m \rhd r$ where r is $\mathtt{BoundedReducer}_G(\tau_2(\ell))$ and $\mathtt{BoundedReducer}_G$ is the bounded integer reducer for the control flow graph G. Our construction uses additional reserved variables to store the first two input values. $A_G^{\tau_2}$ also has two



(a) init_iter() Commands



(b) next() Commands



When $\rho(\texttt{end}) = 1$
n, q_{next0}, ρ, η
$-/-$
n, q_{err}, ρ, η
When $\rho(\texttt{end}) = 0$
n, q_{next0}, ρ, η
a/R
$n, q_{next1}, \rho[\texttt{current} \mapsto a], \eta$
$\overline{\rhd}/-$
$\rhd/-$
$n, q_{nor}, \rho[\texttt{current} \mapsto a, \texttt{end} \mapsto 0], \eta$
$n, q_{nor}, \rho[\texttt{current} \mapsto a, \texttt{end} \mapsto 1], \eta$

(c) Next input Value

Fig. 5: Construction of A_G (continued)

new control states to indicate whether the first two input values are to be read. Since the construction of the last 2DFA is less complicated, we will give informal description for the next construction and skip $A_G^{\tau_2}$ due to page limit. The formal definition is in Appendix B.

Lemma 3. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a control flow graph* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$, $m > 0$, *and* $l_1, l_2, \ldots, l_m, r \in \mathbb{Z}_d$. *Then*

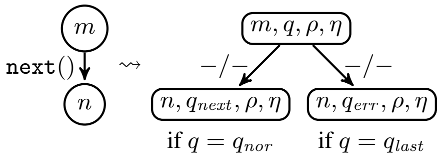
$$L(A_G^{\tau_2}) = \{\lhd l_1 l_2 \cdots l_m \rhd r : \texttt{BoundedReducer}_G(\tau_2([l_1; l_2; \cdots; l_m])) = r\}.$$

Lemma 4. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a control flow graph* $G = \langle N, E, \text{cmd}, n_s, n_e \rangle$. *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 elements in* $\mathbb{Z}_d$.

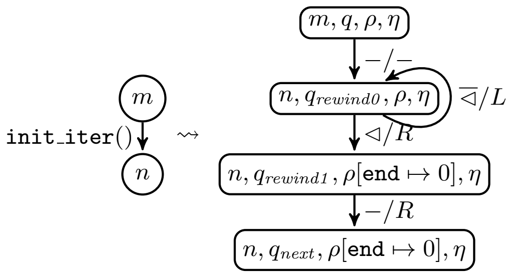
Based on the construction of A_G, we construct another 2DFA named $A_G^{\tau_*}$ which characterizes the computation of the given bounded integer reducer $\texttt{BoundedReducer}_G$ on input values in a different permutation. More precisely, for any non-empty list of input values $\ell = [l_1; l_2; \cdots; l_m]$, $A_G^{\tau_*}$ accepts the string $\lhd l_1 l_2 \cdots l_m \rhd r$ where r is $\texttt{BoundedReducer}_G(\tau_m(\ell))$. For the string $\lhd l_1 l_2 \cdots l_m \rhd r$ on $A_G^{\tau_*}$'s tape, observe that l_2 is the 2*nd input value* of $A_G^{\tau_*}$ and the 1*st input value* of $\texttt{BoundedReducer}_G$.

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 another 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 $\texttt{BoundedReducer}_G$ has been read.

At initialization, $A_G^{\tau_*}$ stores its first input value in the reserved variable fst and transits 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 $\texttt{BoundedReducer}_G$ in the reserved variable current. Retrieving the next input value of $\texttt{BoundedReducer}_G$ is slightly complicated. If there are more input values, $A_G^{\tau_*}$ moves its read head to the right and updates current accordingly. Otherwise, the first input value of $A_G^{\tau_*}$ is the last input value of $\texttt{BoundedReducer}_G$. $A_G^{\tau_*}$ sets current to the value of fst and transits to the state q_{last}.



(a) `next()` Commands

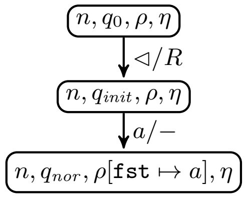


m
init_iter()
n
$\rightsquigarrow$
m, q, ρ, η
$-/-$
$n, q_{rewind0}, \rho, \eta$
$\overline{\lhd}/L$
$\lhd/R$
$n, q_{rewind1}, \rho[\texttt{end} \mapsto 0], \eta$
$-/R$
$n, q_{next}, \rho[\texttt{end} \mapsto 0], \eta$

