Abstract—Commutativity of data structure methods is of ongoing interest, with roots in the database community. In recent years there has been renewed interest, with results showing that commutativity is a key ingredient to enabling multicore concurrency in contexts such as parallelizing compilers, transactional memory, speculative execution and, more broadly, software scalability. Despite this interest, it remains an open question as to how a data structure’s commutativity condition can be verified automatically from its implementation. Existing strategies based on ADT specifications struggle to find the right assertion granularity; and commutativity cannot be reduced to 2-safety in a straightforward way.

We describe techniques to automatically prove the correctness of method commutativity conditions from data structure implementations. The key enabling insight is to characterize the precision necessary for commutativity reasoning using, what we call, mn-differencing relations. We then describe a reduction to reachability that decomposes the problem using mn-differencing relations and observational equivalence relations. Finally, we describe a proof-of-concept implementation and encouraging preliminary experiments, verifying commutativity of simple data structures such as a memory cell, counter, two-place Set, array-based stack, queue and a rudimentary hash table.

Source code is available (perhaps with help from the PC chair) at http://www.erickoskinen.com/cityprover/.

I. INTRODUCTION

For an object $o$, with state $\sigma$, let $o.m(x)/\bar{r}$ denote a method signature, including a vector of corresponding return values $\bar{r}$. Commutativity of two methods, denoted $o.m(x)/\bar{r} \bowtie o.n(y)/\bar{s}$, are circumstances where $m$ and $n$, when applied in either order, lead to the same final state and agree on the intermediate return values $\bar{r}$ and $\bar{s}$. A commutativity condition is a logical formula $\varphi^m_{\bar{r}}(\sigma, \bar{x}, \bar{r}, \bar{y}, \bar{s})$ indicating whether the two operations will always commute from $\sigma$.

Commutativity conditions are typically much smaller than full specifications, yet they are powerful: it has been shown that they are an enabling ingredient in correct, efficient concurrent execution in the context of parallelizing compilers [1], transactional memory [2], [3], [4], optimistic parallelism [5], speculative execution, features [6], layer concurrent programs [7], and software scalability [8]. If two code fragments commute then, when combined with linearizability (for which proof techniques exist, e.g.,[9], [10]) they can be executed concurrently. It is thus important that commutativity be correct and, in recent years, growing effort has been made toward reasoning about commutativity conditions automatically. At present, these works are either unsound [11], [12] or else they rely on data structure specifications as intermediaries [13], [14], which has pitfalls (see Sec. II).

Intuitively, commutativity is a multi-trace property: relating the behaviors in one circumstance with those in another. It is therefore tempting to pose the problem as a $k$-safety problem and attempt to leverage existing techniques for $k$-safety [15], [16], [17], [18], [19]. As we detail in Section II, however, the reduction is not immediate: the post-condition for commutativity necessitates a weaker notion than concrete equivalence and approaches that attempt to use specifications [13], [14] struggle to determine what granularity is appropriate for commutativity. Weak specifications lead to unsound conclusions, while strong specifications are unnecessarily burdensome to derive.

Contributions. We describe the first methods for verifying a given commutativity condition of a data structure, directly from its source code. (1) We begin with a reduction $\text{REDUCE}^n_m$ to automaton reachability that is designed to strengthen the pre-condition by only considering reachable data-structure states and weaken the notion of data-structure equivalence in the post-condition to observational equivalence. (Sec. IV) Although $\text{REDUCE}^n_m$ is sound, reachability solvers struggle to verify the resulting encoding: their abstraction strategies lack the ability to decompose the problem in a manner suitable to commutativity. (2) To resolve this issue, we return to the question of finding the appropriate abstraction granularity for commutativity. We introduce the concept of an mn-differencing abstraction $(\alpha, R_\alpha)$ which gives a requirement for how precise an abstraction $\alpha$ must be so that one can reason in that abstract domain and relate abstract post-states with $R_\alpha$, and yet entail return value agreement in the concrete domain. Intuitively, $R_\alpha$ captures the differences between the behavior of pairs of operations when applied in either order, while abstracting away state mutations that would be the same, regardless of the order in which they are applied. $R_\alpha$-related post-states may not yet be equivalent. We show the pieces fit together by using an observational equivalence relation $I_\beta$. Proving $I_\beta$ is an observational equivalence relation can be done using a separate abstraction $\beta$ that is more appropriate for that concern. Theorem V.1 shows that a proof using this decomposition entails that $\varphi^m_{\bar{r}}$ is a valid commutativity condition. (3) We introduce a second reduction $\text{DAREDUCE}^n_m$, which exploits mn-differencing and observational equivalence relations. $\text{DAREDUCE}^n_m$ emits two reachability tasks: automata $A_\alpha(\varphi^m_{\bar{r}}, I)$ and $A_\beta(I)$. Notably, $A_\beta(I)$ is independent of $m, n$.
and \( v^n_m \), so it can be proved safe once and then reused for every subsequent \( v^n_m \) query. DAREDUCE\(_m^n\) allows reachability analyses to synthesize separate abstractions \( \alpha \) and \( \beta \) for \( A_A(\varphi^n_m, I) \) and \( A_B(I) \), respectively.

(4) We implement REDUCE\(_m^n\) and DAREDUCE\(_m^n\) in a new tool called CITYPROVER\(^1\). (Sec. VII) It takes as input simple data structures in C (with integers, arrays, and some pointers) and a candidate formula \( \varphi^n_m \). CITYPROVER employs Ultimate’s [20] or CPAchecker’s [21] reachability analyses to prove safety of the automata tasks (or generate counterexamples).

(5) We report encouraging preliminary results that CITYPROVER can verify commutativity properties of some simple numeric data structures such as a memory cell, counter, two-place Set, array stack, array queue and rudimentary hash table. (Sec. VIII) We also show that DAREDUCE\(_m^n\) scales better than REDUCE\(_m^n\). Commutativity conditions can be fairly compact so, with CITYPROVER, a user can guess commutativity conditions and rely on CITYPROVER to prove them or report counterexamples.

Limitations. We focused on numeric programs but \( mn \)-differencing abstractions and our reductions generalize to heap programs, left for future work. Our reductions highlight some limitations of existing reachability solvers, namely, the need for improved disjunctive reasoning about permutations, a subject we leave to future work. We plan to release our benchmarks to SVCOMP so that can be used in the future.

II. Motivating Examples

Consider the SimpleSet data structure at the left of Fig. 1. This data structure is a simplification of a Set, capable of storing up to two natural numbers using private integers \( a \) and \( b \). Value -1 is reserved to indicate that nothing is stored in the variable. Method \( \text{add}(x) \) checks to see if there is space available and that \( x \) is not already in the Set, and then stores \( x \) in an open slot (either \( a \) or \( b \)). \( \text{ret} \) means return. Methods \( \text{isin}(y) \), \( \text{getsize} \) and \( \text{clear} \) are straightforward.

Methods \( \text{isin}(x) \) and \( \text{isin}(y) \) always commute because neither modifies the ADT, so we say \( \varphi_{\text{isin}}(x) \equiv \text{true} \). Commutativity of \( \text{add}(x) \) and \( \text{isin}(y) \) is more involved: 

\[
\varphi_{\text{add}}(x) \equiv x \neq y \lor (x = y \land a = x) \lor (x = y \land b = x)
\]

This condition specifies three situations (disjuncts) in which the two operations commute. In the first case, the methods are operating on different values. Method \( \text{isin}(y) \) is a read-only operation and since \( y \neq x \), it is not affected by an attempt to insert \( x \). Moreover, regardless of the order of these methods, \( \text{add}(x) \) will either succeed or not (depending on whether space is available) and this operation will not be affected by \( \text{isin}(y) \). In the other disjuncts, the element being added is already in the Set, so method invocations will observe the same return values regardless of the order and no changes (that could be observed by later methods) will be made by either of these methods. Note that there can be multiple concrete ways of representing the same semantic data structure state: \( a = 5 \land b = 3 \) is the same as \( a = 3 \land b = 5 \). Other commutativity conditions include:

\[
\varphi_{\text{clear}}(y) \equiv (a \neq y \land b \neq y), \quad \varphi_{\text{getsize}}(y) \equiv \text{true}, \quad \varphi_{\text{add}}(y) \equiv \text{false}, \quad \varphi_{\text{clear}}(y) \equiv \text{sz} = 0 \quad \text{and} \quad \varphi_{\text{add}}(y) \equiv a = x \lor b = x \lor (a \neq x \land a = -1 \land b \neq x \land b = -1).
\]

As a second running example, consider an array based implementation of Stack, given at the right of Fig. 1. ArrayStack maintains array \( A \) for data, a top index to indicate end of the stack, and has operations \( \text{push} \) and \( \text{pop} \). The commutativity condition \( \varphi_{\text{push}(x)} \equiv \text{top} > -1 \land A[\text{top}] = x \land \text{top} < \text{MAX} \) captures that they commute provided that there is at least one element in the stack, the top value is the same as the value being pushed and that there is enough space.

The above examples illustrate that commutativity conditions, even for small data-structures, can quickly become tricky to reason about. Nonetheless, correctness of these conditions is important because many parallelization strategies [2], [3], [5], [6], [7], [8] crucially depend on them being correct and, if they aren’t, then parallelization becomes unsafe. Despite some attempts [12], [13], [11], [14], to our knowledge, there are no sound techniques for verifying commutativity conditions from source code. We will now discuss what’s lacking in the state of the art, answering (i) why these existing works are insufficient and (ii) why the problem cannot easily be reduced to 2-safety.

