Online Verification of Commutativity

Anonymous Author(s)

Abstract

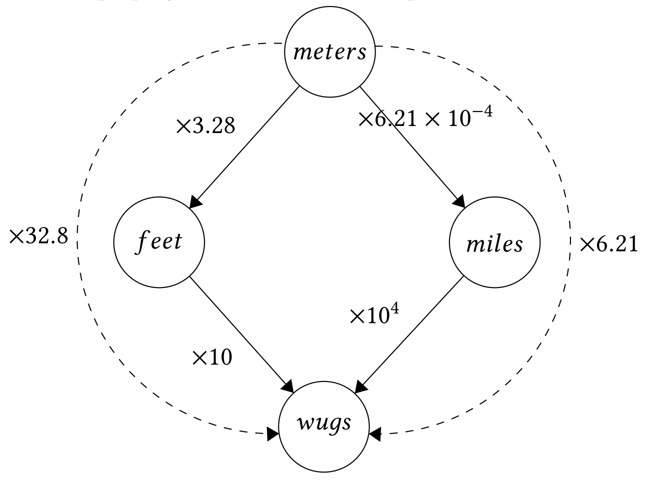
Systems of transformations arise in many programming systems, such as in type graphs of implicit type conversion functions. It is important to ensure that these diagrams commute: that any composing path of transformations from the same source to the same destination yields the same result. However, a straightforward approach to verifying commutativity must contend with cycles, and even so it runs in exponential time. Previous work has shown how to verify commutativity in the special case of acyclic diagrams in $O(|V|^4|E|^2)$ time, but this is a *batch* algorithm: the entire diagram must be known ahead of time. We present an *online* algorithm that efficiently verifies that a commutative diagram remains commutative when adding a new edge. The new incremental algorithm runs in $O(|V|^2(|E|+|V|))$ time. For the case when checking the equality of paths is expensive, we also present an optimization that runs in $O(|V|^4)$ time but reduces to the minimum possible number of equality checks. We implement the algorithms and compare them to batch baselines, and we demonstrate their practical application in the compiler of a domain-specific language for geometry types. To study the algorithms' scalability to large diagrams, we apply them to discover discrepancies in currency conversion graphs.

1 Introduction

Many systems can be understood as diagrams: graphs where nodes represent domains and edges represent directed transformation functions between nodes. A type system with coercions, for example, corresponds to a graph whose nodes are types and whose edges are coercions. Figure 1 illustrates an example in a simple language with units-of-measure types, such as F# [4] or Frink [1]. In such a system, an important correctness criterion is that the diagram *commutes:* when traversing the graph from any start node to any end node, applying every transformation along the path to any input value, the result is the same output value *independent of the path chosen between the two nodes.* With our type coercion example, it is a problem if casting to a supposedly equivalent type as an intermediate step resulted in a different answer than a direct cast. xxx [Next, let's make the example specific: say that the expression "(wugs) a", for example, can produce value X or value Y, depending on whether the compiler uses path A or path B to do the conversion. —AS]

```
var a : meters = 1;
var b : miles = (miles) a;
var c : feet = (feet) a;
define wugs:
    1 mile = 10000 wugs;
    1 foot = 10 wugs;
var d : wugs = (wugs) a;
```

(a) A sample program with user defined type conversion.



(b) Diagram for the type conversions in the program

Figure 1. In this sample program, the user implicitly defines two ways to cast variable a from meters to the new unit wugs. The definitions are different, and a compiler performing implicit conversion would not know which to choose.

This paper is about efficiently checking diagrams that arise in real systems for commutativity. We assume a simple equivalence checker for individual transformation functions: in our type system example, for instance, it is possible to check transformation equivalence by comparing the conversion factors. Our aim is to analyze the graph of transformations and minimize the number of times we need to perform an equivalence check. Since diagrams may change over time in real systems, as new conversions are added, and verifying the entire system from scratch may be computationally expensive, we want an online method that only checks the impact of new edges. In Figure 1, for example, a run-time system can catch the point where the programmer adds a bad conversion definition by verifying each new conversion edge as it is created.

Efficient commutativity checking is not trivial. [Let's flip these next two thoughts, because the first thing you have to xxx

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

deal with is resolving cycles—then you can worry about an efficient algorithm for acyclic graphs. —AS] Naïvely checking if all path pairs that begin and end at the same node in a given diagram commute could require a number of function equality checks that grow as factorial in the number of nodes, because a path consists of an ordering of nodes. Further, the presence of cycles implies a potentially infinite number of paths. Previous work [5] has identified an $O(|E|^2|V|^4)$ algorithm to verify that a complete acyclic diagram commutes; however, it addresses neither online addition nor cyclic diagrams.

xxx [Let's stick to present tense, even if it sounds a little weird. And we can state up front what the paper accomplishes. Something like this: "This paper presents an algorithm for online checking of diagram commutativity and applies it to verifying systems of value conversions. We identify two key insights:" —AS] In the course of efficiently verifying commutativity over online addition, we identified two key insights. The first was that in a commutative diagram, when a new edge is to be added, at most one path needs to be checked against xxx [can we rephrase this passive voice, which makes the sentence somewhat tricky to process? —AS] for a given source and sink pair. Because the diagram commutes, all the paths between a given source and sink are equal and a representative to check against can arbitrarily be chosen. It lead to an $O(|V|^2(|E|+|V|))$ algorithm to verify a diagram remains commutative over the course of online addition, assuming an oracle to check the equality of functions. The algorithm makes an asymptotically optimal number of calls to the oracle.

xxx [This paragraph is a little confusing to me. First, what is "informativeness"? Maybe we can be a little more specific. Second, I thought the previous paragraph said we already had an optimal number of calls to the oracle, but the second half of this paragraphs says that we need an even more expensive algorithm to obtain that minimum. —AS] The second insight was that there is a single rule that places a partial, transitive ordering on the informativeness of paths. It allowed for the creation of a greedy, $O(|V|^4)$ optimization step that results in the number of calls to the oracle being minimal. The optimization is very useful when the equality checking oracle is expensive.

We evaluate our algorithms against random graphs and use them in two case studies. xxx [Let's avoid passive voice in these next two sentences by saying "we use the algorithm to do this" instead of "the algorithm is used to do this." —AS] First, our algorithm is used in the domain specific geometry type language *Gator* [2] to ensure that user defined transformations between spaces stay consistent. Second, our algorithm is used to identify inefficiencies in a currency conversion graph.

We empirically compare our solution to three baseline implementations: a naïve checking of all path pairs with only special handling for cycles, a check for all path pairs that involve the new edge, and an algorithm suggested by previous work to solve the batch version of the problem for acyclic diagrams. Our proposed algorithms run orders of magnitude faster than the baselines.

2 Formal Problem Setup and Terminology

We start by formalizing the notion of a diagram, drawing terminology from the previous acyclic work by Murota [5].