(b) `init_iter()` Commands

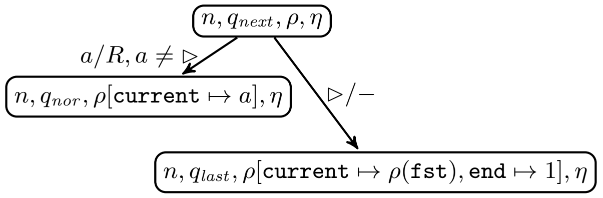
Fig. 6: Construction of $A_G^{\tau_*}$

More concretely, $A_G^{\tau_*}$ transits to the control state q_{next} if it is in the normal operation mode q_{nor} for an `next()` command. It transits to the error state q_{err} when the last input value of $\texttt{BoundedReducer}_G$ has been read (Figure 6a). For an `init_iter()` 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 $\texttt{BoundedReducer}_G$, $A_G^{\tau_*}$ sets `end` to 0 and transits to the control state q_{next} to update the reserved variable `current` (Figure 6b).



n, q_0, ρ, η
$\lhd/R$
n, q_{init}, ρ, η
$a/-$
$n, q_{nor}, \rho[\texttt{fst} \mapsto a], \eta$

(a) Initialization



n, q_{next}, ρ, η
$a/R, a \neq \rhd$
$\rhd/-$
$n, q_{nor}, \rho[\texttt{current} \mapsto a], \eta$
$n, q_{last}, \rho[\texttt{current} \mapsto \rho(\texttt{fst}), \texttt{end} \mapsto 1], \eta$

(b) Next input Value

Fig. 7: Construction of $A_G^{\tau_*}$ (continued)

Figure 7a 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 $\texttt{BoundedReducer}_G$ (Figure 7b). If there are more input values of $A_G^{\tau_*}$, $A_G^{\tau_*}$ updates `current`, 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 $A_G^{\tau_*}$ needs to be read as the last input value of $\texttt{BoundedReducer}_G$. $A_G^{\tau_*}$ hence updates `current` to the value of `fst`, sets `end` to 1, and transits to q_{last}. The complete formal definition is given in Appendix C.

Lemma 5. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a control flow graph* $G = \langle N, E, \text{cmd}, n_s, n_e\rangle$, $m > 0$, *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 6. *Let* $\texttt{BoundedReducer}_G$ *be a bounded integer reducer for a control flow graph* $G = \langle N, E, \text{cmd}, n_s, n_e\rangle$. *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 elements in* $\mathbb{Z}_d$.

By Proposition 1, Lemma 4 and 6, we have the following theorem:

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

Since 2DFA language equivalence problem is decidable, we have the following result:

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

7 Experiments

The reduction in Section 4.2 allows us to use any off-the-shelf program analyzer for checking commutativity of reducers; given a reducer, we construct the program containing assertions (Sec. 4.2) and check for any assertion violations using a program analyzer. The goal of this section is to evaluate the performance of state-of-the-art program analyzers for checking commutativity.

We pick four analyzers, CPAChecker [1], CBMC [3], Klee [2] and our home-grown tool, SymRed. We use two configurations for CPAChecker, namely, 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.

	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	-

Table 1: Verify Integer Reducers with Fixed Numbers of Input Values.

All experiments were conducted on an Xeon 3.07GHz Linux Ubuntu workstation with 16GB memory. The evaluation results are shown in Table 1. The symbol (TO) denotes that the tool cannot finish the analysis within the timeout period of 5 minutes. The symbol (F) denotes that the tool produced an incorrect result. We found that Klee cannot handle programs with division on some of our 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. The code for the benchmarks and other details of the experiments are available in the Appendix D.

Predicate abstraction based CPAChecker uses predicates to separate reachable states and bad states; new predicates are inferred via interpolation from an incorrect error trace. Benchmark sets such as "sum" and "avg" contain no branch conditions and contain 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 using abstract interpretation over octagon domain finishes in seconds on all benchmarks but produces false positives on all commutative ones. We observe that a suitable abstract domain for checking commutativity should be able to simultaneously support (a) permutation orders of the input sequence (b) numerical properties, e.g., the sum of the input list, and (c) equivalence between numerical values. Although domains handling numerical properties of lists [9] and program equivalence [14] exist independently, we are not aware of any work which handles both simultaneously.

We found that reducers with addition and division operations in general are difficult for CBMC. The "avg" and "div" benchmarks use divisions and the model checker cannot handle the case with list length more than 5. The "sep" benchmark does not use divisions: CBMC scales better on this benchmark. For the "rangesum" examples, CBMC can catch 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 of which contain 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 Related Work

We broadly divide the related literature into works on commutativity checking in general and for Map-Reduce computation and regression checking. To our knowledge, no previous work studies the tractability of the commutativity problem for reducers.

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 (different API functions) 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, which does not scale even for small input sizes. Instead we use symbolic encoding of inputs together with properties of symmetric groups to improve scalability of the checker. Our reduction of commutativity checking to program assertion checking allows using a variety of off-the-shelf program analyzers for the problem. Our approach may be seen as a specific form of regression checking [10, 7] where the two versions are identical except permuting the input order. Hueske et al. [11] propose 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; they do not consider the data commutativity problem.