What’s hard about this problem? At first, commutativity seems like it could be easily reduced to a 2-safety problem. To prove that \( \varphi^n_m \) is a commutativity condition for \( m(\bar{x}) \bowtie n(\bar{y}) \), one could attempt to pose the problem as a 2-safety verification, perhaps using the following Hoare quadruple:

\[
\{ \varphi^n_m \land \sigma_1 = \sigma_2 \} \quad r_m^1 := m(\bar{a}); \quad r_2^1 := n(\bar{b}); \\
\{ r_m^1 = n(\bar{b}); \quad r_2^1 = m(\bar{a}) \} \quad r_m^2 = r_m^1 \land r_n^1 = r_n^2 \land \sigma_1 = \sigma_2'
\]

This would be convenient because it would allow us to use existing 2-safety tools such as Descartes [18] or Weaver [19]. If we try the ArrayStack \( \varphi_{\text{push}(x)} \equiv A[\text{top}] = x \land \text{top} > 1 \land \text{top} < \text{MAX} \) example, running an existing tool (e.g. using a product program [17] and Ultimate [20]) yields a counterexample, with the starting state: \( A = [z, y, x, a] \land \text{top} = 2 \). The counterexample shows that in this case the post states

\[[1] \text{Released soon. Available at https://file.io/V0GSL5Qe.}\]
are different. Depending on the order methods are applied, one reaches either \( A = [z, y, x, \alpha] \) or \( A[z, y, x, \alpha] = 2 \) or else \( A[z, y, x, \alpha] = 2 \). Our knowledge of the semantics of a stack tell us that these are the same state (because the garbage in the 3rd array slot does not matter), but automated tools do not know these states are equivalent: concrete equality is too strict. Similarly, for SimpleSet \( \varphi^{\text{add}(y)}(x) = x \neq y \) we would obtain a counterexample complaining that \( (a = x \land b = y) \) is different from \( (a = y \land b = x) \).

It appears we need a better notion of equivalence for the post-states. We might then be tempted to exploit specifications, as in Kim and Rinard [13] and Bansal et al. [14]. Then we ask whether \( \text{Post}_m(\text{Post}_{\alpha}(\sigma_1)) = \text{Post}_{\alpha}(\text{Post}_m(\sigma_1)) \). One limitation with this strategy is that specifications are not always available, especially for ad hoc data structures and inferring such specifications is difficult. However, there is a bigger issue: it is unclear what precision is appropriate for commutativity. Consider a coarse specification such as \{true\}push\{x\}\{true\}. Using this specification in our Hoare quad may lead to a post-relation \{true\}, which seems to indicate that all post-states are related and we would be inclined to incorrectly conclude that any \( \varphi^m \) is a valid commutativity condition. When specifications are too coarse like this one, Bansal et al. [14] would incorrectly synthesize commutativity condition \( \varphi^\text{push}\{x\} = \text{true} \). The problem is that abstraction does not capture effects of \text{push}\{x\} that are relevant to commutativity.

Alternatively, we might try a more fine-grained specification using, e.g., a sequential \( S \) to represent the stack and carefully relating every post-state to the pre-state, with disjunction to account for the various cases, etc. Such fine-grained specifications are particularly hard to come by for programmers’ home-grown data structures, especially at this granularity that is critical for reasoning about return value. We can use, e.g., a predicate abstraction with predicates \( a = y \) and \( b = y \) (along with their negations). This also ignores all other possible values for \( a \) and \( b \): for showing return value agreement, the only relevant aspect of the state is whether or not \( y \) is in the set. Similarly, for ArrayStack \text{push}\{x\}/\text{pop}(), we only need to consider the top value and we can abstract away deeper parts of the stack. While, on the other hand, for \text{pop}() \approx \text{pop}(), the second-from-top matters.

In Sec. V we present \text{mn}-differing abstraction, which formalizes this intuition. We give a requirement for an abstraction \( \alpha \) and a relation \( R_\alpha \) in that domain, that it be precise enough so that reasoning about return value agreement in the abstract domain faithfully covers reasoning about agreement in the concrete domain. For SimpleSet, we can define \( \alpha \) based on the above predicates, and then use the relation:

\[
R_\alpha(\sigma_1, \sigma_2) \equiv [(a = x) \lor (b = x)] \leftrightarrow [(a = x) \lor (b = x)],
\]

i.e., the relation that tracks if \( \sigma_1 \) and \( \sigma_2 \) agree on those predicates. \( R_\alpha \) is a relation on abstract states whose purpose is to “summarize” the possible pairs of post-states that will have agreed on return values.

States that are related by \( R_\alpha \) may not necessarily be observationally equivalent and, thus far, we don’t have a way of summarizing observational equivalence. We next show that the pieces fit together by working with observational equivalence relations. For reasoning about this equivalence, we use a separate abstraction \( \beta \) and a relation \( I_\beta \) in that abstract domain and describe the (standard) conditions under which \( I_\beta \) implies observational equivalence. For the ArrayStack and SimpleSet examples, we can use:

\[
I_{\text{AS}}(\sigma_1, \sigma_2) \equiv \text{top}_1 = \text{top}_2 \land (\forall i \in [0, \text{top}_1], a_1[i] = a_2[i]) \land (a_1 = a_2 \land b_1 = b_2) \lor (a_1 = b_2 \land b_1 = a_2) \land (sz_1 = sz_2).
\]

\( I_{\text{AS}} \) says the two states agree on the (ordered) values in the Stack. (\text{top} means the value of \text{top} in \( \sigma_1 \).) \( I_{\text{SS}} \) specifies that two states are equivalent provided that they are storing the same values—perhaps in different ways—and they agree on the size. These relations are simpler than full ADT specifications. Putting it all together, Theorem V.1 shows that if there is an \( R_\alpha \) and \( I_\beta \) such that \( \varphi^m \) “implies” \( R_\alpha \) and \( R_\alpha \) “implies” \( I_\beta \), then \( \varphi^m \) is a valid commutativity condition.

An outcome of this decomposition is that reasoning about \( I_\beta \) (which pertains to all methods of the ADT) can be separated from reasoning about \( R_\alpha \) (which pertains to a given triple \( m, n, \varphi^m \)). Consequently, the first part can be done once, and then the second part can be done for each new commutativity validity query. In Sec. VI we describe a more modular reduction \text{DAREduce}^m \( \alpha \), which employs \text{mn}-differencing and observational equivalence relations. \text{DAREduce}^m \emits a pair of automata \( A_\alpha(\varphi^m, I) \) and \( A_\emptyset(I) \), such that the safety of the former entails that \( \varphi^m \Rightarrow R_\alpha \) and that \( R_\alpha \Rightarrow I_\beta \) while the safety of the latter entails that \( I_\beta \) captures observational equivalence.

Challenges & Contributions. To begin, in Sec. IV we introduce a “one shot” reduction \text{REDUCE}^m \( \alpha \) from verifying commutativity conditions of an ADT to a (single) automaton reachability problem. \text{REDUCE}^m \( \alpha \) accounts for a few key factors. First, we observe that the reduction can be done pairwise, focusing the problem on the method pair \( m, n \) of concern. Second, \text{REDUCE}^m \( \alpha \) ensures that we only concern ourselves with commutativity from reachable states of the object. Third, in the post-relation, we exploit the automaton-based treatment to weaken the notion of equivalence to observational equivalence. We prove \text{REDUCE}^m \( \alpha \) to be sound but demonstrate that it does not lead to scalable tools. Reachability solvers struggle to effectively decompose the problem.

We thus return to the question, What is the right abstraction granularity for commutativity? which foiled prior works. We first observe that the necessary precision depends on methods under consideration. For example, with SimpleSet and commutativity of \text{isin}(y)/\text{clear}, it is sufficient to use an abstraction that ignores \( sz \). We only need to reason about whether \( y \) is stored in \( a \) or \( b \). We can use, e.g., a predicate abstraction with predicates \( a = y \) and \( b = y \) (along with their negations). This also ignores all other possible values for \( a \) and \( b \): for showing return value agreement, the only relevant aspect of the state is whether or not \( y \) is in the set. Similarly, for ArrayStack \text{push}\{x\}/\text{pop}(), we only need to consider the top value and we can abstract away deeper parts of the stack. While, on the other hand, for \text{pop}() \approx \text{pop}(), the second-from-top matters.
Finally, in Sec. VII we describe a preliminary implementation CityProver and in Sec. VIII we show it can verify commutativity properties of some simple data structures. We also show that DAREDUCE\textsuperscript{n}_m is more tractable than REDUCE\textsuperscript{n}_m.

III. PRELIMINARIES

Language of ADT implementations. We work with a simple model of a (sequential) object-oriented language. Objects can have member fields \( o.a \) and, for the purposes of this paper, we assume them to be integers, structs, or integer arrays. We use \( \bar{a} \) to denote a vector of argument values, \( \bar{u} \) to denote a vector of return values and \( m(\bar{a})/\bar{u} \) to denote a corresponding invocation of a method which we call an action. Methods’ source code is parsed from C into control-flow automata (CFA) [22], discussed in the next Section. Edges are labeled with straight-line code denoted \( s \). For simplicity, we assume one object method cannot call another, and all object methods terminate. Non-terminating object methods are typically not useful and their termination can be confirmed using existing termination tools (e.g. [23], [24], [25], [26], [20]).

