DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java)

Abstract
Modern languages have constricted many classes of programming errors, such as memory and type errors. As a result, algorithmic and semantic errors now present a proportionally greater challenge during the development cycle. Many static tools exist to track down such errors, but data structure bugs, an important class of semantic bugs, resist these tools because scalable heap analysis is very challenging. Dynamic solutions do not fare much better, as user-written or automatically generated data structure invariant checks are often prohibitively slow, commonly incurring at least a ten-fold slowdown.

We present DITTO, an automatic incrementalizer for read-only data structure invariant checks in modern imperative languages like Java and C#. Such checks include verifying the properties of a red-black tree or that the elements in each bucket of a hash have the right hash code. Incrementalization speeds up function execution by using previous execution results and only performing anew the portions of computation that correspond to new input. DITTO exploits properties specific to invariant checks to automate and simplify the process without constraining the kinds of mutations that can be performed.

The source-to-source implementation of DITTO, for Java, is automatic, portable, and efficient, providing speedups on data structures with as few as 100 elements. On larger data structures, it provides the asymptotic speedups characteristic of incrementalizers, roughly 5x at n=5000 and growing linearly with n thereafter.

1. Introduction
Modern imperative languages such as Java and C# contain a number of powerful features: automatic memory management, a strong type system, and dynamic type, pointer, and bounds checks. These features reduce or eliminate many types of programming errors, such as buffer overflows and doubly-freed memory. As a result, algorithmic and semantic errors present a proportionately greater challenge during the development cycle.

One such class are data structure bugs. Many data structures respect high-level invariants not expressible in the base language, such as “the elements of this list are ordered”, “no elements in this priority queue can be in that priority queue”, or “in a red-black tree, the number of black nodes on any path from the root node to a leaf is the same”. Because the heap is persistent over the course of an execution and globally accessible, bugs that violate these invariants may also want to obtain an efficient check rapidly, for example when writing custom checks to serve as data breakpoints for explaining symptoms of a bug. Most importantly, perhaps, the complexity of maintaining incremental code scattered throughout the program may be difficult to incrementalize by hand. For example, after some effort, we gave up on incrementalizing red-black tree invariants.

Furthermore, having to optimize checks manually does not appear economical: programmers may want to develop a large number of small checks and optimizing each seems prohibitive. Programmers may also want to obtain an efficient check rapidly, for example when writing custom checks to serve as data breakpoints for explaining symptoms of a bug. Most importantly, perhaps, the complexity of maintaining incremental code scattered throughout the program may be harder than verifying correctness of the original data structure, which defeats the purpose of relying on invariant checks that are simple and thus correct by inspection.

Recent research by Acar, et al [1] developed a powerful general-purpose framework for incrementalization, based on memoization and change propagation. The framework provides an efficient incrementalization mechanism while offering the programmer considerable flexibility. Working in a functional language setting, the programmer is responsible for (i) identifying locations whose changes should trigger recomputation; and (ii) writing functions that carry out the incremental update on these locations. The actual memoization and recomputation are encapsulated in a library. Acar’s incrementalized algorithms exhibit significant speedup so it is natural to ask how one could automate this style of incrementalization.

In this paper, we identify an interesting domain of computations for which we can develop an automatic incrementalizer. Our domain corresponds to recursive side-effect-free functions, which covers many data structure invariant checks, such as red-black tree in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright (c) ACM [to be supplied] ...$5.00
variants, and common invariants for many other data structures like ordered lists and hash tables.

The properties of this class of invariant checks allow us not only to automate incrementalization but also to offer a very simple implementation. Since the checks are side-effect free, the parts of the computation with the same inputs will return the same results. Checks also commonly establish a global property ("this list is sorted") by aggregating many local results ("this list element is less than the next element"), so small changes often require a small theoretical minimum of recomputation. These two factors can be exploited by an incremental computation to produce giant speedups. Finally, invariant checks often return consistent results from subcomputations, simplifying the incrementalization transformation by allowing for optimistic memoization, a technique described in this paper that aggressively enables local recomputations to reconstruct a global result.

DITTO utilizes these properties to enable incrementalization that

1. Works for imperative languages like Java and C#. Checks are often most useful in these languages because data structure updates can occur anywhere in the program. However, incrementalization is difficult because heap updates can affect dependencies of cached computations or inputs to incrementalized functions at any time.

2. Is automatic. Programmers like simple invariant checks because they can be written quickly (often to track down a particular bug or behavior) and are obviously correct; explicit incrementalization annotations can slow down the process and obfuscate correctness. Furthermore, for better or for worse, optimizations that require annotations rarely gain widespread acceptance. For an important class of read-only checks, DITTO "just works," and we feel that this ease of use is critical to programmer adoption.