9 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 sequences 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: CAV, Springer (2011) 184–190
2. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, ACM (2008) 209–224
3. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS, Springer (2004) 168–176
4. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing mapreduce-style programs. In: FSE. (2011) 504–507
5. de Moura, L.M., Bjorner, N.: Z3: An efficient SMT solver. In: TACAS, Springer (2008) 337–340
6. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. In: PLDI, ACM (2014) 33
7. Felsing, D., Grebing, S., Klebanov, V., Rummer, P., Ulbrich, M.: Automating regression verification. In: ASE. (2014) 349–360
8. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV. (2007) 519–531
9. Halbwachs, N., Peron, M.: Discovering properties about arrays in simple programs. In: PLDI. (2008)
10. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Towards modularly comparing programs using automated theorem provers. In: CADE. (2013) 282–299
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. Volume 73 of Graduate Texts in Mathematics. Springer-Verlag (2003)
13. JONES, J.P.: Universal diophantine equation. THE 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, ACM (2013) 211–222
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) (1997) 942–991
18. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and learn: Finding witnesses without looking all over. In: CAV, Springer (2012) 599–615
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. (2014) 44–53
20. Xu, Z., Hirzel, M., Rothermel, G.: Semantic characterization of mapreduce workloads. In: IISWC. (2013) 87–97

Appendix

A Formal Construction of A_G

Let $G = \langle N, E, \mathrm{cmd}, n_s, n_e\rangle$ be a control flow graph over program variables $\mathbf{x}$ and define a set of reserved variables $\mathbf{s} = \{\mathtt{current}, \mathtt{end}, \mathtt{result}\}$. Define $A_G = \langle Q, \Sigma, \Delta, (n_s, q_{nor}, \rho_0, \eta_0), N \times \{q_f\} \times FVal[\mathbf{s}] \times FVal[\mathbf{x}]\rangle$ where $Q = N \times \{q_{nor}, q_{rewind}, q_{next0}, q_{next1}, q_{return0}, q_{return1}, q_{err}, q_f\} \times FVal[\mathbf{s}] \times FVal[\mathbf{x}]$, $\Sigma = \mathbb{Z}_d \cup \{\triangleleft, \triangleright\}$, $\rho_0 \in FVal[\mathbf{s}]$, $\eta_0 \in FVal[\mathbf{x}]$ are constant 0 functions, and for every $m, n \in N$, $\rho \in FVal[\mathbf{s}]$, $\eta \in FVal[\mathbf{x}]$, and $a \in \Sigma$:

$$
\begin{aligned}
\Delta((m, q_{nor}, \rho, \eta), a) &= ((n, q_{nor}, \rho, \eta[x \mapsto \{\!|e|\!\}_{\rho,\eta}]), -) \quad \text{if } \mathrm{cmd}(m,n) \text{ is } x := e\\
\Delta((m, q_{nor}, \rho, \eta), a) &= ((n, q_{rewind}, \rho, \eta), -) \quad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{init\_iter()}\\
\Delta((m, q_{nor}, \rho, \eta), a) &= ((n, q_{next0}, \rho, \eta), -) \quad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{next()}\\
\Delta((m, q_{nor}, \rho, \eta), a) &= ((n, q_{return0}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta), -)\\
&\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{return}\ e\\
\Delta((m, q_{nor}, \rho, \eta), a) &= ((n, q_{nor}, \rho, \eta), -)\\
&\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{assume}\ e \text{ and } \{\!|e|\!\}_{\rho,\eta} = \mathtt{tt}\\
\Delta((n, q_{rewind}, \rho, \eta), a) &= \begin{cases} ((n, q_{rewind}, \rho, \eta), L) & \text{if } a \neq \triangleleft\\ ((n, q_{next0}, \rho[\mathtt{end} \mapsto 0], \eta), R) & \text{if } a = \triangleleft \end{cases}\\
\Delta((n, q_{next0}, \rho, \eta), a) &= \begin{cases} ((n, q_{next1}, \rho[\mathtt{current} \mapsto a], \eta), R) & \text{if } \rho(\mathtt{end}) = 0\\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{end}) = 1 \end{cases}\\
\Delta((n, q_{next1}, \rho, \eta), a) &= \begin{cases} ((n, q_{nor}, \rho[\mathtt{end} \mapsto 0], \eta), -) & \text{if } a \neq \triangleright\\ ((n, q_{nor}, \rho[\mathtt{end} \mapsto 1], \eta), -) & \text{if } a = \triangleright \end{cases}\\
\Delta((n, q_{return0}, \rho, \eta), a) &= \begin{cases} ((n, q_{return0}, \rho, \eta), R) & \text{if } a \neq \triangleright\\ ((n, q_{return1}, \rho, \eta), R) & \text{if } a = \triangleright \end{cases}\\
\Delta((n, q_{return1}, \rho, \eta), a) &= \begin{cases} ((n, q_f, \rho, \eta), -) & \text{if } \rho(\mathtt{return}) = a\\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{return}) \neq a \end{cases}
\end{aligned}
$$