We fix a single object \( o \), denote that object’s concrete state space \( \Sigma \). We denote \( \sigma^{m(\bar{a})/\bar{u}} \rightarrow \sigma’ \) for the big-step semantics in which the arguments are provided, and the entire method is reduced. We omit the small-step semantics \([s]\). For the big-step semantics, we assume that such a successor state \( \sigma’ \) is always defined (total) and is unique (determinism). Programs can be transformed so these conditions hold, via wrapping [14] and prophecy variables [27], resp.

Definition III.1 (Observational equivalence for commutativity (e.g. [4])). We define relation \( \simeq \subseteq \Sigma \times \Sigma \) as the following gfp:

\[
\forall m(\bar{a}). \sigma_1^{m(\bar{a})/\bar{u}} \rightarrow \sigma_1’ \sigma_2^{m(\bar{a})/\bar{u}} \rightarrow \sigma_2’ \bar{r} = \bar{s} \; \sigma_1’ \simeq \sigma_2’ \sigma_1 \simeq \sigma_2
\]

The above definition expresses that two states \( \sigma_1 \) and \( \sigma_2 \) of an object are observationally equivalent \( \simeq \) provided that, when any given action \( m(\bar{a}) \) is applied to both \( \sigma_1 \) and \( \sigma_2 \), then the respective return values agree. Moreover, the resulting post-states maintain the \( \simeq \) relation. An observational equivalence relation \( I \) is a relation such that \( I \Rightarrow \simeq \).

We next use observational equivalence to define commutativity ([28], [14]) first at the layer of an action, which are particular values, and second at the layer of a method.

Definition III.2 (Commutativity). For methods \( m \) and \( n \), and values \( \bar{a}, \bar{b}, \bar{u}, \bar{v} \), \textbf{actions} \( o.m(\bar{a})/\bar{u} \) and \( o.n(\bar{b})/\bar{v} \) commute, denoted \( o.m(\bar{a})/\bar{u} \simeq o.n(\bar{b})/\bar{v} \), if for all \( \sigma \), if \( \sigma^{m(\bar{a})/\bar{u}} \rightarrow \sigma_m \) and \( \sigma^{n(\bar{b})/\bar{v}} \rightarrow \sigma_n \) then \( \sigma_{mn} \simeq \sigma_{nm} \). (Action commutativity requires return value agreement.)

Methods \( o.m \) and \( o.n \) commute denoted \( o.m \bowtie o.n \) provided that \( \forall \bar{u} \bar{v} \exists \bar{u} \bar{v} \; o.m(\bar{a})/\bar{u} \bowtie o.n(\bar{b})/\bar{v} \).

Quantification \( \forall \bar{a} \bar{b} \bar{u} \bar{v} \) above means vectors of all possible argument and return values. Our work extends to a more fine-grained notion of left-movers and right-movers [29].

We denote a commutativity condition as \( \varphi^m_n \) and assume a decidable interpretation of formula: \( [\varphi^m_n] \colon (\sigma, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \rightarrow \Bbb{B} \). The first argument is the initial state. Commutativity post- and mid-conditions can also be written [13] but for simplicity we focus on pre-conditions.

Definition III.3 (Commutativity Condition). We say that a formula \( \varphi^m_n \) is a commutativity condition for \( m \) and \( n \) provided that \( \forall \sigma \; \bar{a} \bar{b} \bar{u} \bar{v} \; [\varphi^m_n] \; \sigma \bar{a} \bar{b} \bar{u} \bar{v} \Rightarrow m(\bar{a})/\bar{u} \bowtie n(\bar{b})/\bar{v} \).

IV. ONE-SHOT REDUCTION TO REACHABILITY

We now describe an algorithm for reducing the task of verifying commutativity condition \( \varphi^m_n \) to reachability, which incorporates only reachable object states in the precondition, and employs observational equivalence for reasoning about data structure equality. The algorithm is a transformation REDUCE\textsuperscript{n}_m from an input object CFA to an output automaton \( A(\varphi^m_n) \) with an error state \( q_{err} \). We prove if \( q_{err} \) is unreachable in \( A(\varphi^m_n) \), then \( \varphi^m_n \) is a valid commutativity condition.

Object Implementations. Our formalism needs a representation of object implementations, and the output encoding. We build on the well-established notion of control-flow automata:

Definition IV.1 ([22]). A \textbf{(deterministic)} control flow automaton \( A = \langle Q, q_0, X, s, \longrightarrow \rangle \) is a finite set \( Q \) of control locations, initial location \( q_0 \), set \( X \) of typed variables, loop/branch-free statement language \( s \) and finite set of labeled edges \( \longrightarrow \subseteq Q \times s \times Q \).

We summarize other standard definitions (more details in [30]). We define a valuation of variables \( \theta : X \rightarrow \text{Val} \) and \( \theta’ \in [s] \theta \). A run \( r = q_0, \theta_0, q_1, \theta_1, q_2, \ldots \) we say \( A \) can reach automaton state \( q \) provided there exists a run to \( q \). We use \( q_{err} \) as a special error state. We next conservatively extend CFAs to represent data structure implementations:

Definition IV.2 (Object implementation CFAs). An object impl. CFA for object \( o \) with methods \( M = \{ m_1, \ldots, m_k \} \), is:

\[
A_o = \langle Q_o, [q_0^{\text{init}}, q_0^{\text{clone}}, \ldots, q_0^{m_i}], X_o, s, \longrightarrow \rangle
\]

\[
X_o = X^\text{init} \cup \{ \text{this}_i \} \cup \bigcup_{f \in M \cup \{ \text{init,close} \}} (X_f^l \cup X_f^r \cup X_f^e)
\]

(Detailed explanation in [30].)

Above, we will call each \( q_0^{m_i} \) node the \textbf{entry node} for the implementation of method \( m_i \) and we additionally require that, for every method, there is a special exit node \( q_{exit}^{m_i} \). We require that the edges that lead to \( q_{exit}^{m_i} \) contain return(\( \nu \)) statements. Subsets of variables \( X_o \) are reserved for object fields, this, method-local variables, parameters and return variables. For incorporating data structure implementations, we use inlining sugar: \( q -\text{inln}(m_i, o, \bar{x}, \bar{r}) \longrightarrow q' \)

\[
q' = \{ q_{exit}^{m_i} \rightarrow q_{exit}^{m_i} \}
\]

This definition emulates calls to a method \( m_i \), starting from CFA node \( q \). Values \( \bar{x} \) are provided as arguments, and arcs are created to the entry node \( q_0^{m_i} \) for method \( m_i \). Return values are saved into \( \bar{r} \) and there is an arc from the exit node \( q_{exit}^{m_i} \) to \( q' \).

Edges are required to be deterministic. This is not without loss of generality: nondeterminism can be supported through
\[
A(\varphi_{\text{isn}}(y)) \equiv \begin{cases}
1 & \text{SimpleSet } s_1 = \text{new SimpleSet}(); \\
2 & \text{while}(\neg \text{true}) \{ \text{int } t = \ast; \text{assume } (t > 0); \text{switch}(\ast) \{ \\
3 & \text{case } 1: s_1.\text{add}(t); \text{case } 2: s_1.\text{isin}(t); \\
4 & \text{case } 3: s_1.\text{size}(); \text{case } 4: s_1.\text{clear}(); \} \\
5 & \} \text{int } x = \ast; \text{int } y = \ast; \\
6 & \text{assume } (\varphi_{\text{add}}(s_1.\text{x})); \\
7 & \text{SimpleSet } s_2 = \text{clone}().
\end{cases}
\]

Fig. 2. An example of REDUCE\textsubscript{n}, when applied to the source of add(x) and isin(y). Formally, the result is an automaton but here it is depicted as a program. When a candidate \(\varphi_{\text{add}}(s_1.\text{x})\) is supplied to this encoding, a proof of safety entails that \(\varphi_{\text{add}}(s_1.\text{x})\) is a commutativity condition.

proposition variables (see [31]). The semantics \((q, \theta) \xrightarrow{s} (q', \theta')\) of \(\mathcal{A}\) induce a labeled transition system, with state space \(\Sigma_\mathcal{A} = Q_\mathcal{A} \times \Theta\). Naturally, commutativity of an object CFA is defined in terms of this induced transition system.

**Example:** SimpleSet. We first illustrate REDUCE\textsubscript{n} by demonstrating the result of applying it to the SimpleSet example from Section II. Fig. 2 shows the output of REDUCE\textsubscript{n}(\(\mathcal{A}_{\text{SS}}, \text{add}(x), \text{isin}(y)\)), which is an automaton denoted \(A(\varphi_{\text{add}}(x))\) represented in pseudocode. Note ERR is \(q_{err}\). \(A(\varphi_{\text{add}}(x))\) should never be executed. Rather, when a program analysis tool for reachability is applied, the tool is tricked into finding abstractions to prove commutativity. There are three main parts to \(A(\varphi_{\text{add}}(x))\):

(A) Establishing the pre-condition. For any reachable abstract state \(\sigma\) of the SimpleSet object, there will be a run of \(A(\varphi_{\text{add}}(x))\) such that the SimpleSet on Line 7 will be in state \(\sigma\). A program analysis will consider all runs that eventually exit the first loop (we don’t care about those that never exit), and the corresponding reachable state \(s_1\). From \(s_1\), \(A(\varphi_{\text{add}}(x))\) assumes that provided commutativity condition \(\varphi_m\) on Line 6 and runs will clone \(s_1\).