Notation. We start with a directed graph $G = (V, E)$, where V corresponds to sets of elements and edges (u, v) in E correspond to functions that maps elements of u to elements in v. All these functions form a semigroup F, where multiplication is function composition. A semigroup consists of a set and an associative binary operation, which we use to capture function composition. The correspondence between edges and functions is stored as a mapping $f : E \longrightarrow F$, where f maps each edge to the function it represents.

A path is a sequence of edges. The edge-to-function mapping f can be naturally extended to paths: if path $p = e_1 \circ e_2 \circ \cdots \circ e_n$ then $f(p) = f(e_1) \circ f(e_2) \circ \cdots \circ f(e_n)$. We write $\partial(p)^+$ for p's start node, $\partial(p)^-$ for its end node, and $\partial(p)$ to denote the pair $(\partial(p)^+, \partial(p)^-)$. [Simplified this explanation; I hope it's OK. —AS] xxx

A pair of paths p and q is called *parallel* iff their terminal nodes are the same, i.e., $\partial(p) = \partial(q)$. ∂, ∂^+ and ∂^- are extended to apply to parallel pairs. [Let's use a name other than p, which we have already used for paths, for pairs. (Or use something else for paths above.) Also, since we just used p and q for two paths, maybe that would be better than p_1 and p_2. —AS] xxx For parallel pair $p = (p_1, p_2)$, $\partial(p) = \partial(p_1) = \partial(p_2) = (\partial(p)^+, \partial(p)^-)$.

Let R_{all} be the set of all parallel pairs of paths in a given diagram. The diagram commutes iff $\forall (p, q) \in R_{all}, f(p) = f(q)$; that is, the composition of maps along any path connecting any pair u to v is independent of path choice.

Problems. The ONLINE ADDITION PROBLEM, given a commuting diagram and a new edge, returns whether the diagram commutes. Checking function equality is a domain specific, potentially hard problem, dependent on the nature of the graph. For our case study in graphics programming (see Section 5.1), edges are matrices and nodes are vector spaces, so function composition uses matrix multiplication and equivalence checking simply compares matrix values. We therefore assume some oracle for checking transformation function equivalence that will vary by domain. We therefore reduce the ONLINE ADDITION PROBLEM to the VERIFICATION SET PROBLEM. [Although it's not technically a "reduction," is it? Maybe better to say that the latter is what we solve, and we assume an oracle that uses those results to solve the former. —AS] xxx The latter, when given a diagram and a new edge, returns the set of parallel pairs of paths, such that if and only if the members in each pair are equal, then the new graph must commute. [Let's be precise around the "equal" terminology in the previous and next sentences. It's not that the pairs are equal—it's that we check function equivalence for the pairs' corresponding functions. —AS] The output to the Online addition problem can then be obtained as whether equality checking for all pairs succeeds.

The algorithms in the rest of the paper assume that the function equivalence oracle is reflexive, symmetric, and transitive. [The following is a little tricky to understand here... maybe this is best left to the currency section? —AS] For example, the oracle for currency graph (Section 5.2) only approximately preserves transitivity because of floating point error magnification, so while the algorithm catches mistakes, it does not guarantee that the diagram commutes.

3 Baseline Algorithms

To examine the efficacy of our proposed solution to the Verification set problem, we compare it to some potential alternatives. Specifically, we examine a naïve factorial algorithm, a slightly less naïve factorial algorithm which we identify to be a two-flip tolerant path search, and Murota's historical batch solution [5].

3.1 Naïve Baseline Algorithm

[Start with an overview of the steps. Something like: deal with cycles, then deal with the remaining paths in an acyclic graph. Maybe also give overall context to say that this algorithm is going to be exponential but at least it will be finite, unlike trying to check "all paths." Then break up the discussion so there is one paragraph per step instead of one long paragraph. —AS]

We start with the set of all parallel pairs in the diagram (with the new edge added in) [clarify: is this a batch or incremental algorithm? If it's a batch algorithm, then there is no need to discuss a "new edge" —AS], and pare it down to be finite by handling cycles. Then, using a procedure like Johnson's algorithm [3], we find all simple cycles in the diagram. [I'm a tad confused here because this sentence says "then..." as if we had moved on from the previous step, but the previous step says it handled cycles already? —AS] We create a cycle verification set, C, and verify for each cycle that a single traversal is equal to the identity function by adding $(v \longrightarrow v, 1)$ for each node v in the cycle to C. [It seems important to have already explained the "type" of C (i.e., the output format of the algorithm? —AS] Here, $v \longrightarrow v$ is a simple cycle starting and ending at v, 1 is the identity function, and these must be verified to be equal to each other. We then create a set P of all the paths in the diagram with no cycles, and filter the set $P \times P$, excluding pairs where the paths begin or end on different nodes, or are identical, to get the set of all cycle-free parallel pairs Q. The output of the algorithm is $C \cup Q$, the cycle verification pairs and acyclic parallel pairs. After verifying C, it is sufficient to verify only Q (Lemma A.1) because cycles must now be the identity, so for any pair in the set of all parallel paths, any instance of a cycle can be removed to obtain an equivalent pair with shorter, cycle-free paths. [What does it mean to "verify only Q"? This could be a bit more precise... —AS] If the shorter pair has equal paths then the paths in the original pair must also be equal to each other. It is therefore safe to remove all pairs of paths with cycles, leaving only parallel pairs where neither path has a cycle. P is finite, bounded by $2^{|V|}$, as a path without cycles is an ordering on nodes, each node occurring at most once. $|P \times P|$, and consequently, $|Q|$, are also finite, bounded by $2^{2|V|}$. Thus the algorithm terminates and returns a finite (if large) set.

3.2 Baseline Incremental Algorithm

Our second baseline refines the output of the naïve baseline. [But even before saying that, let's give some overall context: remind the reader what the incremental setup looks like. Then we can explain that we are going to use something like the previous section. Perhaps we can also be a little more precise than "refining" the output, which could mean many things. —AS] Like before, it start by creating a cycle verification set C', but includes only the simple cycles that pass through the new edge. Then, instead of Q, the set of all non-cyclic parallel pairs, the algorithm obtains its subset Q' consisting of all non-cyclic parallel pairs such that exactly one path in each pair passes through the new edge. To this end, the algorithm performs a *two-flip tolerant path search* whose output is passed into a *path extraction algorithm.* [Maybe we can give a one-phrase explanation of what TFTPS does: it finds all the parallel pairs of which only one path includes the new edge, right? —AS] The result of the path extraction algorithm to get the final output $Q' \cup C'$.