Example. Let us do a sample run of A_G for Figure 1 on $\triangleleft 231 \triangleright 3$. Define

$$
\begin{aligned}
&\rho_0 \triangleq \{\mathtt{current} \mapsto 0, \mathtt{end} \mapsto 0, \mathtt{result} \mapsto 0\}\\
&\rho_1 \triangleq \rho_0[\mathtt{current} \mapsto 2] \qquad \rho_2 \triangleq \rho_0[\mathtt{current} \mapsto 3]\\
&\rho_3 \triangleq \rho_0[\mathtt{current} \mapsto 1, \mathtt{end} \mapsto 1] \qquad \rho_4 \triangleq \rho_3[\mathtt{result} \mapsto 3]
\end{aligned}
$$

$$
\begin{aligned}
&(n_s, q_{nor}, \rho_0, \eta_0) \triangleleft 231 \triangleright 3 \vdash (n_1, q_{rewind}, \rho_0, \eta_0) \triangleleft 231 \triangleright 3 \vdash \triangleleft(n_1, q_{next0}, \rho_0, \eta_0) 231 \triangleright 3\\
&\vdash \triangleleft 2(n_1, q_{next1}, \rho_1, \eta_0) 31 \triangleright 3 \vdash \triangleleft 2(n_1, q_{nor}, \rho_1, \eta_0) 31 \triangleright 3 \vdash \triangleleft 2(n_2, q_{nor}, \rho_1, \eta_0[\mathtt{m} \mapsto 2]) 31 \triangleright 3\\
&\vdash \triangleleft 2(n_3, q_{nor}, \rho_1, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 31 \triangleright 3 \vdash \triangleleft 2(n_4, q_{nor}, \rho_1, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 31 \triangleright 3\\
&\vdash \triangleleft 2(n_5, q_{nor}, \rho_1, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 31 \triangleright 3 \vdash \triangleleft 2(n_6, q_{nor}, \rho_1, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 31 \triangleright 3\\
&\vdash \triangleleft 2(n_2, q_{next0}, \rho_1, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 31 \triangleright 3 \vdash \triangleleft 23(n_2, q_{next1}, \rho_2, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 1 \triangleright 3\\
&\vdash \triangleleft 23(n_2, q_{nor}, \rho_2, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 2]) 1 \triangleright 3 \vdash \triangleleft 23(n_3, q_{nor}, \rho_2, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 3]) 1 \triangleright 3\\
&\vdash \triangleleft 23(n_4, q_{nor}, \rho_2, \eta_0[\mathtt{m} \mapsto 2, \mathtt{n} \mapsto 3]) 1 \triangleright 3 \vdash \triangleleft 23(n_5, q_{nor}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) 1 \triangleright 3\\
&\vdash \triangleleft 23(n_6, q_{nor}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) 1 \triangleright 3 \vdash \triangleleft 23(n_2, q_{next0}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) 1 \triangleright 3\\
&\vdash \triangleleft 231(n_2, q_{next1}, \rho_2[\mathtt{current} \mapsto 1], \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) \triangleright 3\\
&\vdash \triangleleft 231(n_2, q_{nor}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) \triangleright 3 \vdash \triangleleft 231(n_3, q_{nor}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright 3\\
&\vdash \triangleleft 231(n_5, q_{nor}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright 3 \vdash \triangleleft 231(n_7, q_{nor}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright 3\\
&\vdash \triangleleft 231(n_e, q_{return0}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright 3 \vdash \triangleleft 231 \triangleright (n_e, q_{return1}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) 3\\
&\vdash \triangleleft 231 \triangleright (n_e, q_f, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) 3
\end{aligned}
$$

B Formal Construction of $A_G^{\tau_2}$

Let $G = \langle N, E, \mathrm{cmd}, n_s, n_e\rangle$ be a control flow graph over program variables $\mathbf{x}$, and $\mathrm{s}' = \{\mathtt{current}, \mathtt{end}, \mathtt{result}, \mathtt{fst}, \mathtt{snd}\}$ reserved variables. We use the definition $A_G^{\tau_2} = \langle Q', \Sigma, \Delta', (n_s, q_0, \rho_0, \eta_0), N \times \{q_f\} \times FVal[\mathrm{s}'] \times FVal[\mathbf{x}]\rangle$ where $Q' = N \times \{q_0, q_{fst}, q_{snd}, q_{tl}, q_{init0}, q_{init1}, q_{rewind0}, q_{rewind1}, q_{rewind2}, q_{next0}, q_{next1}, q_{return0}, q_{return1}, q_{err}, q_f\} \times FVal[\mathrm{s}'] \times FVal[\mathbf{x}]$, $\Sigma =$