(B) Product program. We next employ a standard product-program construction [17], using a trivial alignment. This portion causes a program analysis to consider the effects of the methods applied in each order, and whether or not the return values will match on Line 6.

(C) Post-condition with observational equivalence. Lines 11-15 consider any sequence of method calls \(m'(\tilde{a}')\), \(m''(\tilde{a}'')\), . . . that could be applied to both \(s_1\) and \(s_2\). If observational equivalence does not hold, then there will be a run of \(A(\varphi_{\text{add}}(x))\) that applies that sequence to \(s_1\) and \(s_2\), eventually finding a discrepancy in return values and going to \(q_{err}\).

**Transformation REDUCE\textsubscript{n}**. The above example provides intuition and, for lack of space, the details of the following definition are in Appendix A.

**Definition IV.3 (REDUCE\textsubscript{n}).** For an input CFA \(\mathcal{A}_0 = \langle Q_0, \{q_0^0, q_0^1, \ldots, q_m^m\}, X_0, s, \rightarrow\rangle\), the output of REDUCE\textsubscript{n} \((\mathcal{A}_0, m(x), n(y))\) is automaton \(A(\varphi_{\text{add}}(x))\) given in Appendix A.

We now show \(A(\varphi_{\text{add}}(x))\)’s safety entails commutativity.

**Theorem IV.1.** For object implementation \(\mathcal{A}_0\) and resulting encoding \(A(\varphi_{\text{add}}(x))\), if every run of \(A(\varphi_{\text{add}}(x))\) avoids \(q_{err}\), then \(\varphi_{\text{add}}(x)\) is a commutativity condition for \(m(x)\) and \(n(y)\).

**Proof Sketch.** By case analysis on the runs of \(A(\varphi_{\text{add}}(x))\), correlating the variables in the valuation \(\theta\) with the object state. □

While REDUCE\textsubscript{n} is sound, we show in Sec. VIII that tools don’t scale well at proving the safety of REDUCE\textsubscript{n}’s output. In the next Sec. V we describe an abstraction targeted at proving commutativity to better enable automated reasoning.

V. \textit{mm}-DIFFERENCING ABSTRACTION

As seen in Sec. VIII, REDUCE\textsubscript{n} generates an output verification task for which existing tools do not perform well. We now describe how to decompose the reduction into pieces. The challenge is thus: what kind of abstraction is coarse enough to be tractable, yet fine enough to reason about commutativity? We now address this with \textit{mm}-differencing abstractions and later incorporate them into DAREDUCE\textsubscript{n}. We first define \textit{posts} to be the set of all pairs of post-states originating from \(\sigma\) after the methods are applied in the two alternate orders:

\[
\text{posts}(\sigma, m, \tilde{a}, n, \tilde{b}) \equiv \{ (\sigma_1, \sigma_2) \mid \sigma' \xrightarrow{m(\bar{a})/r_1^m} \sigma_1 \land \sigma'' \xrightarrow{n(\bar{b})/r_2^n} \sigma_2 \}
\]

Return value agreement \textit{rvagree} is a predicate indicating that all such post-states agree on return values:

\[
\text{rvagree}(\sigma, m, \tilde{a}, n, \tilde{b}) \equiv \{ (\sigma_1, \sigma_2) \mid \sigma' \xrightarrow{m(\bar{a})/r_1^m} \sigma_1 \land \sigma'' \xrightarrow{n(\bar{b})/r_2^n} \sigma_2 \}
\]

The idea of \textit{mm}-differencing can be visualized as follows:

On the left, we start with states \(\sigma_1\) and \(\sigma_2\) that are exactly equal. The product program leads to \(\sigma_1'\) and \(\sigma_2'\). For these post states, we require return value agreement: \(X' \equiv r_1^{m(\bar{a})/r_1^n} \land r_2^{n(\bar{a})/r_2^m}\). Next, we have an abstraction \(\alpha_n\) specific to this \(m/n\) pair, and a product program in this abstract domain.

The key idea is that (i) relation \(R_{\alpha}\) relates abstract post-states whose return values agree in the abstract domain, and (ii) \(\alpha\) is required to be precise enough that return values agree for
all state pairs in the concretization of $R_\alpha$. We can then check whether an initial assumption of $\varphi_n^m$ on $\sigma_1$ implies such an $R_\alpha$, i.e., checking return value agreement using $\alpha$ which is just precise enough to do so. For $\text{isin}(x)/\text{clear}$, let $\alpha$ be a predicate abstraction, with predicates $\{a = x, a \neq x, b = x, b \neq x\}$ that tracks whether $x$ is in the set. Then $R_\alpha(\sigma_1, \sigma_2) \equiv (a = x)_1 \lor (b = x)_1 \leftrightarrow (a = x)_2 \lor (b = x)_2$, i.e. the relation that tracks if $\sigma_1$ and $\sigma_2$ agree on those predicates. Formally, $mn$-differencing is defined as:

**Definition V.1.** For an object with state space $\Sigma$, and two methods $m$ and $n$. Let $\alpha : \Sigma \rightarrow \Sigma^m$ be an abstraction of the states, and $\gamma : \Sigma^m \rightarrow P(\Sigma)$ the corresponding concretization. Let $R_\alpha : \Sigma^m \rightarrow \Sigma^m$ be a relation on abstract states. We say that $(\alpha, R_\alpha)$ is an $mn$-differencing abstraction if

$$\forall \sigma_1, \sigma_2 \in \Sigma^m. R_\alpha(\sigma_1, \sigma_2) \implies \forall \sigma \bar{a} \bar{b}. \text{posts}(\sigma, m, \bar{a}, \bar{b}, \bar{n}) \in \gamma(\sigma_1^m) \times \gamma(\sigma_2^m) \implies \text{rvsagree}(\sigma, m, \bar{a}, \bar{b})$$

A relation $R_\alpha$ may not hold for every initial state $\sigma$, hence the need for a commutativity condition. So we need to ask whether $R_\alpha$ holds, under the assumption that $\varphi_n^m$ holds in the pre-condition, defined as follows:

**Definition V.2.** Let $(\alpha, R_\alpha)$ be an $mn$-differencing abstraction and $\varphi_n^m$ a logical formula on concrete states and actions of $m$ and $n$. We say $\varphi_n^m$ implies $(\alpha, R_\alpha)$ if $\forall \sigma \bar{a} \bar{b} \bar{r} \bar{s}. \varphi_n^m(\sigma, \bar{a}, \bar{b}, \bar{r}, \bar{s}) \implies R_\alpha(\sigma(1), \sigma(2))$ where $(\sigma(1), \sigma(2)) \in \text{posts}(\sigma, m, \bar{a}, \bar{b}, \bar{n})$.

If we let $\varphi_{\text{clear}}(\varphi_{\text{init}}(x)) \equiv a \neq x \land b \neq x$, this will imply $R_\alpha$ in the $\text{posts}$: the post states will agree on whether $x$ is in the set.

**Post-state equivalence.** States that are $R_\alpha$-related are not necessarily equivalent. We identify the next stage of reasoning with observing equivalence relations and separate abstractions there for. This is fairly standard definition of observational equivalence relations [32]. Importantly, we can use an abstraction $\beta$ here that is separate from $\alpha$. Formally,

**Definition V.3.** Let $\beta : \Sigma \rightarrow \Sigma^\beta$ be an abstraction, with concretiz. $\delta : \Sigma^\beta \rightarrow P(\Sigma)$, and let $I_\beta : \Sigma^\beta \times \Sigma^\beta \rightarrow \{0, 1\}$. $I_\beta$ is an observational equivalence relation iff: $\forall \sigma_1^\beta, \sigma_2^\beta \in \Sigma^\beta. I_\beta(\sigma_1^\beta, \sigma_2^\beta) \rightarrow \forall \sigma_1 \in \delta(\sigma_1^\beta), \sigma_2 \in \delta(\sigma_2^\beta), \sigma_1 \simeq \sigma_2$

$I_{SS}$ and $I_{AS}$, defined earlier, are such relations. We connect $R_\alpha$ with $I_\beta$ as follows:

**Definition V.4.** We say $(\alpha, R_\alpha)$ implies $(\beta, I_\beta)$ iff:

$$\forall \sigma_1, \sigma_2 \in \Sigma. R_\alpha(\alpha(1), \alpha(2)) \rightarrow I_\beta(\beta(1), \beta(2))$$

To satisfy this implication, $R_\alpha$ may need to be more precise than simply witnessing return value agreement. In the case of SimpleSet, $R_\alpha$ must be refined so that it also implies that $s_1 = s_2$ and $(a_1 = a_2 \land b_1 = b_2) \lor (a_1 = b_2 \land a_2 = b_1)$. Finally, sufficient conditions for a commutativity condition for the two methods, with respect to these abstractions are:

**Theorem VI.1.** Let $\varphi_n^m$ be a logical formula on $\Sigma$ and actions of $m$ and $n$. If there exists $(\alpha_n^m, R_n^m)$ and $(\beta, I_\beta)$, that $\varphi_n^m$ implies $(\alpha_n^m, R_n^m)$ and $(\beta, I_\beta)$ then $\varphi_n^m$ is a commutativity condition. (Proof sketch in Apx. B.)