Paper contributions and structure. The main contributions of this paper are

1. The DITTO automatic incrementalizer for a class of data structure invariant checks written in a modern object-oriented language.
2. A portable implementation of DITTO in Java.
3. An evaluation of Java DITTO on several benchmarks.

Section 2 describes how a simple invariant check is incrementalized, from a programmer's point of view. Section 3 describes DITTO's incrementalization algorithms and Section 4 provides some implementation details. Section 5 evaluates DITTO on several small and large benchmarks, while Section 6 discusses related work and Section 7 concludes.

2. Example

In this section, we give a high-level overview of DITTO's incrementalization process. First, we define the class of invariant checks that DITTO can incrementalize.

**Definition 1.** The inputs to a function consist of its explicit arguments, the values of its formal parameters, its implicit arguments, values accessed on the heap via formal parameters or static fields, and its callee return values, the results of function calls it makes.

**Definition 2.** A data structure invariant check is a collection of deterministic, recursive side-effect-free functions that return primitive types. In each function, heap loads, function calls, and control flow cannot depend on both a callee return value and another class of input. 1

---

1 This technical restriction, described further in Section 3.4, is required to ensure that the original functions and their incrementalized versions have the same termination properties. We have not found it to be an impediment in practice.

---

Most iterative invariant checks can be written just as cleanly in a recursive fashion, so we do not handle iterative checks separately here. Additionally, throughout this paper, we will often assume a check is specified as a single recursive function, for programmer convenience. However, DITTO supports multiple recursive functions just as well (see Figure 8 for an example); in the case of a collection of functions, the check is identified by the single entrypoint function that is invoked from the rest of the program.

The rest of the program has no restrictions on its behavior. We assume that invariant checks running in multithreaded programs either operate on thread-local data or are atomic to ensure data integrity during the check.

To illustrate how DITTO works, we walk through the incrementalization of a simple invariant check, isOrdered, in an example class OrderedIntList; see Figure 1.

The invariant verifies that the values of the list elements increase monotonically from the beginning of the list to the end. Calls to isOrdered might be inserted automatically by a development tool, or manually by the programmer, at the method boundaries of OrderedIntList. In this case, they are inserted at method entry, to check for modifications done externally (such as if an OrderedIntElem were mistakenly exposed and modified by code elsewhere in the program), and at method exit, to ensure that each list operation maintains the invariant.

It is easy to see from this common usage scenario that an unoptimized isOrdered can dominate the performance of the program.

However, the function is amenable to incrementalization under most common list modifications. For instance, if an element is inserted into the middle of the list, only that new element and its predecessor need to be checked for orderedness; the success of the previous invocation of isOrdered, before the change, guarantees the property for the other elements of the list, which have seen no changes since then. This optimization reduces the cost of the check from linear in the size of the list to constant.

**Incrementalizing isOrdered.** DITTO automatically incrementalizes isOrdered using the following simple process.

1. During the first execution of invariants, remember the series of recursive calls to isOrdered, their inputs, and their results.
2. During the subsequent execution of the program, track all changes to any of these inputs.
3. The next time invariants is invoked, identify the recursive calls to isOrdered whose inputs have changed, and re-execute only those calls, reusing the memoized results from the rest of the calls. Update the memoized results so that further executions can be incrementalized as well.

In this way, only the invocations of isOrdered that deal with modified portions of the underlying OrderedIntList are re-executed.

We describe these behaviors in detail in the context of a modification scenario, in which an element is inserted into the list and another element, further down the list, is deleted. (In this scenario, the invariant check is only run after both modifications, and not in between.) See Figure 2.

In this example, each invocation of isOrdered has one explicit argument, the formal parameter e; three implicit arguments, the fields e.next, e.value, and e.next.value; and one callee return value, that of the recursive call to isOrdered. During the previous execution of invariants, DITTO had stored the particular inputs for each invocation of isOrdered.

The modification shown in Figure 2 updates two fields already in the list: A.next and D.next. Based on the stored inputs from the previous invocation, DITTO identifies the calls isOrdered(A) and isOrdered(D) as using the values stored in these fields as inputs; they must be rerun using the new input values to generate the correct output. Since isOrdered(A) occurred first in the previous execution, it is rerun first here as well.

The re-running of isOrdered(A) uses the new implicit arguments, specifically the new value of A.next, which now points to B. Execution continues to the recursive call, isOrdered(B). Since DITTO has never seen isOrdered with the argument B before, it adds this new call to its memoization table and continues executing, reaching the recursive call to isOrdered(C).

At this point, DITTO identifies a memoized version of isOrdered(C) with the same explicit and implicit arguments, since C has not changed since the previous execution of isOrdered. However, there is no guarantee that the last input, the callee return value from isOrdered(C.next), will return the same value it did originally, since there is a change further down the list; isOrdered(D) might return a different value that ultimately affects the value returned by isOrdered(C.next).