$\mathbb{Z}_d \cup \{\lhd, \rhd\}$, $\rho_0 \in FVal[\mathbf{s}']$ and $\eta_0 \in FVal[\mathbf{s}']$ are constant 0 functions, and for every $m, n \in N$, $\rho \in FVal[\mathbf{s}']$, $\eta \in FVal[\mathbf{x}]$ and $a \in \Sigma$,

$$
\begin{aligned}
\Delta'((n, q_0, \rho, \eta), \lhd) &= ((n, q_{init0}, \rho, \eta), R) \\
\Delta'((m, q, \rho, \eta), a) &= ((n, q, \rho, \eta[x \mapsto \{\!|e|\!\}_{\rho,\eta}]), -) \\
&\qquad\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } x := e \text{ and } q \in \{q_{fst}, q_{snd}, q_{tl}\} \\
\Delta'((m, q, \rho, \eta), a) &= ((n, q_{rewind0}, \rho, \eta), -) \\
&\qquad\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{init\_iter}() \text{ and } q \in \{q_{fst}, q_{snd}, q_{tl}\} \\
\Delta'((m, q, \rho, \eta), a) &= ((n, q_{return0}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}], \eta), -) \\
&\qquad\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{return}\ e \text{ and } q \in \{q_{fst}, q_{snd}, q_{tl}\} \\
\Delta'((m, q, \rho, \eta), a) &= ((n, q, \rho, \eta), -) \\
&\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{assume}\ e,\ \{\!|e|\!\}_{\rho,\eta} = \mathtt{tt}, \text{ and } q \in \{q_{fst}, q_{snd}, q_{tl}\} \\
\Delta'((m, q_{fst}, \rho, \eta), a) &= ((n, q_{snd0}, \rho, \eta), -) \qquad\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{next}() \\
\Delta'((n, q_{snd0}, \rho, \eta), a) &= \begin{cases} ((n, q_{snd1}, \rho[\mathtt{current} \mapsto \rho(\mathtt{fst})], \eta), R) & \text{if } \rho(\mathtt{end}) = 0 \\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{end}) = 1 \end{cases} \\
\Delta'((n, q_{snd1}, \rho, \eta), a) &= \begin{cases} ((n, q_{snd}, \rho[\mathtt{end} \mapsto 0], \eta), -) & \text{if } a \neq \rhd \\ ((n, q_{snd}, \rho[\mathtt{end} \mapsto 1], \eta), -) & \text{if } a = \rhd \end{cases} \\
\Delta'((m, q, \rho, \eta), a) &= ((n, q_{next0}, \rho, \eta), -) \\
&\qquad\qquad \text{if } \mathrm{cmd}(m,n) \text{ is } \mathtt{next}() \text{ and } q \in \{q_{snd}, q_{tl}\} \\
\Delta'((n, q_{init0}, \rho, \eta), a) &= ((n, q_{init1}, \rho[\mathtt{fst} \mapsto a], \eta), R) \\
\Delta'((n, q_{init1}, \rho, \eta), a) &= \begin{cases} ((n, q_{fst}, \rho[\mathtt{snd} \mapsto \rho(\mathtt{fst})], \eta), -) & \text{if } a = \rhd \\ ((n, q_{fst}, \rho[\mathtt{snd} \mapsto a], \eta), -) & \text{if } a \neq \rhd \end{cases} \\
\Delta'((n, q_{rewind0}, \rho, \eta), a) &= \begin{cases} ((n, q_{rewind0}, \rho, \eta), L) & \text{if } a \neq \lhd \\ ((n, q_{rewind1}, \rho, \eta), R) & \text{if } a = \lhd \end{cases} \\
\Delta'((n, q_{rewind1}, \rho, \eta), a) &= ((n, q_{rewind2}, \rho[\mathtt{current} \mapsto \rho(\mathtt{snd})], \eta), R) \\
\Delta'((n, q_{rewind2}, \rho, \eta), a) &= \begin{cases} ((n, q_{fst}, \rho[\mathtt{end} \mapsto 1], \eta), -) & \text{if } a = \rhd \\ ((n, q_{fst}, \rho[\mathtt{end} \mapsto 0], \eta), -) & \text{if } a \neq \rhd \end{cases} \\
\Delta'((n, q_{next0}, \rho, \eta), a) &= \begin{cases} ((n, q_{next1}, \rho[\mathtt{current} \mapsto a], \eta), R) & \text{if } \rho(\mathtt{end}) = 0 \\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{end}) = 1 \end{cases} \\
\Delta'((n, q_{next1}, \rho, \eta), a) &= \begin{cases} ((n, q_{tl}, \rho[\mathtt{end} \mapsto 0], \eta), -) & \text{if } a \neq \rhd \\ ((n, q_{tl}, \rho[\mathtt{end} \mapsto 1], \eta), -) & \text{if } a = \rhd \end{cases} \\
\Delta'((n, q_{return0}, \rho, \eta), a) &= \begin{cases} ((n, q_{return0}, \rho, \eta), R) & \text{if } a \neq \rhd \\ ((n, q_{return1}, \rho, \eta), R) & \text{if } a = \rhd \end{cases} \\
\Delta'((n, q_{return1}, \rho, \eta), a) &= \begin{cases} ((n, q_f, \rho, \eta), -) & \text{if } \rho(\mathtt{result}) = a \\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{result}) \neq a \end{cases}
\end{aligned}
$$