This narrowing can be done because the original diagram commutes. Pairs where both paths do not involve the new edge would remain equal (this would apply to cycles too; cycles that do not pass through the new edge must be the identity). Also, pairs where both paths involved the new edge would have to be equal. To see why this is true, each path could be thought of as consisting of the composition of three segments. For path pair p, and new edge from node S to node T, the first segment extends from $\partial(p)^+$ to the S, the second, the new edge (S, T) itself, and the third, from T to $\partial(p)^-$. The new edge could only appear once because cycles have already been dealt with so only pairs where the path includes the new edge once need be checked (Lemma A.1). The first segment of both pairs would have to be equal because they existed as parallel pairs in the original diagram, and similarly the third segments would also have to be equal. The second segments, consisting of the same edge, would also have to be equal because the equivalence oracle is reflexive. Therefore, the two paths of the pair, a composition of these three equal components, would be equal, since the oracle would preserve

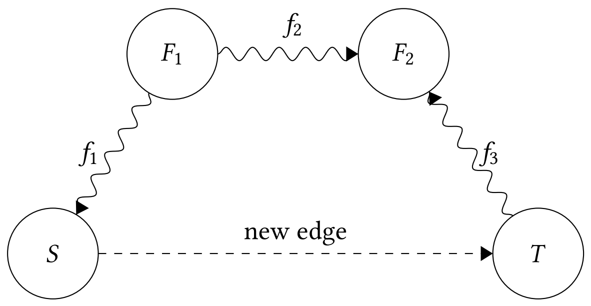


Figure 2. Two flip tolerant path.

transitivity of equality. We are left only with parallel pairs where exactly one of the paths passes through the new edge.

xxx [Maybe a quick paragraph here to remind the reader what is still coming in the rest of this section? —AS]

Two flip tolerant path search. We use a "two-flip tolerant" path search from the source (S) to the sink (T) of the new edge to identify the pairs of paths where exactly one path includes the new edge. The idea is that every such pair corresponds to a two flip tolerant path from S to T. xxx [However, "two flip tolerant path" has not yet been defined, so this remark sounds a little strange. —AS]

In a normal directed graph path search, only forward edges, i.e., edges that go outward from the current node while executing the search are considered. A *two flip path* consists of up to three phases: in the first phase, only backward edges—pointing inward to the source of the search—are accepted. In the second phase, only forward edges are accepted, and in the third phase, again only backward edges are accepted. For a two flip tolerant path p, let $t_1(p)$ map to the first phase, $t_2(p)$, to the second, and $t_3(p)$, to the third. The node between the first two phases we refer to as the *first flipping point*, which has both edges pointing outward; similarly, we refer to the node between the latter two phases as the *second flipping point*, at which both edges point inwards.

We present the idea diagrammatically in Figure 2. Squiggly arrows represent path phases (these are the composition of zero or more edges, not a single edge). The new edge is represented with a dashed arrow. Here, $f_1 \circ f_2 \circ f_3$ is a two flip path, and $f_1 \circ (S,T) \circ f_2$ xxx [should that be f_2 at the end? —AS] is a new path created because of the addition of (S,T) that forms a parallel pair with (f_2).

The two flip tolerant path search returns the set of all paths between a given source and sink that have up to two flips (paths that omit one or more of the three phases are also accepted).

Path extraction algorithm. Next, the *path extraction algorithm* then transforms the output of the two flip path search to the verification set, $Q' \cup C'$. Given a set of two flip tolerant paths from the new edge source to sink, the algorithm outputs a set of pairs to verify.

Let the new edge added to the diagram be (S, T) and the input set of paths, P. The algorithm processes every two flip tolerant path p in P case-wise to obtain pairs to add to the output set. xxx [Bullet points (itemize) are often a good way to lay out cases? —AS] In the case where p has two flips, $t_1(p) \circ (S,T) \circ t_3(p)$ and $t_2(p)$ form a parallel pair. When p has only the first flip (which is to say, the third phase of the path is missing), the parallel paths are $t_1(p) \circ (S,T)$ and $t_2(p)$. Similarly when only the second flipping point is present (so that there is no first phase), then the parallel pair is $(S,T) \circ t_3(p)$ and $t_2(p)$. Finally when no flipping points are present, there are two possibilities. Either p is a path from S to T, in which case the parallel paths are simply the edge (S,T) and p, or p is a path from T to S. In this case, we have found a cycle, $p \circ (S,T)$, to be paired with the identity function. Like with the naïve algorithm, for every node v in the cycle, we add the pair $(v \to v, 1)$, where $v \to v$ is the cycle $p \circ (S,T)$ written to start and end at v.

xxx [A little signposting here might be helpful... maybe a new \paragraph and a brief reminder to the reader about why we need a theorem to tie things up? —AS]

Theorem 3.1. *Perform the two-flip tolerant path search from the source to sink node of the edge that is to be added followed, and on the output, apply the path extraction algorithm. The result is the set $O = Q' \cup C'$ of new parallel pairs with exactly one path passing through the new edge and neither paths containing any cycles, and the set of simple cycles passing through the new edge.*

Proof. Every element in the output of the path extraction algorithm was by construction an element of O.

It is also clear that every cycle in C' can be expressed as $(S,T) \circ p$, and corresponds to the input two flip tolerant path p.

It remains to show that every new parallel pair p in Q' corresponds to a two flip tolerant path. Let $\partial(p)^+ = F_1$ and $\partial(p)^- = F_2$. Only one path passes through (S,T). Let it be called p_1, and the other path, p_2. The two flip tolerant path from S to T can be constructed as follows: phase 1 is the segment of p_1 from F_1 to S, phase 2 is p_2, and phase 3 is the segment of p_1 from T to F_2. Effectively, F_1 corresponds to the first flipping point, and F_2, to the second. It is possible that some of F_1, F_2, S and T coincide (e.g., p starts at S, i.e., $F_1 = S$), in which case the corresponding segments between the coinciding nodes can be considered the identity; the resultant path simply has fewer than two flips. □

Analysis. An upper bound on the number of pairs that this algorithm returns is $O(|V|^2 2^{|V|})$, since two flip tolerant paths are an ordering on nodes, each node appearing at most once, followed by a selection of the flip points. In practice, the algorithm significantly outperforms the naïve batch baseline because it looks only at parallel pairs that involve the new edge, which is usually a small subset of all parallel pairs. Empirical results are presented in Section 6.

3.3 Optimal Batch Solution

Murota's main result [5] is a solution to a batch version of VERIFICATION SET; given an acyclic diagram, it returns the minimal set of equality checks that would succeed if and only if the diagram commutes. The paper describes an algorithm to find the ($|V|^2|E|$ bounded) minimal set of pairs that needs to be checked.

The approach in this algorithm, at a high level, is to define a function that takes in a subset of pairs and returns the subset of pairs whose equivalence is implied by the equivalence of the pairs in the input set. Then the algorithm greedily eliminates redundancies until a minimal set is reached.