Regardless, DITTO optimistically assumes that the recursive call will return the same value as before, since recursive invariant check calls often do. This optimistic memoization strategy allows DITTO to terminate the recursion immediately, reusing the cached result for isOrdered(C). Execution bubbles back up to isOrdered(A), which returns true — the same value it returned before. Thus, whatever function invoked isOrdered(A) need not be re-run, since all of its inputs are the same as before; execution of this modified portion of the computation stops.

DITTO then re-executes isOrdered(D). Again, execution recurses to isOrdered(P), where DITTO reuses its cached result and ceases recursion. In this case, isOrdered(D) evaluates to true, matching its previous result, so local recomputation ends.

DITTO returns the cached result of the entire invariant check, true. isOrdered(E) is no longer reachable in the computation and is ignored.

isOrdered(D) were instead to return a different result, false, this value would be propagated back up to its caller, which would be rerun, and so on, until either (a) a caller is reached that evaluates to the same result that it did before, or (b) execution reaches the first caller, isOrdered(head), and the new overall result is cached for future usage and returned. It is possible for the upward propagation to reach isOrdered(A), rendering the optimistic memoization decision made there incorrect. In this case, isOrdered(A) is rerun like the other nodes during this propagation phase.

Either way, the new inputs and result for each call that was rerun are stored for reuse during the next execution. The memoization data for isOrdered(E) is garbage collected. All of this maintenance of the execution data ensures that DITTO will be able to incrementalize the invariant check during its next execution as well.

Of course, data structure modifications can take on more complex forms than simple inserts and deletes. The next section describes how all modifications are handled in a general way, and Section 5 examines the performance of DITTO on invariants of considerably greater complexity.

3. Algorithm

DITTO takes as input a program containing some data structure invariant checks, and a list of checks to optimize, and outputs a modified version of the program that, with the help of a runtime support library, incrementally computes the invariant checks. The list of checks to optimize can be generated by a tool such as jmlic [5] or provided manually by the user, in the form of a list of entrypoint functions, such as isOrdered in Section 2.

DITTO’s approach to incrementalizing an invariant check c is to construct and maintain a computation graph, similar to, but less complicated than, a dynamic dependence graph [1], that reflects the most recent execution of c. The computation graph stores the final result of c, so incrementally updating the graph to reflect the new invocation of c amounts to incrementally recomputing c’s new final result. Intermediate results in the graph are reused where appropriate to avoid recomputation. This section of the paper describes the construction and maintenance of this graph in detail.

The computation graph consists of computation nodes, each of which stores the following information:

<table>
<thead>
<tr>
<th>f</th>
<th>explicit_args</th>
<th>implicit_args</th>
<th>children</th>
<th>return_val</th>
</tr>
</thead>
</table>

Here f is a function, explicit_args is a list of values, implicit_args is a list of heap and static memory locations, children is a list of edges to other nodes, and return_val is a value. A node represents the following information: when f is invoked with formal parameters explicit_args, it accesses static and heap locations implicit_args, makes recursive calls to children, and returns return_val. Callee return values are implicit, stored as the return values of child nodes.

Note that to construct a node, only f and explicit_args are provided; the execution of f(explicit_args) determines what implicit_args, children, and return_val are.

An edge from node x to node y in the graph means that y is a child of x: x invokes y with y’s explicit arguments, and receives a return value from y. The number of nodes is bounded by the size of c’s computation; furthermore, if c terminates, then the graph must be acyclic (if a function transitively invokes itself with the same inputs, this cycle will repeat and c will not terminate).

We now address the following questions: (1) How is the information comprising a computation nodes gathered? (2) How is the initial computation graph constructed? (3) How is the graph updated after data structure modifications?
3.1 Gathering computation node information

DITTO instruments each function \( f \) in an invariant check \( c \) during an offline transformation phase to record the data necessary to construct a computation node.

The transformation inserts code at the beginning of \( f \) to check for a memoized version of the function in the form of an existing node with the same explicit inputs already in the computation graph. This check is done via the `getMemoized` function.

```java
ComputationNode getMemoized(int funcId, Object[] args, ComputationNode parent) {
    ComputationNode n = memoMaps[funcId].get(args);
    if (n == null) {
        n = new ComputationNode(funcId, args);
        memoMaps[funcId].put(args, n);
    }
    graph.addEdge(parent, n);
    return n;
}
```

In this function, `args` are the explicit function arguments, `parent` is the computation node corresponding to the caller function, and `memoMaps` is an array of maps, each of which matches explicit arguments to computation nodes for a particular function.

Nodes whose implicit arguments have changed are removed from the memoization table before the instrumented function is run (see Section 3.3), and callee return values are optimistically assumed to be the same (see Section 3.4), so it is sufficient just to check for the same explicit arguments.