Example. Let us do a run of $A_G^{\tau_2}$ for Figure 1 on $\lhd\mathtt{231} \rhd \mathtt{3}$. Define

$$
\begin{aligned}
\rho_0 &\triangleq \{\mathtt{fst} \mapsto 0, \mathtt{snd} \mapsto 0, \mathtt{current} \mapsto 0, \mathtt{end} \mapsto 0, \mathtt{result} \mapsto 0\} \\
\rho_1 &\triangleq \rho_0[\mathtt{fst} \mapsto 2, \mathtt{snd} \mapsto 3] \qquad & \rho_2 &\triangleq \rho_1[\mathtt{current} \mapsto 3] \\
\rho_3 &\triangleq \rho_1[\mathtt{current} \mapsto 2] \qquad & \rho_4 &\triangleq \rho_1[\mathtt{current} \mapsto 1, \mathtt{end} \mapsto 1] \\
\rho_5 &\triangleq \rho_3[\mathtt{result} \mapsto 3]
\end{aligned}
$$

$$
\begin{aligned}
&(n_s, q_0, \rho_0, \eta_0) \lhd \mathtt{231} \rhd \mathtt{3} \vdash \lhd(n_s, q_{init0}, \rho_0, \eta_0)\mathtt{231} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_s, q_{init1}, \rho_0[\mathtt{fst} \mapsto 2], \eta_0)\mathtt{31} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{2}(n_s, q_{fst}, \rho_1, \eta_0)\mathtt{31} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_1, q_{rewind0}, \rho_1, \eta_0)\mathtt{31} \rhd \mathtt{3} \vdash \lhd(n_1, q_{rewind0}, \rho_1, \eta_0)\mathtt{231} \rhd \mathtt{3} \\
&\vdash (n_1, q_{rewind0}, \rho_1, \eta_0) \lhd \mathtt{231} \rhd \mathtt{3} \vdash \lhd(n_1, q_{rewind1}, \rho_1, \eta_0)\mathtt{231} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_1, q_{rewind2}, \rho_2, \eta_0)\mathtt{31} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{2}(n_1, q_{fst}, \rho_2, \eta_0)\mathtt{31} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_2, q_{fst}, \rho_2, \eta_0[\mathtt{m} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{2}(n_3, q_{fst}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_4, q_{fst}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{2}(n_5, q_{fst}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \vdash \lhd\mathtt{2}(n_6, q_{fst}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{2}(n_2, q_{snd0}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{31} \rhd \mathtt{3} \vdash \lhd\mathtt{23}(n_2, q_{snd1}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{23}(n_2, q_{snd}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \rhd \mathtt{3} \vdash \lhd\mathtt{23}(n_3, q_{snd}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{1} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{23}(n_5, q_{snd}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{1} \rhd \mathtt{3} \vdash \lhd\mathtt{23}(n_6, q_{snd}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{1} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{23}(n_2, q_{next0}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{1} \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{231}(n_2, q_{next1}, \rho_1[\mathtt{current} \mapsto 1], \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{231}(n_2, q_{tl}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \rhd \mathtt{3} \vdash \lhd\mathtt{231}(n_3, q_{tl}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{231}(n_5, q_{tl}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \rhd \mathtt{3} \vdash \lhd\mathtt{231}(n_7, q_{tl}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \rhd \mathtt{3} \\
&\vdash \lhd\mathtt{231}(n_e, q_{return0}, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \rhd \mathtt{3} \vdash \lhd\mathtt{231} \rhd (n_e, q_{return1}, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1])\mathtt{3} \\
&\vdash \lhd\mathtt{231} \rhd (n_e, q_f, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1])\mathtt{3}
\end{aligned}
$$

C Formal Construction of $A_G^{\tau_*}$

Let $G = \langle N, E, \mathrm{cmd}, n_s, n_e \rangle$ be a control flow graph over program variables $\mathbf{x}$ and define reserved variables $\mathbf{s}'' = \{\mathtt{current}, \mathtt{end}, \mathtt{result}, \mathtt{fst}\}$. Consider 2DFA $A_G^{\tau_*} = \langle Q'', \Sigma, \Delta'', (n_s, q_0, \rho_0, \eta_0), N \times \{q_f\} \times FVal[\mathbf{s}''] \times FVal[\mathbf{x}] \rangle$ where $Q'' = N \times \{q_0, q_{last}, q_{nor}, q_{init}, q_{rewind0}, q_{rewind1}, q_{next}, q_{return0}, q_{return1}, q_{err}, q_f\} \times FVal[\mathbf{s}''] \times FVal[\mathbf{x}]$, $\Sigma = \mathbb{Z}_d \cup \{\triangleleft, \triangleright\}$, $\rho_0 \in FVal[\mathbf{s}'']$ and $\eta_0 \in FVal[\mathbf{s}'']$ are constant 0 functions, and for every $m, n \in N$, $\rho \in FVal[\mathbf{s}'']$, $\eta \in FVal[\mathbf{x}]$ and $a \in \Sigma$,