**VI. Reachability and Differencing Abstractions**

We now employ $mn$-differencing abstractions to introduce DAREDEUCE$^{m,n}_a$ that decomposes reasoning into two phases: (A) finding a sufficient $R_\alpha$ that implies $I_\beta$ and then (B) proving that $I_\beta$ is an observational equivalence relation. In short, commutativity proving is reduced to the question: $\exists I \forall R. A_\alpha(\varphi_n^m, R, I)$ is safe and $A_B(\varphi_n^m)$ is safe. This allows tools separately synthesize abstractions $\beta$ and $\alpha$, each targeted to the reachability needs of the phase. Moreover, when operating on the automata from DAREDEUCE$^{m,n}_a$ tools can sometimes synthesize $I$ and, in all cases we evaluated, tools synthesized $R$. Finally, $A_B(\varphi_n^m)$ turns out to be independent of method $m$, and can be proved method once for the ADT and then commutativity conditions only need to be plugged into $A_\alpha$.

**Definition VI.1 (DAREDEUCE$^{m,n}_a$).** For an input object CFA $A_o$, DAREDEUCE when applied to methods $m(\bar{x}), n(\bar{y})$, yields two automata $A(\varphi_n^m, I)_A$ and $A(I)_B$ defined as follows:

**Phase A:** Proving that $\varphi_n^m \Rightarrow I$ via $R_\alpha$:

$A(\varphi_n^m, I)_A = \langle Q_A, q_0^A, X_A, s_A, \longrightarrow_A \rangle$ where $\longrightarrow_A$ is:

- $q_0^A \xrightarrow{\text{inl}(\text{init}, \text{nil}, [], [o_1])} q_1$ (o’s source) (Construct.)
- $q_1 \xrightarrow{\text{rvsagree}(\sigma_1, m, \bar{a}, \bar{b})} q_1 \xrightarrow{\text{inl}(\text{init}, m, \bar{a}, \text{nil})} q_1$ (Reachbl. $o_1$)

**Phase B:** Proving that $I$ is an obs. eq. relation via $\beta$:

$A(I)_B = \langle Q_B, q_0^B, X_B, s_B, \longrightarrow_B \rangle$ where $\longrightarrow_B$ is:

- $q_0^B \xrightarrow{\text{assume}(\bar{a})} q_1$ (o’s source)
- $q_2 \xrightarrow{\text{inl}(\text{clone}, [o_1], [], [\bar{a}, \bar{b}])} q_2$ (Clone $o_1$)
- $q_2 \xrightarrow{\text{inl}(m_1, \bar{a}, \bar{b}, \bar{r})} q_2 \xrightarrow{\text{inl}(m_2, \bar{a}, \bar{b}, \bar{r})} q_3$ (Any $m_1$)

The definitions for $Q_A$, initial state $q_0^A$, variables $X_A$, statements $s_A$ are straightforward. Same for $A(I)_B$.

The DAREDEUCE$^{m,n}_a$ output encoding $A(\varphi_n^m, I)_A$ (Phase A), like REDUCE$^m_a$, begins with a region that ensures that, for any reachable state of the ADT, there will be a run of $A(\varphi_n^m, I)_A$ to location $q_1$ where $o$ is in that reachable state. It also sets up the preconditions that $\varphi_n^m$ hold and that $o_1 = o_2$ (concretely) and constructs the product program between the two orders of method application. Unlike REDUCE$^{m,n}_a$, $A(\varphi_n^m, I)_A$ ends with a possible transition to $q_{\text{exit}}^A$ whenever $\neg I$ holds.
To prove that \( \neg I \) can never hold, an analysis on \( A ( \varphi_\text{m}^n, I)_A \) must establish an invariant at \( q_3 \) that is (1) strong enough to capture return value agreement and (2) strong enough to imply \( I \). This invariant will indeed be an \( mn \)-differencing relation due to (1), and it will imply \( I \) due to (2).

The \( \text{DAREDUCE}_m \) output \( A (I)_B \) (Phase B) is designed so that a safety proof on \( A (I)_B \) entails that \( I \) is an observational equivalence relation. There is a pre-condition edge that assumes \( I \), and edges to the entry node of each possible method \( m_i \), nondeterministically selecting arguments. To prove that \( I \) is an observational equivalence relation, a reachability solver will synthesized an appropriate abstraction \( \beta \).

**Theorem VI.1.** For object \( \text{CFA} \ A_o \), and automata \( A ( \varphi_\text{m}^n, I)_A \) and \( A (I)_B \) resulting from \( \text{DAREDUCE}_m \), if there exists an \( I \) such that for every run of \( A ( \varphi_\text{m}^n, I)_A \) avoids \( q^A_B \), and every run of \( A (I)_B \) avoids \( q^B \), then \( \varphi_\text{m} \) is a commutativity condition.

**Proof.** Similar to the proof of Theorem IV.1 by employing Theorem V.1 to combine phases.

\( \text{DAREDUCE}_m \) improves over \( \text{REDUCE}_m \) by decomposing the verification problem, making it more tractable to automation (see Sec. VIII). One can employ a \( mn \)-differencing abstraction \( \alpha \) for finding \( R_o \) and a separate abstraction \( \beta \) for observational equivalence. Phase B does not depend on the method pair under consideration. Consequently, a proof of safety of \( A (I)_B \) can be done once for the entire ADT.

**VII. Implementation**

We developed a simple prototype implementation called \( \text{CITYPROVER}^2 \). \( \text{CITYPROVER} \) is written in Perl and OCaml and takes, as input, C source code. Examples input ADTs can be found in Appendix E. We have written them as C macros so that our experiments focus on commutativity rather than testing existing tools’ inter-procedural reasoning. Also provided as input is a commutativity condition \( \varphi_\text{m}^n \). \( \text{CITYPROVER} \) then implements \( \text{REDUCE}_m \) and \( \text{DAREDUCE}_m \) via a program transformation.

**VIII. Experiments**

There are no other existing tools for verifying ADT commutativity directly from source code. Our goals were to evaluate whether our reductions enable existing reachability solvers to verify commutativity conditions and whether \( \text{DAREDUCE}_m \) performs better than \( \text{REDUCE}_m \). To this end, we created some small examples (with integers, simple pointers, structs and arrays) and ran \( \text{CITYPROVER} \) on them. Our experiments were run on a Quad-Core Intel(R) Xeon(R) CPU E3-1220 v6 at 3.00 GHz, inside a QEMU virtual host.

We began with simple ADTs including: a Memory cell; an Accumulator with increment, decrement, and a check whether the value is 0; and a Counter that also has a clear method. For each object, we considered some example method pairs with both a valid commutativity condition and an incorrect

To be released on GitHub. See https://file.io/V0GSL5Qe.

\[ \begin{array}{|c|c|c|c|c|c|}
\hline
\text{ADT} & \text{Methods} & \varphi_\text{m}^n & \text{Exp.} & \text{REDUCE}_m & \text{DAREDUCE}_m \\
\hline
\text{Memory} & \text{rd \& wr} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\text{Memory} & \text{wr \& rd} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\text{Accum.} & \text{inc \& isz} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\text{Accum.} & \text{inc \& inc} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\text{Counter} & \text{dec \& dec} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\text{Counter} & \text{inc \& inc} & \varphi_\text{m}^n & \text{oneshot} & \text{CPS} & \text{Ult} \\
\hline
\end{array} \]
a) Commutativity reasoning: Bansal et al. [14] synthesize commutativity conditions from provided pre/post specifications, rather than implementations. They assume these specifications are precise enough to faithfully represent all effects relevant to commutativity. As discussed in Section II, if specifications are coarse, Bansal et al. would emit unsound commutativity conditions. On the other hand, precise specifications are harder to come by (by hand or by static analysis) because the precision needed may be tantamount to full functional correctness specifications. We instead capture just what is needed for commutativity.

Gehr et al. [11] describe a method based on black-box sampling. Both Aleen and Clark [12] and Tripp et al. [33] identify sequences of actions that commute (via random interpretation and dynamic analysis, resp.). Kim and Rinard [13] verify commutativity conditions from specifications. Commutativity is also used in dynamic analysis [28]. Najafzadeh et al. [34] describe a tool for weak consistency, that reports commutativity checking of formulae, but not ADT implementations.

b) k-safety, product programs, reductions.: Reduction to reachability have been used in security. Self-composition [15], [16]—reduces (some forms of) hyper-properties [35] to properties of a single program. More recent works include product programs [17], [36] and a number of techniques for automated verification of k-safety properties. Cartesian Hoare Logic [18] is a program logic for reasoning about k-safety properties, automated via a tool called DESCARTES. Antonopoulos et al. [37] described an alternative automated k-safety technique based on partitioning the traces. Farzan and Vendikas [19] discuss a technique and tool WEAVER for verifying hypersafety properties, based on the observation that a proof of some representative runs in a product program can be sufficient to prove that the hypersafety property holds of the original program. Others explore logical approaches to relational reasoning across multiple programs [38], [39].