If a matching node already exists, the function immediately returns with its cached result; if not, a new node is created, and implicit arguments and the return value are recorded. In both cases, an edge is added to the graph from the parent (caller) node to the child (callee) node.

The `isOrdered` function in Figure 1 is instrumented as follows.

```java
Boolean isOrdered(IntListElem e, ComputationNode parent) {
    ComputationNode n = getMemoized(isOrderedId, [e], parent);
    if (n.hasResult) return (Boolean) n.result;
    n.addImplicit(addressOf(e.next));
    if (e == null || e.next == null) {
        n.setResult(true);
        return true;
    }
    n.addImplicit(addressOf(e.next.value));
    n.addImplicit(addressOf(e.next.value));
    if (e.value > e.next.value) {
        n.setResult(false);
        return false;
    }
    n.setResult(isOrdered(e.next, n));
    return n.result;
}
```

Here, `[e]` is an object array containing the single element `e`. In the implementation, the `addressOf` function uses the base address of `e` and a unique static identifier for each field it contains to uniquely identify individual fields of objects.

3.2 Initial computation graph construction

DITTO’s offline transformation phase also diverts all invocations of the check \( c \) – calls to functions in \( c \) from functions not in \( c \) – to the catch-all `doIncremental` runtime library function, which is detailed in Section 3.3. For instance, the call to `isOrdered(head)` in invariants in Figure 1 is rewritten to invoke `doIncremental` instead.

3.3 Updating the computation graph

After the initial run, the following invariant holds: the computation graph represents the series of calls, arguments, and results of the actual execution of \( c \). Thus the return value of the root node of the graph is equal to the return value of an uninstrumented \( c \), and can be used in its place.

As the program continues execution, writes to data structure elements or other inputs to \( c \) may break the invariant. The next time \( c \) is invoked, DITTO re-establishes the invariant by updating the compu-
tation graph to reflect the intervening modifications to the data that c accesses. By incrementally updating the graph, Ditto effectively incrementalizes c. The equivalence invariant ensures that the updated graph result is the same as the updated result that c would evaluate to if it were rerun from scratch.

Let g represent the graph corresponding to c invoked on its old inputs (which we know by the invariant is equivalent to constructing a graph from running c afresh on the old inputs) and g′ represent the graph corresponding to c invoked on its new inputs, after the intervening heap updates have occurred. The steps of the updating process that transforms g to g′ are detailed below. The pseudocode of the updating routine, doIncremental, is presented in Figure 5. The InvariantData structure contains all the incrementalization data structures pertinent to a particular invariant check instance. The steps detailed below are demarcated in the pseudocode as well.

**Step 1: Detecting changes to computation node inputs.** Nodes in the computation graph whose inputs have not changed since the previous computation are guaranteed to return the same result; they must be identical in g′ as in g. Thus, incrementally updating the graph must start with nodes whose inputs have changed.

We identify exactly how inputs to any computation nodes can change. Explicit arguments are passed in from a parent node, so they may only differ if the parent node itself has changed inputs. Similarly, callee return values may only differ if the child nodes have modified inputs. Hence, the only inputs that can change externally are (i) the initial explicit arguments to the entrypoint function of c (e.g. the head of isOrdered(head) in Figure 1), and (ii) implicit arguments used by any node (e.g. a particular IntListElem’s next field in the same example).

Changes to the initial explicit arguments are easy to detect, since they are passed during the entrypoint function’s first invocation. Changes to implicit inputs are detected by program-wide write barriers inserted by Ditto’s offline transformation phase. Details on their implementation are provided in Section 4.

When c is executed after heap modifications, doIncremental collects the modified implicit inputs caught by the write barriers (getWrittenLocations() in Figure 5), and identifies the the computation nodes (henceforth dirty nodes) that depend on any of them (getAffectedComputationNodes()) by consulting the reverse mapping created in Section 3.2; see Figure 4.

**Step 2: Recompute or remove nodes.** Once dirty nodes are identified, they need to be rerun to reflect the changes in the computation that result from the changes in their input. However, some nodes present in g, the pre-modification computation graph, may not be present in g′, the post-modification one. For instance, a list operation might remove the last 50 elements from a list. Computation nodes that rely on any of those elements, even ones that have changed (making the corresponding node dirty), should not be rerun, as g′ contains none of them; otherwise, exceptions or nontermination might result.

Thus, nodes closest to the root of the computation graph are rerun first, since a node with no dirty ancestors must still be in the graph (since no inputs to its transitive callers have changed, it must be invoked, and with the same explicit arguments).