$$
\begin{aligned}
\Delta''((n, q_0, \rho, \eta), \triangleleft) &= ((n, q_{init}, \rho, \eta), R) \\
\Delta''((m, q, \rho, \eta), a) &= ((n, q, \rho, \eta[x \mapsto \{\!|e|\!\}_{\rho,\eta}]), -) \\
&\qquad \text{if } \mathrm{cmd}(m, n) \text{ is } x := e \text{ and } q \in \{q_{last}, q_{nor}\} \\
\Delta''((m, q, \rho, \eta), a) &= ((n, q_{rewind0}, \rho, \eta), -) \\
&\qquad \text{if } \mathrm{cmd}(m, n) \text{ is } \mathtt{init\_iter}() \text{ and } q \in \{q_{last}, q_{nor}\} \\
\Delta''((m, q, \rho, \eta), a) &= ((n, q_{return0}, \rho[\mathtt{result} \mapsto \{\!|e|\!\}_{\rho,\eta}]), -) \\
&\qquad \text{if } \mathrm{cmd}(m, n) \text{ is } \mathtt{return}\ e \text{ and } q \in \{q_{last}, q_{nor}\} \\
\Delta''((m, q, \rho, \eta), a) &= (n, q, \rho, \eta) \\
&\qquad \text{if } \mathrm{cmd}(m, n) \text{ is } \mathtt{assume}\ e, \{\!|e|\!\}_{\rho,\eta} = \mathtt{tt}, \text{ and } q \in \{q_{last}, q_{nor}\} \\
\Delta''((m, q_{nor}, \rho, \eta), a) &= ((n, q_{next}, \rho, \eta), -) \qquad \text{if } \mathrm{cmd}(m, n) \text{ is } \mathtt{next}() \\
\Delta''((m, q_{last}, \rho, \eta), a) &= ((n, q_{err}, \rho, \eta), -) \qquad \text{if } \mathrm{cmd}(m, n) \text{ is } \mathtt{next}() \\
\Delta''((n, q_{init}, \rho, \eta), a) &= ((n, q_{nor}, \rho[\mathtt{fst} \mapsto a], \eta), -) \\
\Delta''((n, q_{rewind0}, \rho, \eta), a) &= \begin{cases} ((n, q_{rewind0}, \rho, \eta), L) & \text{if } a \neq \triangleleft \\ ((n, q_{rewind1}, \rho[\mathtt{end} \mapsto 0], \eta), R) & \text{if } a = \triangleleft \end{cases} \\
\Delta''((n, q_{rewind1}, \rho, \eta), a) &= ((n, q_{next}, \rho, \eta), R) \\
\Delta''((n, q_{next}, \rho, \eta), a) &= \begin{cases} ((n, q_{nor}, \rho[\mathtt{current} \mapsto a], \eta), R) & \text{if } a \neq \triangleright \\ ((n, q_{last}, \rho[\mathtt{current} \mapsto \rho(\mathtt{fst}), \mathtt{end} \mapsto 1], \eta, -) & \text{if } a = \triangleright \end{cases} \\
\Delta''((n, q_{return0}, \rho, \eta), a) &= \begin{cases} ((n, q_{return0}, \rho, \eta), R) & \text{if } a \neq \triangleright \\ ((n, q_{return1}, \rho, \eta), R) & \text{if } a = \triangleright \end{cases} \\
\Delta''((n, q_{return1}, \rho, \eta), a) &= \begin{cases} ((n, q_f, \rho, \eta), -) & \text{if } \rho(\mathtt{result}) = a \\ ((n, q_{err}, \rho, \eta), -) & \text{if } \rho(\mathtt{result}) \neq a \end{cases}
\end{aligned}
$$

Example. Let us do a run of $A_G^{\tau_*}$ for Figure 1 on $\triangleleft \mathtt{231} \triangleright \mathtt{3}$. Define

$$
\begin{aligned}
&\rho_0 \triangleq \{\mathtt{fst} \mapsto 0, \mathtt{current} \mapsto 0, \mathtt{end} \mapsto 0, \mathtt{result} \mapsto 0\} \\
&\rho_1 \triangleq \rho_0[\mathtt{fst} \mapsto 2] \qquad \rho_2 \triangleq \rho_1[\mathtt{current} \mapsto 3] \\
&\rho_3 \triangleq \rho_1[\mathtt{current} \mapsto 1] \qquad \rho_4 \triangleq \rho_1[\mathtt{current} \mapsto 2, \mathtt{end} \mapsto 1] \\
&\rho_5 \triangleq \rho_4[\mathtt{result} \mapsto 3]
\end{aligned}
$$