A bilinking is defined to be a parallel pair that is disjoint but for their terminal nodes. The set of all bilinkings is R_0. In an acyclic diagram, if all bilinkings are equal, all parallel pairs must also be equal since they any given pair can be expressed as a composition of bilinkings.

Define $r_1 > r_2$ for bilinkings $r_1 = \{p_1, q_1\}$, $r_2 = \{p_2, q_2\} \in R_0$, if there exists a path p such that $\partial(p) = \partial(r_1)$ and p contains p_2. Define $\langle\rangle$ as: $\langle r\rangle = \{s \in R_0 | r > s\}$.

For bilinking s, let $F(s)$ be the vector in $GF(2)^{|E|}$ representing the edges present in s (the n^{th} dimension of $F(s)$ is 1 if the corresponding edge is in s, and 0 otherwise). xxx [GF(2) or $\mathbb{F}_2$ is standard notation for the Galois field of two elements. Do I need to write more about it here? —AK] xxx [Probably just to briefly tell the reader what it is, not to fully explain it. —AS] Let this function be extended to sets, so that for some set of bilinkings S, $F(S) = \{F(s) | s \in S\}$. A notion of linear independence in this vector field exists.

For a set of bilinkings R, the closure function cl is defined as: $cl(R) = \{s \in R_0 | s \text{ is linearly dependent on } F(R)\}$. The closure function on R basically captures all the pairs that can be made by made by composing or "gluing together" the bilinkings in R. Using these two functions, we finally define the function σ on a set of bilinkings R as $\sigma(R) = \{s \in R_0 | s \in cl(R \cap \langle s\rangle)\}$. This is the function used to capture all the pairs whose equivalence is implied by the equivalence of pairs in R. σ is used xxx [passive voice, and a somewhat awkward start to the sentence —AS] to iteratively check if a given pair is redundant. Bilinkings are eliminated until a minimum "spanning" subset is reached.

Roughly, the algorithm proceeds by first efficiently finding a *spanning* set of bilinkings (a subset whose verification implies the verification of all bilinkings in the graph). It does this by, starting at every node, finding the reachable subsection of the graph, and a spanning tree for the subsection. From each edges in the reachable section that is not a part of the tree, it generates a bilinking using the edge and a path in the tree that is parallel to the edge (Algorithm 1).

xxx [Probably put this in the appendix —DG]

```
Result: Find a spanning set Rs = [$r_1$, ... ,$r_k$].
Graph existingGraph
$R_s \leftarrow \{\}$
foreach node v in V do
    S ← existingGraph.extractReachableSection(v)
    T ← createMinimumSpanningTree(S)
    excludedEdges = S.edges - T.edges
    foreach edge e ∈ excludedEdges do
        firstPath = T.findPath(source: e.source, sink:
          e.sink)
        $R_s$.addElement(new Bilinking(firstPath, e)
    end
end
return $R_s$
```

Algorithm 1: finding spanning set

With the spanning set thus initialized, it greedily tries to remove each pair from the spanning set if the set remains spanning even after removing the edge (Algorithm 2).

```
Result: Find a minimal spanning set R.
Function σ(inputSet S, codomain Rs):
    output ← {}
    for bilinking ∈ Rs do
        /* Get fragments that could build up
           to the bilinking.                    */
        smallerPairs ← getSmallerPairs(bilinking)
        consideredPieces ← smallerPairs ∩ S
        /* See if bilinking can be built from
           these pieces.                        */
        if linearlyDependent(consideredPieces,
          bilinking) then
            output.add(bilinking)
        end
    end
    return output
R ← Rs
for i=1 to K do
    if $r_i$ ∈ σ(R-$r_i$) then
        R ← R-$r_i$
    end
end
return R
```

Algorithm 2: Minimal spanning set

The proof of correctness can be found in Murota[5]. The number of checks returned by the algorithm is at worst $O(|V|^2|E|)$. The overall run time of an optimized implementation is $O(|V|^4|E|^2)$.

4 Solving the Online Addition Problem

We present a polynomial time solution to the VERIFICATION SET problem. [Elsewhere, "problem" is also in small-caps. Either way is fine, but consistency is important. —AS] Like in the online baseline algorithm, we do not concern ourselves with parallel pairs where neither or both paths pass through the new edge. The key observation that allows us to improve on the online baseline is that, for a given source and sink pair, only a single parallel pair needs to be verified. This is an implication of Theorem 4.1, expanded on later. Theorem A.2 shows that should our selected set of pairs along with cycles passing through the new edge be verified commutative, the entire diagram must commute. The approach is to identify a parallel pair with exactly one path through the edge for each (source, sink) pair (Algorithm 3). [Instead of putting the reference in parens, say "Algorithm X is the strategy for doing Y. Some more details include Z." —AS]

The try block is executed at most $O(|V|^2)$ times, so that this is the bound on the number of pairs verified. The bound is asymptotically tight. This can be seen in the case where the graph contains $2N$ nodes besides S and T. [Introduce this concept by saying that "we imagine dividing the nodes into two groups..." —AS] We consider N of the nodes to be in group 1 and the other N to be in group 2. Every node in group 1 has a forward node to every node in group 2 as well as to S. T has a forward edge to every node in group 2. In this diagram, on the addition of edge (S, T), N^2 paths need to be verified which is polynomial in the total number of nodes, $2N + 2$.

Also notice that if trying to optimize for path length (say, if composing functions is expensive) then "find any path" can be replaced with "find the shortest path."

An efficient implementation of the algorithm can run in $O(|V|^2(|V|+|E|))$ time, with space complexity not exceeding the asymptotic $O(|V|^2)$ bound on the output. In such an implementation, path finding from a given source node to all potential sink nodes could be done in a single $O(|V| + |E|)$ breadth first search.

4.1 Optimization Step

In the case where equality checks are very expensive, we begin by finding the minimal set of (source, sink) pairs such that checking for these pairs logically implies having checked the full diagram.

We observe that there are some redundancies in the diagram. Consider the situation in Figure 3. [This needs a bit more explanation. What is the reader looking for? Also, the caption of this figure could probably be a little clearer. —AS]

Theorem 4.1. *If parallel paths $g_2 = f_2; (S, T); h_2$ then it must be that $g_1 = f_1; (S, T); h_1$.*

Proof. We use the fact that f_1=l; f_2 and h_1=h_2; m.

$$g_2 = f_2; (S, T); h_2 \Rightarrow l; g_2 = l; f_2; (S, T); h_2$$

```
Data: existing graph, new edge.
Result: Set of parallel pairs to verify.
Graph existingGraph; Edge newEdge;
parallelPairs ← {}
for S in existingGraph.Nodes do
    for T in existingGraph do
        try:
            Path pathWithNewEdge ← FindPath(
              graph: existingGraph, sourceNode: S,
              sinkNode: newEdge.Source()) +
            newEdge as Path +
            FindPath( graph: existingGraph,
              sourceNode: newEdge.Sink(), sinkNode:
              T)
            if S == T then
                pathInOldGraph ← identity(S)
            end
            else
                Path pathInOldGraph ← FindPath(
                  graph: existingGraph, sourceNode: S,
                  sinkNode: T)
            end
            parallelPairs.add((pathInOldGraph,
              pathWithNewEdge))
        catch PathFindingFailedException:
            /* No comparable pairs from node S
               to node T that need to be
               checked                          */
            continue
        end
    end
end
return parallelPairs
```
Algorithm 3: Online polynomial time algorithm to find parallel pair set