To rerun a node n, Ditto executes n’s instrumented invariant check function f with its explicit arguments. This re-running may recurse down to (i) existing dirty nodes, which are rerun as well, (ii) function invocations on new arguments, which are then added to the graph in the form of new nodes, or (iii) existing non-dirty nodes, whose memoized results are immediately returned, ceasing that line of recursion. See Figures 6(a) and 6(b). In cases (i) and (ii), the computation graph is updated to store the new inputs for the rerun nodes.

In case (iii), the memoized result is used even though Ditto only knows that the explicit and implicit arguments to the node have not changed. It is possible for the callee return values to change

```java
Object doIncremental(InvariantData d) {
    Set locations = getWrittenLocations(); // Step 1
    Set dirty = d.getAffectedComputationNodes(locations);
    for (l : locations)
        // removes l from any nodes that depend on it
        d.removeFromImplicitMapping(l);
    for (n : dirty)
        memoMap[d.id].remove(n);
    List differing = [];
    while (dirty.size() > 0) { // Step 2
        ComputationNode n = d.getNearestToRoot(dirty);
        if (n.beenPruned) continue;
        List children = n.getChildren();
        n.removeChildEdges();
        Object old_result = n.result;
        Object new_result = d.run(n.funcId, n.explicitArgs);
        if (!old_result.equals(new_result))
            differing.addAll(n);
        for (c : children)
            if (c.getChildren().size() == 0)
                prune(c);
    }
    ComputationGraph g;
    for (n : differing) // Step 3
        // copies nodes; still in original graph
        g.addNodes(n.getParents());
        while (differing.size() > 0) {
            ComputationNode n = differing.remove(0);
            if (n.getChildren().size() == 0) {
                Object old_val = n.result;
                Object new_val = d.run(n.funcId, n.explicitArgs);
                if (old_val.equals(new_val))
                    removeToJoin(n, g);
                else
                    differing.addAll(n.getChildren()); // appends
                    g.remove(n);
            }
        }
    void removeToJoin() {
        if (n.getChildren().size() == 0)
            List parents = n.getChildren();
            g.remove(n);
            for (p : parents)
                removeToJoin(p, g);
    }
    void prune() {
        n.beenPruned = true;
        memoMap[d.id].remove(n);
        Set children = n.getChildren();
        n.removeChildEdges();
        foreach (c : children)
            if (c.getChildren().size() == 0)
                prune(c, d);
    }
```

Figure 5. Pseudo-code for the main incrementalizing algorithm. Code for creating the initial computation graph and dealing with changes to the initial arguments to the invariant check is straightforward and elided here. Code corresponding to each step described in the text is demarcated.
3.4 Optimistic memoization and function restrictions

During the rerunning step (Step 2), DITTO optimistically assumes that a memoized computation with the same explicit and implicit inputs will also have the same callee return values and thus will return the same value. This technique of optimistic memoization allows DITTO to aggressively limit the extent of recomputation necessary to account for local changes, while also allowing for a simple implementation.

Figure 7 shows the three forms of input to a node $m$ in a computation graph. When $m$ is invoked, explicit and implicit arguments are directly provided by the caller function and the heap, while the callee return value can only be known by executing the callee $m$ and all of its descendants. Let $m$ be dirty and $n$ not. The figure makes it clear that there is an cyclic dependency between $m$ and $n$, which is a problem if both $m$ and $n$ have modified inputs (say if $m$ and a descendant of $n$ are both dirty): which should be executed first? However, the cycle does not represent a true cyclic data dependency, since the actual execution occurs sequentially; it is instead an artifact of the construction of the computation graph. Thus, solutions exist for overcoming it.

One solution is to explicitly break the cycle, by separating $m$ into two nodes, the first of which, $m_a$, does not utilize the callee return value, and the second of which, $m_b$, does. Results would propagate forward only, from $m_a$ to $n$, and so on, eventually to $m_b$ and $l_2$. By never moving results backward, to already-visited nodes, dependency cycles are prevented. (Nodes may be split into more parts if there are more function calls.)

However, this solution would result in a much larger computation graph and is complicated to implement. It would require a form of closure and continuation passing, possibly in the form of extensive static rewriting, to invoke parts of a function with the necessary state.

The extra machinery required by this solution is only necessary in the case that $n$’s callee return values differs from the originals (since $n$ is not dirty, none of its other inputs could have changed). However, the more common case is that $n$ and its descendants all have unchanged inputs, so $n$ must evaluate to its cached value, breaking the cycle since $n$ does not need to be recomputed.

It is also possible for $n$ to have changed inputs and still return the same value, also successfully breaking the cycle. This case is especially common for data structure invariant checks, which often return the same “success” value (such as true) for subcomputations. So even if one of $n$’s descendants does have changed inputs, $n$ will only return a different value if the changes actually result in a violation of the invariant, a rare occurrence.