X. Conclusion

We have described a theory, algorithm and tool for automatically verify commutativity conditions of data structure implementations. The key insight is that mn-differencing relations can be used to target commutativity reasoning and this can be employed in reduction DAREUCE\textsuperscript{m} to decompose the problem to make it more amenable to general-purpose reachability solvers. Our proofs can enable commutativity conditions to be more safely integrated into various concurrency settings. In the future work, we hope to explore techniques for more complex data-structures, especially those with layout permutations.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback on earlier drafts of this work.
A commutativity condition. The key features of $A$ are:

1) Reachable ADT states. For any reachable object state $q$, there will be some run of $A(q_m)$ that witnesses that state in a valuation $\theta$ at $q_1$. This is accomplished by edges from $q_1$ into the entry node of each possible $m_i$, first letting $Q_E$ non-deterministically set arguments $\bar{x}$.

2) Commutativity pre-condition. From $q_1$, non-deterministic choices are made for the method arguments $m(\bar{a})$ and $n(\bar{b})$, and then candidate condition $\varphi^n$ is assumed.

3) Product program. The edge to $q_2$ causes a run to clone $o_1$ to $o_2$, invoke $m(\bar{a})$; $n(\bar{b})$ on $o_1$ and invoke $n(\bar{b})$; $m(\bar{a})$ on $o_2$.

4) Return values. From $q_3$, there is an edge to $q_{er}$ which is feasible if the return values disagree.

5) Obs. Equivalence. From $q_4$, for any possible method $m_i$, there is an edge with a statement $\bar{a} := \bar{a}$ to choose non-deterministic values, and then invoke $m_i(\bar{a})$ on both

$\varphi^n$ and $Q_E$ is the union of $Q_o$ and all CFA nodes above, $s_E$ is the union of $s$ and all additional statements above, and $X_E = X_o \cup \{o_1, o_2, a, b, r_m, r_n, r^1_m, r^2_m, r^1_n, r^2_n, r^1_m, r^2_n, m, n\}$.

Fig. 5. One-shot reduction $\text{REDUCE}^n_m$ emits automaton $A(q_m^n)$ defined above.

## Appendix

### A. Transformation $\text{REDUCE}^n_m$

We now define $\text{REDUCE}^n_m$. Below we let $\text{assume}(\bar{x} \neq \bar{y})$ mean the disjunction of inequality between corresponding vector elements.

**Definition A.1** (Transformation). For an input object CFA $A_o = (Q_o, [q_0, q_0, \ldots, q_0, X_o, s, \ldots])$, the output of $\text{REDUCE}^n_m(A_o, m(\bar{x}), n(\bar{y}))$ is automaton $A(q_m^n)$ given in Figure 5.

In Figure 5 node $q_0^n$ is the initial node of the automaton. The transformation employs the implementation source code of the data structure CFA, given by $\rightarrow$. The key theorem below says that non-reachability of $q_{er}$ entails that $\varphi^n$ is a commutativity condition. The key features of $A(q_m^n)$ are:

1) **Reachable ADT states**. For any reachable object state $q$, there will be some run of $A(q_m^n)$ that witnesses that state in a valuation $\theta$ at $q_1$. This is accomplished by edges from $q_1$ into the entry node of each possible $m_i$, first letting $Q_E$ nondeterministically set arguments $\bar{x}$.

2) **Commutativity pre-condition**. From $q_1$, non-deterministic choices are made for the method arguments $m(\bar{a})$ and $n(\bar{b})$, and then candidate condition $\varphi^n$ is assumed.

3) **Product program**. The edge to $q_2$ causes a run to clone $o_1$ to $o_2$, invoke $m(\bar{a})$; $n(\bar{b})$ on $o_1$ and invoke $n(\bar{b})$; $m(\bar{a})$ on $o_2$.

4) **Return values**. From $q_3$, there is an edge to $q_{er}$ which is feasible if the return values disagree.

5) **Obs. Equivalence**. From $q_4$, for any possible method $m_i$, there is an edge with a statement $\bar{a} := \bar{a}$ to choose non-deterministic values, and then invoke $m_i(\bar{a})$ on both $o_1$ and $o_2$. If it is possible for the resulting return values to disagree, then a run could proceed to $q_{er}$.

### B. Proof of Theorem V.1

From Definition V.2, we have that $R$ holds for the $\alpha$-abstraction of post states, and then from Definition V.1 it follows that the return values agree. On the other hand, from Definition V.4 it follows that $I_\beta$ holds for the $\beta$-abstraction of post states as well, and from Definition V.3 it follows that the (concrete) post states are observationally equivalent.
### C. Detailed version of Figure 3

<table>
<thead>
<tr>
<th>ADT</th>
<th>Methods</th>
<th>$m(x_1), n(y_1)$</th>
<th>Exp.</th>
<th>CPA</th>
<th>Ult</th>
<th>CPA</th>
<th>DAREDUC$^m_n$</th>
<th>mm-differencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>read $\bowtie$ write</td>
<td>$s_1.x = y_1$</td>
<td>✓</td>
<td>1.4</td>
<td>0.7</td>
<td>1.3</td>
<td>✓ + 1.3 (+) + oe = 3.9</td>
<td>Ø ✓ + 0.3 (+) + oe = 1.3</td>
</tr>
<tr>
<td>Memory</td>
<td>read $\bowtie$ write</td>
<td>true</td>
<td>𝛾</td>
<td>1.4</td>
<td>0.2</td>
<td>1.3</td>
<td>(+) + n/a(+) + oe = 1.3</td>
<td>0.2 (+) + n/a(+) + oe = 0.2</td>
</tr>
<tr>
<td>Memory</td>
<td>write $\bowtie$ write</td>
<td>$y_1 = x_1$</td>
<td>✓</td>
<td>1.4</td>
<td>0.5</td>
<td>1.3</td>
<td>(+) + 1.3 (+) + oe = 3.9</td>
<td>0.2 (+) + 0.3 (+) + oe = 0.8</td>
</tr>
<tr>
<td>Memory</td>
<td>write $\bowtie$ write</td>
<td>true</td>
<td>𝛾</td>
<td>1.3</td>
<td>0.3</td>
<td>1.2</td>
<td>(+) + 1.2 (+) + oe = 2.4</td>
<td>0.2 (+) + 0.2 (+) + oe = 0.4</td>
</tr>
<tr>
<td>Memory</td>
<td>read $\bowtie$ read</td>
<td>true</td>
<td>✓</td>
<td>1.4</td>
<td>0.6</td>
<td>1.3</td>
<td>(+) + 1.3 (+) + oe = 3.9</td>
<td>0.3 (+) + 0.3 (+) + oe = 1.3</td>
</tr>
<tr>
<td>Accum.</td>
<td>decr $\bowtie$ isz</td>
<td>$s_1.x &gt; 1$</td>
<td>✓</td>
<td>1.5</td>
<td>2.2</td>
<td>1.3</td>
<td>(+) + 1.4 (+) + oe = 4</td>
<td>0.7 (+) + 1.0 (+) + oe = 2.6</td>
</tr>
<tr>
<td>Accum.</td>
<td>decr $\bowtie$ isz</td>
<td>true</td>
<td>𝛾</td>
<td>1.5</td>
<td>0.7</td>
<td>1.2</td>
<td>(+) + n/a(+) + oe = 1.2</td>
<td>0.6 (+) + n/a(+) + oe = 0.6</td>
</tr>
<tr>
<td>Accum.</td>
<td>incr $\bowtie$ incr</td>
<td>$s_1.x &gt; 1$</td>
<td>✓</td>
<td>1.5</td>
<td>1.3</td>
<td>1.3</td>
<td>(+) + 1.5 (+) + oe = 4.1</td>
<td>0.3 (+) + 0.4 (+) + oe = 1.7</td>
</tr>
<tr>
<td>Accum.</td>
<td>incr $\bowtie$ isz</td>
<td>true</td>
<td>𝛾</td>
<td>1.5</td>
<td>1.2</td>
<td>1.3</td>
<td>(+) + 1.4 (+) + oe = 4</td>
<td>0.3 (+) + 0.3 (+) + oe = 1.5</td>
</tr>
<tr>
<td>Accum.</td>
<td>incr $\bowtie$ incr</td>
<td>true</td>
<td>𝛾</td>
<td>1.4</td>
<td>1.5</td>
<td>1.4</td>
<td>(+) + 1.4 (+) + oe = 4</td>
<td>0.3 (+) + 0.3 (+) + oe = 1.6</td>
</tr>
<tr>
<td>Accum.</td>
<td>decr $\bowtie$ incr</td>
<td>true</td>
<td>𝛾</td>
<td>1.5</td>
<td>1.5</td>
<td>1.3</td>
<td>(+) + 1.3 (+) + oe = 3.9</td>
<td>0.3 (+) + 0.4 (+) + oe = 1.6</td>
</tr>
<tr>
<td>Accum.</td>
<td>decr $\bowtie$ decr</td>
<td>$s_1.x &gt; 1$</td>
<td>✓</td>
<td>1.5</td>
<td>2.6</td>
<td>1.3</td>
<td>(+) + 1.4 (+) + oe = 4</td>
<td>0.3 (+) + 0.6 (+) + oe = 1.9</td>
</tr>
<tr>
<td>Accum.</td>
<td>isz $\bowtie$ isz</td>
<td>true</td>
<td>✓</td>
<td>1.4</td>
<td>4.3</td>
<td>1.4</td>
<td>(+) + 1.3 (+) + oe = 4</td>
<td>1.8 (+) + 0.7 (+) + oe = 3.4</td>
</tr>
<tr>
<td>Counter</td>
<td>decr $\bowtie$ decr</td>
<td>true</td>
<td>𝛾</td>
<td>1.9</td>
<td>1.5</td>
<td>1.4</td>
<td>(+) + 1.4 (+) + oe = 4.2</td>
<td>1.2 (+) + n/a(+) + oe = 1.2</td>
</tr>
<tr>
<td>Counter</td>
<td>decr $\bowtie$ decr</td>
<td>$s_1.x &gt;= 2$</td>
<td>✓</td>
<td>1.5</td>
<td>13.0</td>
<td>1.3</td>
<td>(+) + 1.4 (+) + oe = 4.1</td>
<td>2.0 (+) + 2.4 (+) + oe = 5.9</td>
</tr>
<tr>
<td>Counter</td>
<td>decr $\bowtie$ incr</td>
<td>true</td>
<td>𝛾</td>
<td>1.6</td>
<td>0.3</td>
<td>1.4</td>
<td>(+) + n/a(+) + oe = 1.4</td>
<td>0.3 (+) + n/a(+) + oe = 0.3</td>
</tr>
<tr>
<td>Counter</td>
<td>decr $\bowtie$ incr</td>
<td>$s_1.x &gt;= 1$</td>
<td>✓</td>
<td>1.6</td>
<td>6.8</td>
<td>1.4</td>
<td>(+) + 1.4 (+) + oe = 4.2</td>
<td>1.5 (+) + 1.0 (+) + oe = 3.8</td>
</tr>
<tr>
<td>Counter</td>
<td>incr $\bowtie$ isz</td>
<td>true</td>
<td>𝛾</td>
<td>1.5</td>
<td>0.8</td>
<td>1.2</td>
<td>(+) + n/a(+) + oe = 1.2</td>
<td>0.7 (+) + n/a(+) + oe = 0.7</td>
</tr>
<tr>
<td>Counter</td>
<td>incr $\bowtie$ isz</td>
<td>$s_1.x &gt; 0$</td>
<td>✓</td>
<td>1.5</td>
<td>5.3</td>
<td>1.3</td>
<td>(+) + 1.4 (+) + oe = 4.1</td>
<td>0.6 (+) + 0.5 (+) + oe = 2.6</td>
</tr>
<tr>
<td>Counter</td>
<td>incr $\bowtie$ isz</td>
<td>$s_1.x &gt; 0$</td>
<td>✓</td>
<td>1.5</td>
<td>?</td>
<td>1.4</td>
<td>(+) + 1.5 (+) + oe = 4.4</td>
<td>1.1 (+) + 2.3 (+) + oe = 6.9</td>
</tr>
<tr>
<td>Counter</td>
<td>incr $\bowtie$ clear</td>
<td>true</td>
<td>𝛾</td>
<td>1.3</td>
<td>0.4</td>
<td>1.3</td>
<td>(+) + 1.2 (+) + oe = 2.5</td>
<td>0.2 (+) + 0.2 (+) + oe = 0.4</td>
</tr>
</tbody>
</table>