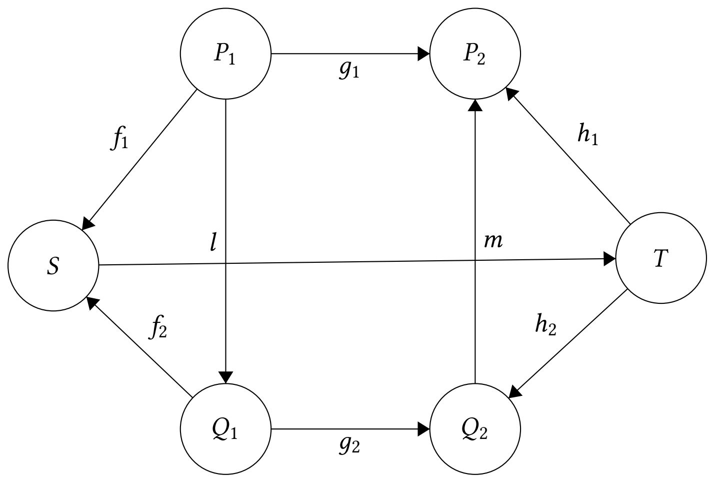
$$\Rightarrow l; g_2; m = l; f_2; (S, T); h_2; m \Rightarrow g_1 = f_1; (S, T); h_1$$

□

The proof is not affected if any of these [which? —AS] paths is the identity, e.g., if f_1 is the identity and S and P_1 are actually the same node. [Maybe this observation belongs inside the proof block, where it's most relevant? —AS]

We conclude that verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of all path pairs (Q_1, Q_2) such that Q_1 is a successor of P_1 and P_2 is a successor of Q_2. A successor S to node N is any node such that there exists a path from N to S. Nodes are also their own successors and predecessors. [At some point, we have switched to capital letters for both paths and node variables... maybe make this consistent? —AS]

[I couldn't quite follow the following, perhaps because the figure is not quite enough for me to understand what the



Each arrow represents a path, and (S, T) is the new edge being added.

Figure 3. Reduction rule.

"rule" is here. Which makes it hard to see why "operations" are relevant in this context. —AS] Under the assumption that the only path operations allowed are composition and replacement of one path by a different, equal path, as would be true when edges are generic functions, and no other information is available, so that F is a semi-group, this "reduction" rule is also the only rule to reduce the set of path pairs to check by finding implications.

That is to say, if verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of a pair with endpoints (Q_1, Q_2), then it must be that Q_1 is a successor of P_1 and P_2 is a successor of Q_2.

Using this information it is possible to choose a minimal subset of path pairs to verify, as in Algorithm 4. We construct a graph with a node for each possible (source, sink) pair in the graph: each node then represents a possible choice for parallel pair endpoint pairs, and we greedily search for the smallest set of nodes from which the entire graph would be reachable. The idea is to look for "roots" in the graph that have to be included in the ultimate verification set because they have no predecessor an cannot be verified "through" the verification of some other pair. Then all the successors whose verification is implied by the roots are eliminated.

The result of the algorithm cannot be further reduced. [Do we need an argument for why? —AS] Also, the verification of the parallel pairs returned in the algorithm implies that the output of the previous algorithm must commute, and transitively with Theorem A.2 that the entire diagram must commute.

The run time of the first step is $O(|V|^4)$, and that of the second step is $O(|V|)$, so that the overall bound is $O(|V|^4)$. Space complexity remains $O(|V|^2)$.

```
**Data:** Existing graph, new edge.
**Result:** Set of parallel pairs to verify.
Graph existingGraph
Edge (S, T)
predecessors ← S.predecessors(existingGraph)
successors ← T.successors(existingGraph)
Graph terminalPairGraph
for q ∈ successors do
    for p ∈ predecessors do
        terminalPairGraph.addNode(q, p)
        for qPred ∈ q.predecessors(existingGraph) do
            for pSucc ∈ p.successors(existingGraph) do
                terminalPairGraph.addEdge((qPred,
                  pSucc))
            end
        end
    end
end
verificationSet ← {}
while len(terminalPairGraph.nodes) > 0 do
    currentNode ← terminalPairGraph.nodes[0]
    visitedNodes ← {}
    while len(currentNode.parents()) > 0 do
        visitedNodes.add(currentNode)
        currentNode ← currentNode.parents()[0]
        if currentNode ∈ visitedNodes then
            edges ← getAllEdges(visitedNodes,
              terminalPairGraph)
            terminalPairGraph.removeNodes(visitedNodes)

            terminalPairGraph.addNode(currentNode,
              edges)
        end
    end
    verificationSet.add(currentNode)
    terminalPairGraph.removeNodes(
      currentNode.successors(terminalPairGraph))
end
return verificationSet
```

Algorithm 4: Minimal set finding algorithm

4.2 Verification

Ultimately verification is performed by calling an equality oracle (that is beyond the scope of this work) for every comparable pair returned by the previous stage, as in Algorithm 5.

5 Applications

[Just a brief overview here to explain to the reader why these applications are important and appropriate (and what we hope to learn from the case studies). —AS]

```
for (path1, path2) ∈ parallelPairs do
    if path1 != path2 then
        return False
    end
end
return True
```

Algorithm 5: Verification algorithm

To demonstrate our algorithms applied to a real world situation, we search for inconsistencies in diagrams of geometry transformations, and in a diagram of the exchange rate between currencies.

5.1 Gator

Gator is a domain specific language designed around Geometry types, which are used to describe properties and transformations of geometric objects. A key feature of Gator is in expressions, which insert code to automatically transform between two geometry types. For example, given a point p represented in 2-dimensional Cartesian coordinates (which has type cart2), we can transform this point into polar coordinates using the expression p in polar.

A key property of these in expressions is that the path chosen between two representations is arbitrary; thus we would like to ensure that every path is equivalent. This problem defines a natural application space for working with our commutative diagrams, so long as we can define a transformation graph and oracle.

For simplicity, and to stay in scope for this paper, we focus on geometry transformations between *reference frames*, which are the geometry equivalent of transforming between linear algebra basis vectors. Each edge on our transformation graph is a thus a matrix, with composition of edges as matrix multiplication and an oracle checking matrix equality (up to a rounding error ϵ).