DITTO’s solution is to optimize for these common cases by optimistically assuming that $n$’s callee return values will be the same as they were when $n$ was first executed and its result memoized. In this way, $m$’s invocation of $n$ can be substituted with a memoized result instead of recomputation if just the explicit and implicit arguments to $n$ match.

This design decision simplifies DITTO’s incrementalization framework (fewer than 1000 lines of Java code in total), and keeps the computation graph small and simple: each function call is encapsulated by a single node, and data dependencies only flow back and forth along function invocations.

The first cost of optimistic memoization is recompeting $m$ when the optimistic assumption about a callee return value is incorrect. This occurs when some node further down in the graph returns a different value. Step 3 above will propagate this value upward, possibly recomputing $m$ again. In practice, we have found that this situation occurs rarely in data structure invariant checks, and when it does (such as when deterministically computing the sum of a list that has an element inserted near its head and near its tail), the cost of the incorrect optimistic assumption is dwarfed by the cost of propagating the new value through many other nodes.
The second cost deals with termination. Say \( n \) represents the function call \( f(e_1) \) and \( m \) represents the function call \( g(e_2) \) where \( e_1 \) and \( e_2 \) are the explicit arguments. If \( n \)’s return value ultimately differs from the cached value, \( m \)’s usage of the optimistically assumed (incorrect) value might cause \( m \) to loop infinitely or throw an exception or some other unexpected behavior that would not occur if the right value were used.

However, we know that whichever previous execution of \( f \) invoked \( g(e_2) \) did in fact terminate. Thus any nontermination could only arise through interactions with values specific to this invocation of \( f \), the explicit and implicit arguments of \( m \). So DITTO imposes the following restriction on the functions in a data structure invariant check that it is optimizing: no load from the heap, function call, or control flow decision can be based on both a callee return value and an explicit or implicit argument. The values can be combined in other ways, such as in the checkBlackDepth invariant in Figure 9.

This restriction ensures that optimistic callee return values will interact with the calling function in a consistent way (e.g. continued termination) even when the other arguments to that function change. Since invariant check functions are side-effect free, aliasing is not a problem, and the restriction can be verified by a simple static analysis. In practice, our experience is that this restriction is more of a technicality than a real burden; we have yet to find an invariant check we would like to write that violates it.

### 4. Implementation

DITTO is implemented as a Java bytecode transformation and accompanying runtime libraries. This approach does not allow for an optimized runtime implementation. For instance, the write barriers are implemented in Java, which requires two null checks and one array bounds check per barrier; an efficient JVM implementation would require far less overhead, as the barriers could be inserted at a lower level, circumventing these Java safety checks. However, the bytecode transformation approach offers the strong advantage of being as portable as Java is. It can be used with any JVM on any platform.

The implementation of DITTO supports multiple invariants per class instantiation, multiple class instantiations per class, and multiple classes. Below are specifics about some aspects of the implementation. The bytecode transformation is implemented using the excellent Javassist package [6].

**Hashing of objects.** In previous work on incrementalization [1], the definitions of object equality are left to the programmer; this flexibility allows the programmer to, say, equate two objects if they differ only in fields that she knows are irrelevant to the incremental computation. Since DITTO is automatic, an all-purpose strategy is required.

DITTO’s memoization table, which maps a list of explicit arguments, stored in an `Object[]`, to a particular computation node that represents a function call on those arguments, is implemented as a hash table for efficiency. This requires a notion of argument array equality and hashing. In terms of equality, pointer equality of `Object[]` is obviously insufficient. Instead, equality is defined as the conjunction of pointer equality for the elements (arguments) that are object references, and semantic equality for the elements that are primitive types; the hash code is defined analogously, as a combination of `System.identityHashCode()` or `Object.hashCode()` for the case of primitive types like `Integer` or `Boolean`. This strategy conservatively preserves the semantic equality of all arguments, while preventing sharing of non-primitive types (if the same computation node were to operate on two objects, semantically equal but in different locations on the heap, and only one was updated, then the node’s cached result could be incorrect for one set of arguments.) In theory, semantic equality and hashing could be applied to any immutable type.

Our benchmarks indicate that this conservative notion of equality, though not optimally flexible, performs well in practice on DITTO’s target domain.

**Efficient implementation of write barriers.** Since the write barriers are implemented in Java, some care must be taken to ensure reasonable performance. DITTO employs two main optimization tactics. First, during the offline bytecode transformation phase, DITTO gathers the set of fields accessed by the invariant checks it is optimizing. Write barriers are only inserted on updates to these fields, since only writes to these fields could possibly affect the implicit arguments to the invariant checks.