Extended version of Fig. 3. Results of applying CITYPROVER to the simple benchmarks. For each benchmark, we report the time to use both REDUCE$^m_n$ and DAREDUC$^m_n$, using either CPAchecker [26] or Ultimate [20] to solve the reachability tasks. Note that $s_1.x$ is the object field, and $x_1, y_1$ are $m, n$ parameters, resp.
### D. Detailed version of Figure 4

<table>
<thead>
<tr>
<th>ADT</th>
<th>Methods</th>
<th>n(x)</th>
<th>n(y)</th>
<th>Exp.</th>
<th>Ult</th>
<th>Ult</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleSet</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>isin</td>
<td>true</td>
<td>✓</td>
<td>137.8</td>
<td>13.6 (✓) + 6.3 (✓) + oe = 36.7 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>add</td>
<td>x₁ ≠ y₁</td>
<td>✓</td>
<td>84.8</td>
<td>11.4 (✓) + 8.9 (✓) + oe = 37.1 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>add</td>
<td>true</td>
<td>✓</td>
<td>2.4</td>
<td>1.5 (✓) + n/a(n/a) + oe = 1.5 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>clear</td>
<td>true</td>
<td>✓</td>
<td>2.6</td>
<td>1.6 (✓) + n/a(n/a) + oe = 1.6 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>clear</td>
<td>x₁ ≠ y₁</td>
<td>✓</td>
<td>2.4</td>
<td>1.6 (✓) + n/a(n/a) + oe = 1.6 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>clear</td>
<td>a₁ ≠ x₁ ∧ s₁.b ≠ y₁</td>
<td>✓</td>
<td>3.1</td>
<td>1.4 (✓) + n/a(n/a) + oe = 1.4 ✓</td>
</tr>
<tr>
<td>SimpleSet</td>
<td>isin</td>
<td>getsize</td>
<td>true</td>
<td>✓</td>
<td>14.0</td>
<td>0.9 (✓) + 1.5 (✓) + oe = 19.3 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>pop</td>
<td>a₁[top₁] = x₁∧top₁ &gt; 1∧top₁ &lt; 5 – 1</td>
<td>✓</td>
<td>MO –</td>
<td>34.6 (✓) + 25.5 (✓) + oe = 95.5 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>pop</td>
<td>true</td>
<td>✓</td>
<td>2.2</td>
<td>2.0 (✓) + n/a(n/a) + oe = 2.0 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>push</td>
<td>true</td>
<td>✓</td>
<td>7.6</td>
<td>17.0 (✓) + n/a(n/a) + oe = 17.0 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>push</td>
<td>top₁ &lt; 3</td>
<td>✓</td>
<td>3.9</td>
<td>229.3 (✓) + 1.4 (✓) + oe = 230.7 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>push</td>
<td>x₁ = y₁</td>
<td>✓</td>
<td>19.8</td>
<td>17.3 (✓) + n/a(n/a) + oe = 17.3 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>push</td>
<td>push</td>
<td>x₁ = y₁ ∧ top₁ &lt; 3</td>
<td>✓</td>
<td>MO –</td>
<td>3.6 (✓) + 116.3 (✓) + oe = 155.1 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>pop</td>
<td>pop</td>
<td>top₁ = -1</td>
<td>✓</td>
<td>TO –</td>
<td>0.7 (✓) + 2.0 (✓) + oe = 38.0 ✓</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>pop</td>
<td>pop</td>
<td>true</td>
<td>✓</td>
<td>2.1</td>
<td>1.4 (✓) + n/a(n/a) + oe = 1.4 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>enq</td>
<td>true</td>
<td>✓</td>
<td>39.8</td>
<td>35.0 (✓) + n/a(n/a) + oe = 35.0 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>deq</td>
<td>deq</td>
<td>true</td>
<td>✓</td>
<td>3.5</td>
<td>3.1 (✓) + n/a(n/a) + oe = 3.1 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>deq</td>
<td>deq</td>
<td>size₁ = 0</td>
<td>✓</td>
<td>TO –</td>
<td>2.2 (✓) + 63.1 (✓) + oe = 174.4 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>enq</td>
<td>true</td>
<td>✓</td>
<td>27.8</td>
<td>23.3 (✓) + n/a(n/a) + oe = 23.3 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>enq</td>
<td>x₁ = y₁</td>
<td>✓</td>
<td>63.8</td>
<td>22.6 (✓) + n/a(n/a) + oe = 22.6 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>isempty</td>
<td>isempty</td>
<td>true</td>
<td>✓</td>
<td>TO –</td>
<td>6.6 (✓) + TO (?) + oe = TO –</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>deq</td>
<td>size₁ = 1 ∧ x₁ = a₁[front₁]</td>
<td>✓</td>
<td>TO –</td>
<td>2.3 (✓) + 360.6 (✓) + oe = 472.0 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>deq</td>
<td>true</td>
<td>✓</td>
<td>MO –</td>
<td>8.4 (✓) + n/a(n/a) + oe = 8.4 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>isempty</td>
<td>size₁ &gt; 0</td>
<td>✓</td>
<td>MO –</td>
<td>6.1 (✓) + TO (?) + oe = TO –</td>
</tr>
<tr>
<td>Queue</td>
<td>enq</td>
<td>isempty</td>
<td>true</td>
<td>✓</td>
<td>MO –</td>
<td>7.4 (✓) + n/a(n/a) + oe = 7.4 ✓</td>
</tr>
<tr>
<td>Queue</td>
<td>deq</td>
<td>isempty</td>
<td>size₁ = 0</td>
<td>✓</td>
<td>MO –</td>
<td>2.1 (✓) + 0.8 (✓) + oe = 135.8 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>HashTable</td>
<td>put</td>
<td>put</td>
<td>1,&lt;sup&gt;1&lt;/sup&gt;&lt;sub&gt;put&lt;/sub&gt;</td>
<td>✓</td>
<td>MO –</td>
<td>262.5 (✓) + n/a(n/a) + oe = 97.5 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>put</td>
<td>put</td>
<td>2,&lt;sup&gt;2&lt;/sup&gt;&lt;sub&gt;put&lt;/sub&gt;</td>
<td>✓</td>
<td>202.7</td>
<td>136.9 (✓) + n/a(n/a) + oe = 136.9 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>put</td>
<td>put</td>
<td>3,&lt;sup&gt;3&lt;/sup&gt;&lt;sub&gt;put&lt;/sub&gt;</td>
<td>✓</td>
<td>566.5</td>
<td>1.5 (✓) + 288.9 (✓) + oe = 297.5 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>put</td>
<td>put</td>
<td>true</td>
<td>✓</td>
<td>TO –</td>
<td>102.3 (✓) + n/a(n/a) + oe = 102.3 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>get</td>
<td>get</td>
<td>o₁.keys = 0</td>
<td>✓</td>
<td>TO –</td>
<td>TO (?) + n/a(n/a) + oe = TO –</td>
</tr>
<tr>
<td>HashTable</td>
<td>get</td>
<td>get</td>
<td>true</td>
<td>✓</td>
<td>TO –</td>
<td>TO (?) + n/a(n/a) + oe = TO –</td>
</tr>
<tr>
<td>HashTable</td>
<td>get</td>
<td>get</td>
<td>true</td>
<td>✓</td>
<td>TO –</td>
<td>TO (?) + n/a(n/a) + oe = TO –</td>
</tr>
<tr>
<td>HashTable</td>
<td>get</td>
<td>put</td>
<td>x₁ ≠ y₁</td>
<td>✓</td>
<td>TO –</td>
<td>50.0 (✓) + 2.0 (✓) + 47.7 (✓) + oe = 56.8 ✓</td>
</tr>
<tr>
<td>HashTable</td>
<td>get</td>
<td>put</td>
<td>true</td>
<td>✓</td>
<td>TO –</td>
<td>TO (?) + n/a(n/a) + oe = TO –</td>
</tr>
</tbody>
</table>