These sorts of transformations are very common in graphics programming, and there are several examples of reasonably complicated transformation graphs that we can pick from. Gator includes several graphics examples as part of its examples package; for this evaluation, we looked at the *phong*, *reflection*, and *shadow map* examples, sample images of which can be seen in Appendix B.

We implemented a system for interfacing with the optimal set incremental checker in Gator xxx [Which algorithm was it? —DG] xxx [Algorithm 4 —AK]. Gator uses TypeScript for CPU computations, so it was reasonable to build a system to manage graphs for interoperation with the path independence algorithm.

Using this algorithm, we found a bug in the reflection model, where a matrix was not being divided correctly by its determinant xxx [List other things I find, if anything —DG] xxx [I think it may be nice to add more detail for at least one bug (code, what was the program doing, what the graph/conflict was, time taken, etc.) —AK]. We also inserted previous bugs to check the algorithm was working correctly, and it identified all inserted bugs.

5.2 Currency Graph

xxx [To perhaps excuse why this section doesn't have a proper programming language attached, maybe we should briefly say that we are imagining a units-of-measure type system—but to make it more interesting and scale to large graphs easily, we are using currencies as units. —AS]

Consider a diagram with nodes as currencies and a directed edge being the conversion rate from its source node's currency to its sink node's currency. Since the transformation exchange rate of money from any given base currency to a target currency can be expected to be the same regardless of which intermediate currency transformations are used, this diagram should commute.

Using a web API[1] for currency data, we built the fully connected diagram of exchange rates between 32 currencies on a given day. To ensure that it indeed commuted, we started with an empty diagram, and added in edges one by one. Before the addition of each edge, we used the algorithms (Algorithm 3 and Algorithm 4) xxx [remind the reader what these are: our algorithms? baselines? batch? incremental? etc. —AS] to ensure the addition of a new edge did not introduce inconsistencies in the existing diagram. If a new edge was, in fact, problematic, the algorithms returned an example inconsistent pair that would arise from the addition of the edge. The pair would consist of two currency transformation sequences with the same source currency and ultimate destination currency, but with different effective exchange rates values, as computed by taking the product of all the exchange rates encountered through the chain.

We allowed an "error tolerance" so that differences reported would not be the trivial consequences of a floating point error. However, this relaxation of the equality oracle into imprecision meant that the mathematical reasoning that allow the algorithms to remove redundant path checks no longer applied. For instance, composing a new function with two approximately equal functions does not lead to equal results, so Theorem 4.1 fails with this approximate equality. When the algorithms reported no inconsistencies, it was still possible that the graph possessed inconsistencies above the given threshold and did not commute. Nonetheless, both algorithms were effective in catching inconsistencies. Algorithm 3 started finding inconsistencies at error tolerances to the order of 10^{-3}, and Algorithm 4, which makes more invalid redundant path check removals, at error tolerances to the order of 10^{-7}.

Averaging over evaluation for the first 30 days of 2020, building and verifying a diagram to completion (inclusive of the time required by network calls) took 243±19 seconds

[1]https://exchangeratesapi.io

Algorithm	average seconds of computation
Naïve baseline	0.77
Two Flip tolerant	0.075
Batch algorithm	7.55
Algorithm 3	0.0038
Algorithm 4	0.00086

Table 1. Computation time for 9 node graph of density 0.4, averaged over ten runs

Algorithm	average number of output pairs
Naïve baseline	39754.9
Two Flip tolerant	748.9
Batch algorithm	23
Algorithm 3	78.3
Algorithm 4	1

Table 2. Output size for 9 node graph of density 0.4, averaged over ten runs

using Algorithm 4, and in 133±13 seconds with Algorithm 3. xxx [What is the conclusion the reader should take away? —AS]

6 Evaluation

We compare performance of the following path checking algorithms: (1) the naïve baseline, (2) the less naïve two-flip baseline, (3) the batch baseline, (4) Algorithm 3, and (5) Algorithm 4. xxx [Again, the reader probably needs a reminder about what the latter ones are. —AS] The two aspects we look at are time for response and size of response set (smaller sets—tighter output results—would mean less calls to the oracle). We use randomly generated graphs for benchmarking, and vary their size. Given a graph and a new edge, we time how long it takes for an algorithm to return the set of pairs that need to be verified. All computation was performed on a Macbook Pro 2015, 2.9 GHz Dual-Core Intel Core i5.

The naïve baseline performs poorly, taking well over a thousand seconds for even small graphs of 10 nodes. The batch algorithm brings timing down to seconds. Algorithm 3 performs only slightly better. Surprisingly, the optimal set algorithm cuts time cost by several orders of magnitude, and runs in milliseconds for small graphs. All implementations are sensitive to density, performing better when density is very low or very high.

6.1 Comparison of Algorithm Time Cost

The average time taken by each algorithm over the course of 10 runs over randomly generated graphs with 9 nodes and 32 edges is listed in Table 1. xxx [Perhaps the previous paragraph summarizing the findings from this table should be moved here? —AS]

6.2 Scaling of Time with Input Size

xxx [In general, I recommend the syntax "Figure X shows the time results. They are interesting." over "The time results indicate something interesting, as Figure X shows." Putting the reference up front helps the reader immediately see what figure they should be cross-referencing. (Occurs in both paragraphs below.) —AS]

Algorithms scale as expected with time, as seen in Figure 4. xxx [The plots are weirdly tiny on my machine... any chance your plotting tool can output PDF images? —AS] Both Algorithm 4 and Algorithm 2 exhibit graphs that are polynomial in appearance. [Algorithm 2 does not appear in the figure. —AS] xxx The naïve baseline as well as the two flip tolerant baseline display quick growth. The batch algorithm also grows fast, though not as much as the online checking baselines.

We define density to be the ratio of the number of edges in the graph to the total possible number of edges ($|V|^2$, where $|V|$ is the number of nodes). Run time relates to the density of edges in the input graph. The degree of the effect differs with the algorithms, as Figure 4 shows. Generally, denser graphs entail longer computation time. For the batch algorithm we use lower densities since the input graph must be acyclic. This puts an upper bound on density that approaches 0.5 in large graphs.

6.3 Variance

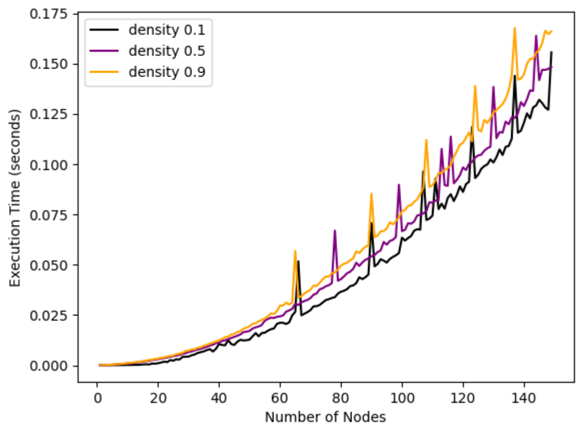
The periodic spikes in Figure 4a are striking. We plot the spread of results in Figure 5a to understand what is happening. Grey points are the results of evaluation on individual points, and error bars show standard deviation. The black curve traces the mean. We find Algorithm 4 has outliers about two standard deviation above the mean responsible for the spikes in the average. The outliers themselves follow a polynomial curve, appearing almost periodically. We have not yet identified the cause of the behavior. Figure 5 depicts the situation for Algorithm 3, where no such effect is observed.