Secondly, each memory address caught by the barriers incurs a hash table lookup to determine what computation nodes are affected by its mutation, even if the object at that address is unrelated to any invariant checks and affects no computation nodes at all. If there are many such other writes (or if the first optimization did not sufficiently reduce the number of barriers inserted), these lookups
Boolean checkHashBuckets(int i) {
    if (i >= buckets.length) return true;
    return checkHashElements(buckets[i], i) &&
    checkHashBuckets(i+1);
}

Boolean checkHashElements(HashElement e, int i) {
    if (e == null) return true;
    return (e.key.hashCode() % buckets.length == i) &&
    checkHashElements(e.next, i);
}

Figure 8. Invariant for the hash table. The invariant is invoked as checkHashBuckets(0).

5. Evaluation

All measurements were performed on a Pentium M 1.6 GHz computer with 1GB of RAM, running the HotSpot 1.5 JVM.

5.1 Data structure benchmarks

We measured DITTO on several data structure benchmarks. Each data structure is instantiated at several sizes and then modified 10,000 times. We measured only small sizes (from 50 to 3,200) to reflect what we believe is common real-world usage. (Incrementalization generally produces asymptotic improvement, so arbitrary speedups can be had at large data structure sizes.) In each case, wall-clock time, including GC and all other VM and incrementalization overheads, is measured. The data structures and their modification patterns are described below.

If an operation requires a “random” element, it is selected at random from the set of elements guaranteed to fulfill the operation. For instance, the element for a deletion is chosen at random from the elements already in the data structure.

Ordered List. The OrderedIntList and its invariant isOrdered were described in Section 2. The modifications were 50% insertion of a random element, 25% deletion of a random element, and 25% deletion of the first element in the list (as in a queue).

50% random inserts and 50% random deletes.

void invariants() {
    if (!isRedBlack(root) || checkBlackDepth(root) == -1 ||
        !isSorted(root, Integer.MIN_VALUE, Integer.MAX_VALUE))
        complain();
}

Boolean isSorted(Node n, int lower, int upper) {
    if (n == nil)
        return true;
    if (n.key <= lower || n.key >= upper)
        return false;
    if (n.key <= n.left.key || n.key >= n.right.key)
        return false;
    return isOrdered(e.left, lower, n.key) &&
        isOrdered(e.right, n.key, upper);
}

Integer checkBlackDepth(Node n) {
    if (n == nil)
        return 1;
    int left = checkBlackDepth(n.left);
    int right = checkBlackDepth(n.right);
    if (left == right || left == -1)
        return left;
    return left + (n.color == BLACK ? 1 : 0);
}

Figure 9. Invariants for the red-black tree. nil is a special dummy node in the implementation that is always black.

Hash Table. The HashTable data structure maps keys to values, using chaining to store multiple entries in the same bucket. The invariant check, shown in Figure 8, verifies that no entry is in the wrong bucket. Note that the single invariant encompasses two functions. The modifications were 50% random insertions and 50% random deletes.

Red-Black Tree. We used the open-source GNU Classpath version of TreeMap, which implements a red-black tree in 1600 lines of Java. The invariants verify the required properties of a red-black tree, and check the following properties: (i) the tree is well-ordered (ii) local red-black properties (e.g. a red node has black children) (iii) the number of black nodes along any path from the root to a leaf is the same. See Figure 9 for the code. The modifications consisted of 50% random inserts and 50% random deletes.

A red-black tree is particularly well suited to dynamic invariant checks because

1. It is a complex data structure with nontrivial behaviors for even simple operations such as insert and delete that are hard to “get right”.
2. It has several invariants that are difficult to analyze statically but are relatively easy to write as code.

However, its complexity also challenges DITTO: a single operation can alter the data structure layout significantly, reordering, adding, and removing nodes. Additionally, two of the invariants en-
Boolean checkTop(int col) {
    if (col == width) return;
    return checkEmpty(col, top[col]) &&
    checkFull(col, top[col]-1) &&
    checkTop(col+1);
}

Boolean checkFull(int col, int row) {
    if (row == 0) return true;
    return jewels[col][row] != nullJewel &&
    checkFull(col, row-1);
}

Boolean checkEmpty(int col, int row) {
    if (row == height) return true;
    return jewels[col][row] == nullJewel &&
    checkEmpty(col, row+1);
}

Figure 11. The invariant check that verifies that a netcols grid has no floating jewels.

force global constraints, requiring nontrivial incremental updates to
the computation graph. For these reasons, we considered the red-
black tree an acid test for the feasibility of DITTO.

5.1.1 Analysis

The results of incrementalization for these data structures at vari-
ous sizes are presented in Figure 10. In each case DITTO success-
fully incrementalized the invariant, producing an asymptotic speedup
over the unincrementalized version. The average speedup at 3200 el-
ements is 7.5x.