<sup><sup>1</sup>put</sup> = a₁.o₁[0₁.top] = x₁ ∧ o₁.top > 1 ∧ o₁.top < MAX  
<sup><sup>2</sup>put</sup> = x₁ ≠ y₁ ∧ o₁.table[x₁ % CAP]%key = -1 ∧ o₁.table[y₁ % CAP]%key = -1  
<sup><sup>3</sup>put</sup> = x₁ ≠ y₁ ∧ x₁ % CAP ≠ y₁ % CAP ∧ o₁.table[x₁ % CAP]%key = -1 ∧ o₁.table[y₁ % CAP]%key = -1

Extended version of Fig. 4. Some commutativity conditions are listed below the table. Our implementation actually subdivides Phase A into two parts: first one in which we prove return value agreement (i.e. Rₐ) and then one in which we prove that Iₜ is implied. Hence, in the DAREDUCEₐ column, we report the some of these two sub-phases plus the time for Phase B. For each benchmark, we re-ran the Phase B, even though all could have shared one run. For each ADT above, we report an example of the time to prove Aₜ(Iₜ) with both Ultimate and CPAchecker. We mostly used Ultimate but, in some of the Queue or Hashable cases, it did not perform well and we instead tried CPAchecker. Those cases are denoted in blue.

**Results.** We show that CityProver is tractable at proving commutativity conditions. Moreover, DAREDUCEₐ improves over REDUCEₐ: in most cases it is faster, sometimes by as much as 2× or 3×. In 7 cases, DAREDUCEₐ is able to generate an answer, while REDUCEₐ suffers from a timeout/memout. For Hashable, Ultimate timed out on Phase B. We still used Ultimate in some Phase A cases, because it can report a counterexample in Phase A (even if it timed out in B). We also could use Ultimate for Phase A, given that CPAchecker already proved Phase B, with the same Iₜ.
E. Benchmark Sources

Memory

```c
struct state_t { int x; }

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_clone(src) {
    struct state_t* tmp;
    o_new(tmp);
    tmp->x = (src)->x;
    dst = tmp;
}
#define o_read(st,rv) rv = (st)->x
#define o_write(st,rv,v) {
    (st)->x = v; rv = 0;
}

Counter

```c
struct state_t { int x; }

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_close(src) {
    struct state_t* tmp;
    o_new(tmp);
    tmp->x = src->x;
    dst = tmp;
}
#define o_incr(st,rv) { (st)->x += 1; rv = 0; }
#define o_clear(st,rv) { (st)->x = 0; rv = 0; }
#define o_decr(st,rv) { if ((st)->x == 0) rv = -1; else { (st)->x -= 1; rv = 0; } }
#define o_isz(st,rv) { rv = ((st)->x == 0 ? 1 : 0); }

Accumulator

```c
struct state_t { int x; }

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_close(src) {
    struct state_t* tmp;
    o_new(tmp);
    tmp->x = src->x;
    dst = tmp;
}
#define o_incr(st,rv) { (st)->x += 1; rv = 0; }
#define o_decr(st,rv) { (st)->x -= 1; rv = 0; }
#define o_isz(st,rv) { rv = ((st)->x == 0 ? 1 : 0); }

Array Queue

```c
#define MAXQUEUE 5

struct state_t { int a[MAXQUEUE]; int front; int rear; int size; }
```
```c
#include <stdlib.h>

#define o_new(res) { 
    res = malloc(sizeof(struct state_t)); 
    res->front = 0; 
    res->rear = MAXQUEUE - 1; 
    res->size = 0; 
}

#define o_enq(st,rv,v) { 
    if (((st)->size == MAXQUEUE) rv = 0; 
    else { 
        (st)->size++; (st)->rear = ((st)->rear + 1) % MAXQUEUE; (st)->a[(st)->rear] = v; rv = 1; } }

#define o_deq(st,rv) { 
    if (((st)->size==0) rv = -1; 
    else { int r = (st)->a[(st)->front]; 
        (st)->front = ((st)->front + 1) % MAXQUEUE; 
        (st)->size--; 
        rv = r; } }

#define o isempty(st,rv) rv = ( (st)->size==0 ? 1 : 0)

Stack

#define MAXSTACK 5

struct state_t { int a[MAXSTACK]; int top; };

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t)); res->top = -1;

#define o_push(st,rv,v) { 
    if (((st)->top == (MAXSTACK-1)) {rv = 0; } 
    else { (st)->top++; (st)->a[(st)->top] = v; rv = 1; } }

#define o_pop(st,rv) { 
    if (((st)->top == -1) rv = -1; 
    else rv = (st)->a[ (st)->top-- ]; }

#define o isempty(st,rv) rv = ((st)->top== -1 ? 1 : 0)

Hash Table

#define HTCAPACITY 11

struct entry_t { int key; int value; };

struct state_t { struct entry_t table[HTCAPACITY]; int keys; };

#include <stdlib.h>

#define o_new(st) { s1 = malloc(sizeof(struct state_t)); 
    for (int i=0;i<HTCAPACITY;i++) { s1->table[i].key = -1; } 
    s1->keys = 0; }

#define o_put(st,rv,k,v) { int slot = k % HTCAPACITY; 
    if (((st)->table[slot].key == -1) { 
        (st)->table[slot].key = k; 
        (st)->table[slot].value = v; 
    }
```
```c
#include <assert.h>
#include <stdlib.h>

#define o_new(st) { 
    st = malloc(sizeof(struct state_t)); 
    st->a = -1; st->b = -1; st->sz = 0; }

#define o_add(st,rv,v) { 
    if ((st)->a == -1 && (st)->b == -1) { (st)->a = v; (st)->b = (st)->b+0; (st)->sz++; rv = 0; } 
    else if ((st)->a != -1 && (st)->b == -1) { (st)->b = v; (st)->a = (st)->a+0; (st)->sz++; rv = 0; } 
    else if ((st)->a == -1 && (st)->b != -1) { (st)->a = v; (st)->b = (st)->b+0; (st)->sz++; rv = 0; } 
    else { rv = 0; }
}

#define o_isin(st,rv,v) { 
    rv = 0; 
    if ((st)->a == v) rv = 1; 
    if ((st)->b == v) rv = 1; }

#define o_getsize(st,rv) { 
    rv = (st)->sz; }

#define o_clear(st,rv) { 
    (st)->a = -1; (st)->b = -1; (st)->sz = 0; rv = 0; }

#define o_norm(st,rv) { 
    if ((st)->a > (st)->b) { 
        int t = (st)->b;
        (st)->b = (st)->a;
        (st)->a = t;
    } 
    rv = 0; }
```

### Simple Set

```c
struct state_t { int a; int b; int sz; };

#include <assert.h>
#include <stdlib.h>

#define o_new(st) { 
    st = malloc(sizeof(struct state_t)); 
    st->a = -1; st->b = -1; st->sz = 0; }

#define o_add(st,rv,v) { 
    if ((st)->a == -1 && (st)->b == -1) { (st)->a = v; (st)->b = (st)->b+0; (st)->sz++; rv = 0; } 
    else if ((st)->a != -1 && (st)->b == -1) { (st)->b = v; (st)->a = (st)->a+0; (st)->sz++; rv = 0; } 
    else if ((st)->a == -1 && (st)->b != -1) { (st)->a = v; (st)->b = (st)->b+0; (st)->sz++; rv = 0; } 
    else { rv = 0; }
}

#define o_isin(st,rv,v) { 
    rv = 0; 
    if ((st)->a == v) rv = 1; 
    if ((st)->b == v) rv = 1; }

#define o_getsize(st,rv) { 
    rv = (st)->sz; }

#define o_clear(st,rv) { 
    (st)->a = -1; (st)->b = -1; (st)->sz = 0; rv = 0; }

#define o_norm(st,rv) { 
    if ((st)->a > (st)->b) { 
        int t = (st)->b;
        (st)->b = (st)->a;
        (st)->a = t;
    } 
    rv = 0; }
```