6.4 Size of Output

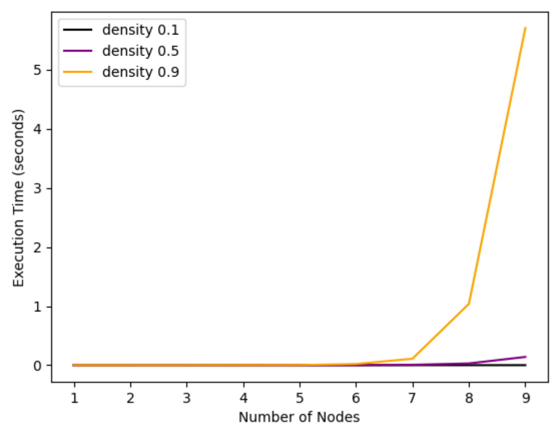
Output size is a metric of interest, should the equality checking oracle be expensive. Table 2, summarizes the number of output pairs that the algorithms returned on average over 10 runs, for graphs with 9 nodes and 32 edges. [Conclusions. xxx Are they as expected? If so, it's OK to just say why they were expected. —AS]

7 Related Work

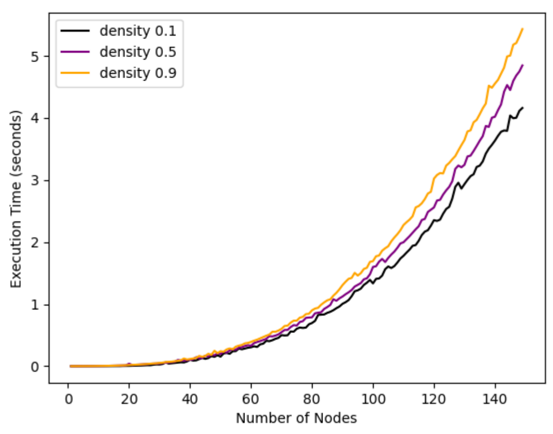
Section 3.3 describes Murota's solution to efficiently finding the minimal set of path pairs that need to be compared to check if a given acyclic graph commutes [5]. We did not find any other work that solves the question of verifying that diagrams commute.



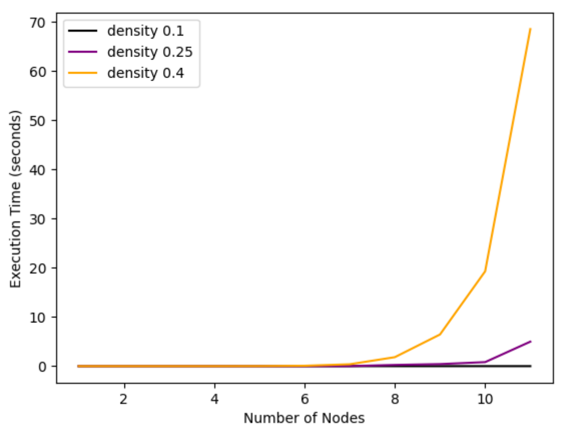
(a) Algorithm 4



(d) Two flip tolerant baseline

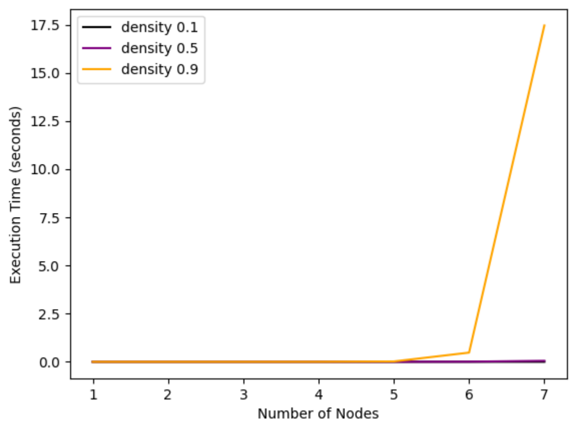


(b) Algorithm 3



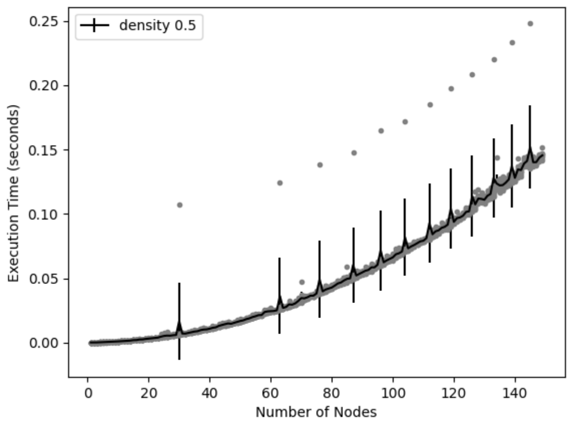
(e) Batch algorithm baseline

Figure 4. Plots of scaling of time with input size and density

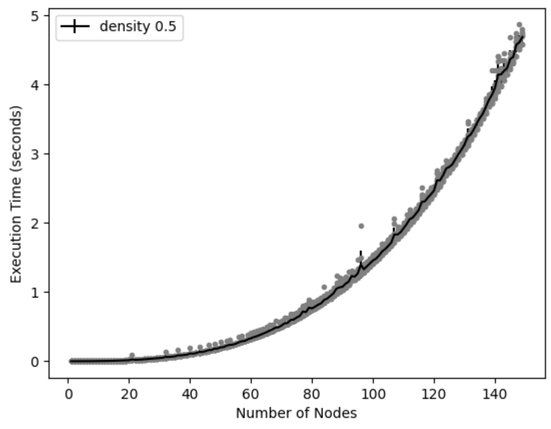


(c) Naive baseline

However, the question of commuting does come up in programming languages with implicit type conversion. Gator [2], as described in Section 5.1, supports automatic type conversion between geometry types. The language implements some restrictions to eliminate obvious cases of non-commuting graphs, but does not verify that defined graphs commute, allowing scope for non-commuting graph definitions. Frink [1] is a language that supports automatic conversion between units and infinite precision floating point numbers. It does not appear to support the implicit definition of conversion between unites [Need to spend some time confirming this —AK] xxx but if extended to do so, would need to contend with the problem of commuting graphs. The same is true for F# which has support for units of measure [4] and Ada's GNAT compiler [6].