DITTO performs well for medium to large sized data structures.
However, there is some baseline overhead due to write barriers and
the incrementalization data structures that have to be maintained. To
more closely analyze behavior on smaller data structures, for each
structure we measured the crossover size, the data structure size at
which it is faster to run DITTO’s incrementalized version of a check
than the original, all overheads considered.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Crossover Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordered list</td>
<td>≈ 250</td>
</tr>
<tr>
<td>Hash table</td>
<td>≈ 100</td>
</tr>
<tr>
<td>Red-black tree</td>
<td>≈ 200</td>
</tr>
</tbody>
</table>

These crossover sizes suggest that DITTO can be used as part of
the development process for programs with relatively small data
structures as well.

5.2 Sample applications

Netcols is a Tetris-like game written by a colleague in 1600 lines
of Java. Jewels fall from the sky through a rectangular grid and must
be made to form patterns as they land. The program keeps an array
top of the position of the highest landed jewels in each column, and
maintains the invariant that no jewels are floating – i.e. there are no
empty spaces below the highest spot in each column, and there are
no bejeweled squares about it; see Figure 11 for the code.

The main event loop averaged 80ms end-to-end time with the
invariant check running, noticeably sluggish. With DITTO, the event
loop averaged 15ms.

JSO [12] is a JavaScript obfuscator written in 600 lines of Java.
It renames JavaScript functions, and keeps a map from old names
to new so that if the same function is invoked again, its correct new
name will be used. However, functions whose names have certain
properties or that are on a list of reserved keywords should not be
renamed. Thus, we check the invariant that keys in the renaming map
do not meet any exclusionary criteria. See Figure 12. To enable this
invariant, we maintain an auxiliary list of map keys, names.

Figure 13 shows the results of feeding JSO JavaScript inputs of
varying sizes. DITTO’s incrementalized version of the check is able
to mitigate much of the overhead.

6. Related Work

Languages such as JML [13] and Spec# [4] provide motivation for
this work. These languages enable the user to write data structure in-
variant checks (among other specifications) directly into their code.
In some cases, these checks are statically verifiable, in which case
DITTO provides a complimentary solution: very small offline over-
head followed by a moderate runtime overhead, as opposed to a
larger offline overhead and no runtime overhead. On the other hand,
the cases where the checks must be verified at runtime are perfectly
suited to DITTO.

Software model checking [3, 9, 21] is a powerful technique for
static verification. However, most model checkers do not perform
well when required to maintain a precise heap abstraction, such as
when verifying red-black tree invariants, often failing to verify
structures of depth greater than five. Recent work by Darga et al. [7]
has made progress toward verification of complex invariants, but the
depth bound is still small for complex data structures and ghost
fields and programmer annotations are required.

Algorithm incrementalization has been the subject of consid-
erable research [8, 16, 17, 18, 11]; see [20] for a comprehensive
bibliograph of early work. Initial research often focused on hand-
incrementalizing particular algorithms [19].

Liu et al. began to devise a systematic approach to increment-
alization [15], culminating with recent work [14] that presented a

\[
\text{JSO performance}
\]

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>No invariants</th>
<th>Incrementalized</th>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5000</td>
<td>15000</td>
</tr>
<tr>
<td>5000</td>
<td>15000</td>
<td>20000</td>
<td>25000</td>
</tr>
<tr>
<td>10000</td>
<td>25000</td>
<td>30000</td>
<td>35000</td>
</tr>
</tbody>
</table>

\[\text{Figure 12. Invariant check for JSO that ensures that a protected function is not renamed.}\]

\[\text{Figure 13. Performance numbers for JSO.}\]

3 In [1], a crossover point is also mentioned, often occurring at size 1. Though
our attempt to contact the author failed, we imagine that this point is mea-
suring a different phenomenon, perhaps a theoretical crossover point without
runtime overheads.
semi-automated incrementalizer for object-oriented languages. This work differs from DITTO in two respects. First, it incrementalizes algorithms primarily through memoization (rather than a hybrid dependence/memoization solution), which may require recomputation even though true dependencies have not been modified. Second, it requires a library of hints, one for each type of input modification, that describe how the modification pertains to the incrementalization; DITTO allows for arbitrary updates.

Most recently, Acar et al. [1, 2] have developed a robust framework for incrementalization that uses both memoization and change propagation. This framework, for a functional language, exposes a number of library functions with which a programmer can annotate his program to incrementalize a wide variety of functions and achieve considerable speedups. In contrast, DITTO tackles a much smaller domain of programs, but is automatic, works for modern imperative languages, and has a simplified computation representation.

7. Conclusion
In this paper we have presented DITTO, a novel incrementalizer targeted towards a valuable set of functions, data structure invariant checks. By limiting its domain to a class of these checks and exploiting their common properties, DITTO is able to incrementalize automatically, for imperative languages like Java and C#, and simply, via optimistic memoization.

References