$$
\begin{aligned}
&(n_s, q_0, \rho_0, \eta_0) \triangleleft \mathtt{231} \triangleright \mathtt{3} \vdash \triangleleft(n_s, q_{init}, \rho_0, \eta_0)\mathtt{231} \triangleright \mathtt{3} \vdash \triangleleft(n_s, q_{nor}, \rho_1, \eta_0)\mathtt{231} \triangleright \mathtt{3} \\
&\vdash \triangleleft(n_1, q_{rewind0}, \rho_1, \eta_0)\mathtt{231} \triangleright \mathtt{3} \vdash (n_1, q_{rewind0}, \rho_1, \eta_0) \triangleleft \mathtt{231} \triangleright \mathtt{3} \\
&\vdash \triangleleft(n_1, q_{rewind1}, \rho_1, \eta_0)\mathtt{231} \triangleright \mathtt{3} \vdash \triangleleft\mathtt{2}(n_1, q_{next}, \rho_1, \eta_0)\mathtt{31} \triangleright \mathtt{3} \vdash \triangleleft\mathtt{23}(n_1, q_{tl}, \rho_2, \eta_0)\mathtt{1} \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{23}(n_2, q_{tl}, \rho_2, \eta_0[\mathtt{m} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \vdash \triangleleft\mathtt{23}(n_3, q_{tl}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{23}(n_4, q_{tl}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \vdash \triangleleft\mathtt{23}(n_5, q_{tl}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{23}(n_6, q_{tl}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \vdash \triangleleft\mathtt{23}(n_2, q_{next}, \rho_2, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3])\mathtt{1} \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231}(n_2, q_{tl}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 3]) \triangleright \mathtt{3} \vdash \triangleleft\mathtt{231}(n_3, q_{tl}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231}(n_5, q_{tl}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright \mathtt{3} \vdash \triangleleft\mathtt{231}(n_6, q_{tl}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231}(n_2, q_{next}, \rho_3, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright \mathtt{3} \vdash \triangleleft\mathtt{231}(n_2, q_{last}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 1]) \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231}(n_3, q_{last}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \triangleright \mathtt{3} \vdash \triangleleft\mathtt{231}(n_5, q_{last}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231}(n_7, q_{last}, \rho_4, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \triangleright \mathtt{3} \vdash \triangleleft\mathtt{231}(n_e, q_{return0}, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2]) \triangleright \mathtt{3} \\
&\vdash \triangleleft\mathtt{231} \triangleright (n_e, q_{return1}, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{3} \vdash \triangleleft\mathtt{231} \triangleright (n_e, q_f, \rho_5, \eta_0[\mathtt{m} \mapsto 3, \mathtt{n} \mapsto 2])\mathtt{3}
\end{aligned}
$$

D Programs Reduced from Checking m-Commutative

Here we give additional information to the experiments. The source code of the 5 sets of examples are given below in Figure 8. For CBMC, we use the parameter `--unwind N+1`. For CPACHECKER, we use the parameter `-preprocess -config octagonAnalysis.properties` for octagon domain analysis and `-preprocess -setprop cpa.predicate.maxArrayLength=N -config predicateAnalysis-PredAbsRefiner-ABElf.properties` for predicate abstraction via interpolation. For KLEE, we do not use any additional parameter.

```
int sum(int x[N])
{
  int i, ret;
  ret=0;
  for(i=0;i<N;i++)
  {
    ret=ret+x[i];
  }
  return ret;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=sum(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=sum(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=sum(x);

  assert(ret==ret2
       &&ret==retm);
  return 1;
}
```

(a)sumN.c

```
int avg(int x[N])
{
  int i;
  int ret;
  ret=0;
  for(i=0;i<N;i++)
  {
    ret=ret+x[i];
  }
  return ret/N;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=avg(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=avg(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=avg(x);

  assert(ret==ret2
       &&ret==retm);
  return 1;
}
```

(b)avgN.c

```
int max(int x[N])
{
  int i;
  int ret;
  ret=0;
  for (i=0;i<N;i++)
  {
    ret=ret<x[i]?
        x[i]:ret;
  }
  return ret;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=max(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=max(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=max(x);

  assert(ret==ret2
       &&ret ==retm);
  return 1;
}
```

(c)maxN.c

```
int sep(int x[N])
{
  int i;
  int ret=0;
  for(i=0;i<N;i++){
    if(x[i]%2==0)
      ret++;
    else
      ret--;
  }
    return ret;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=sep(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=sep(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=sep(x);

  assert(ret==ret2
       &&ret ==retm);
  return 1;
}
```

(d)sepN.c

```
int dis(int x[N])
{
  int i;
  int ret;
  ret=0;
  int cnt=0;
  for (i=0;i<N;i++)
  {
    if(x[i]>100000)
    {
      ret=ret+x[i];
      cnt=cnt+1;
    }
  }
  if(cnt!=0)
    return ret/cnt;
  else
    return 0;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=dis(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=dis(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=dis(x);

  assert(ret==ret2
       &&ret ==retm);
  return 1;
}
```

(e)disN.c

```
int rangesum(int x[N])
{
  int i;
  int ret;
  ret=0;
  int cnt=0;
  for (i=0;i<N;i++)
  {
    if(i>N/2)
    {
      ret=ret+x[i];
      cnt=cnt+1;
    }
  }
  if(cnt!=0)
    return ret/cnt;
  else
    return 0;
}

int main ()
{
  int x[N],temp,ret,
      ret2,retm;
  ret=rangesum(x);

  temp=x[0];
  x[0]=x[1];
  x[1]=temp;
  ret2=rangesum(x);

  temp=x[0];
  int i=0;
  for(;i<N-1;i++)
    x[i]=x[i+1];
  x[N-1]=temp;
  retm=rangesum(x);

  assert(ret==ret2
       &&ret ==retm);
  return 1;
}
```

(f)rangesumN.c

Fig. 8: Source code of the examples