(a) Algorithm 4



(b) Algorithm 3

Figure 5. Plots of spread of results

8 Conclusion

Being able to verify if diagrams commute can be a very useful program analysis tool. It allows a compiler to make deterministic automatic type conversions and can catch inconsistencies of definition in a program with user defined conversions. In this paper, we have presented verification algorithms that given a commuting diagram, a new edge, efficiently compute the set of paths that would equal to each other if and only if the diagram would still commute after addition of the edge.

Integrating conversion consistency checks into widely used languages such as scala could provide a lot of value to the program. Beyond implementation work, this requires an answer to the question of how to write function equality oracles.

In the evaluation section, we found that the graph for the optimal set path checker displayed a peculiar regularity in outliers. Future work might include investigating this behavior.

References

[1] Frink. http://frinklang.org/#Features. Accessed: 2020-08-10.
[2] Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. Geometry types for graphics programming. In *OOPSLA*, 2020.
[3] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4:77–84, 1975.
[4] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, CEFP'09, page 268–305, Berlin, Heidelberg, 2009. Springer-Verlag.
[5] Kazuo Murota. Homotopy base of an acyclic graph—a combinatorial analysis of commutative diagrams by means of preordered matroid. *Discrete Appl. Math.*, 17(1-2):135–155, May 1987.
[6] Edmond Schonberg and Vincent Pucci. Implementation of a simple dimensionality checking system in ada 2012. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, HILT '12, page 35–42, New York, NY, USA, 2012. Association for Computing Machinery.

A Theorems and Additional Cases

Consider all the cycles in the graph that pass through the new edge, unique to their terminal point. Define a cycle-pair corresponding to cycle c to mean the pair of paths (c, identity on terminal node of c). Let C be the set of all cycle-pairs (unique to their terminal point) corresponding to cycles that pass through the new edge.

Let V, the set of pairs to verify, be $C \cup R_0$.

Lemma A.1. *A path that involves the new edge more than once is equal to a path without multiple occurrences of the edge, under the assumption that the pairs in C are verified.*

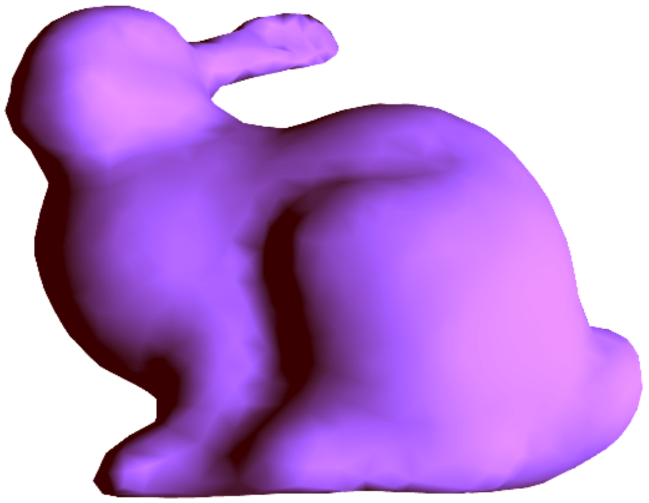
Proof. Consider the part of the path between the first occurrence of the new edge and the second. This forms a cycle, which has a corresponding check in C and must be verified to be equal to the identity. This means that the loops containing the multiple occurrences of the new edge may be ignored and the path with a single occurrence of the new edge with the cycles removed is equal to this path. □

Theorem A.2. *Verifying that all pairs in V commute implies that all pairs in R_{all} commute.*

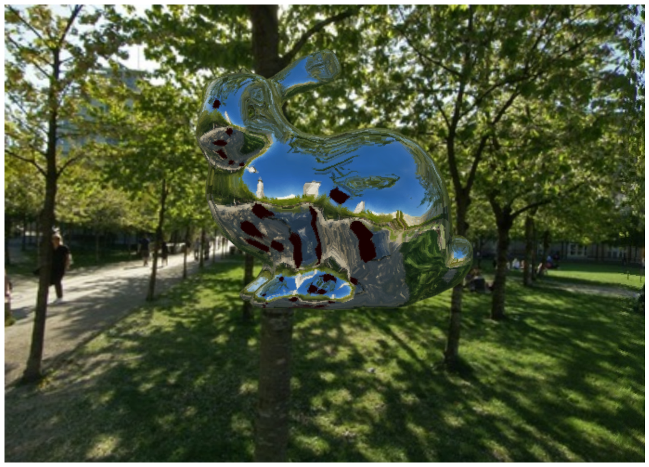
Proof. Each pair in R_{all} that is not in V would fall into one of these categories:

Case A: pairs that do not involve the new edge: By assumption that the original diagram commutes, these pairs must commute.

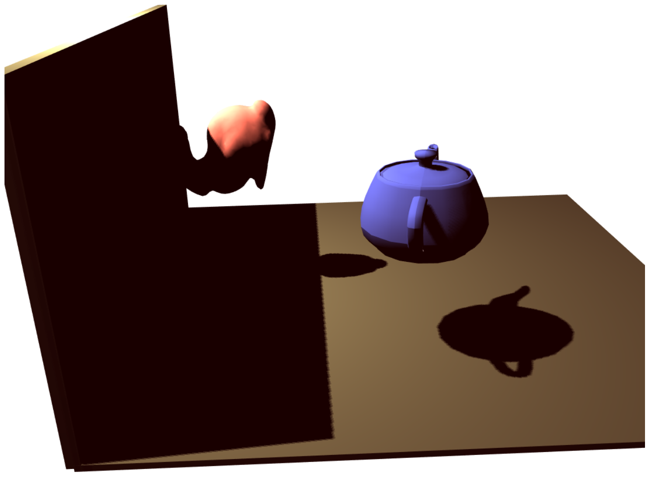
Case B: Cases where there is a different pair corresponding to this choice of (source, sink) in V already. There are four paths to be considered, two from each pair. Those paths which do not include the new edge must already be equal by the assumption that the original diagram commutes. Those paths which do include the new edge (S,T) can be reduced, by the case for multiple occurrences of a new edge, to a path



(a) Phong Lighting Model



(b) Reflection Model



(c) Shadow Map Model

Figure 6. Example Images from Gator Programs

with a single occurrence of the new edge as described in the preceding lemma A.1. These new paths can be divided into three segments: the segment from the source to S, the new edge, and the segment from the last occurrence of T to the sink. The first segments of all paths are equal by assumption that the original diagram commutes. This is also true of the third segment of all the paths. The second segment consists of only the new edge and is the same for all paths. The composition of equal functions is equal, so all the paths passing through the new edge must be equal. Therefore it is sufficient to check one path that passes through the new edge and one that doesn't. Such a pair belongs to V by construction. □

B Example Gator Images