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

Ajeet Shankar
UC Berkeley
aj@cs.berkeley.edu

Rastislav Bodík
UC Berkeley
bodik@cs.berkeley.edu

Abstract

We present DITTO, an automatic incrementalizer for dynamic, side-effect-free data structure invariant checks. Incrementalization speeds up the execution of a check by reusing its previous executions, checking the invariant anew only on the changed parts of the data structure. DITTO exploits properties specific to the domain of invariant checks to automate and simplify the process without restricting what mutations the program can perform. Our incrementalizer works for modern imperative languages such as Java and C#. It can incrementalize, for example, verification of red-black tree properties and the consistency of the hash code in a hash table bucket. Our source-to-source implementation for Java is automatic, portable, and efficient. DITTO provides speedups on data structures with as few as 100 elements; on larger data structures, its speedups are characteristic of non-automatic incrementalizers: roughly 5-fold at 5,000 elements, and growing linearly with data structure size.

Categories and Subject Descriptors D.m [*Miscellaneous*]

General Terms Algorithms, Languages, Performance

Keywords Automatic, dynamic optimization, incrementalization, program analysis, data structure invariants, optimistic memoization

1. Introduction

Type safety of modern imperative languages such as Java and C# eliminates many types of programming errors, such as buffer overflows and doubly-freed memory. As a result, algorithmic errors present a proportionately greater challenge during the development cycle. One such class of errors are data structure bugs. Many data structures bugs can be detected as violations of high-level invariants 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." Verifying such invariants, however, remains non-trivial. Data structure invariants are particularly difficult for static tools to verify because static heap analysis scales poorly and current verifiers require extensive annotations.

An alternative approach is dynamic verification of invariant checks. Dynamic checks operate on the concrete data structure and are thus typically simple to write and validate. Thanks to tools such as `jmlc` [6], dynamic checking has become more accessible to programmers. However, dynamic checks can incur a significant run time overhead, hindering the development and testing. Since checks are executed frequently and commonly traverse the entire data structure, a program with checks may run 10–100 times slower, which may be prohibitively slow for all but the most patient programmer. Consequently, dynamic checks are rarely employed, even in debugging.

This paper introduces DITTO, an incrementalizer for a class of dynamic-data structure invariant checks written in modern imperative languages like Java and C#. We allow the programmer to write these checks in the language itself. DITTO then automatically incrementalizes such checks, rewriting them so that they only re-check the parts of a data structure that have been modified since the last check. Incremental checks typically run linearly faster than the original (about 10-times faster on data structures with 10,000 elements). We believe that the incrementalization makes dynamic checks practical in a development environment.

The goal of incrementalization is to modify an algorithm so that it computes anew only on changed input data and reuses all repeated subcomputations. Traditionally, incrementalization is designed and implemented by hand: an algorithm is modified to be aware of data modifications and to cache and reuse its previous intermediate results [20]. While hand-incrementalization can produce the desired speedups of invariant checks, manual incrementalization has several practical limitations:

- The programmer may overlook possible modifications to the data structure (as in the infamous Java 1.1 `getSigners` bug [11]) and thus omit necessary incremental updates. The result is an incorrect invariant check that may fail to detect bugs.
- Some invariant checks may be difficult to incrementalize by hand. For example, after some effort, we gave up on incrementalizing red-black tree invariants.
- Manual incrementalization does not appear economical, as each data structure may require several checks. Programmers may also want to obtain an efficient check rapidly, for example, when writing "data-breakpoint" checks for explaining the symptoms of a particular bug.
- Perhaps most importantly, incremental code is complex and scattered throughout the program. The complexity of its maintenance may defeat the purpose of relying on invariant checks that are simple and verifiable by inspection.

Recent research by Acar et al. [1] developed a powerful general-purpose framework for incrementalization of functional programs, based on memoization and change propagation. This framework provides an efficient incrementalization mechanism while offering the programmer considerable flexibility. To incrementalize a program, the programmer (i) identifies locations whose changes should trigger recomputation; and (ii) explicitly annotates reads and writes to

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006…$5.00

expose data dependencies. 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 develop an automatic incrementalizer. Our domain includes recursive side-effect-free functions, which cover many invariants of common data structures such as red-black trees, ordered lists, and hash tables. While we support only functional checks, the checks can be executed from within arbitrary programs written in imperative languages such as Java and C#. In these languages, checks are useful to the programmer because manual verification of invariants is complicated by the fact that data structure updates can occur anywhere in the program. For the same reason, incrementalization is difficult, which should make automatic incrementalization attractive.

Properties of invariant checks allow us not only to automate incrementalization but also to offer a simple and effective implementation.

- *Simplicity.* An invariant check typically returns always the same result (i.e., "the check passed") and so do its subcomputations that are recursively invoked on parts of the data structure. This observation allows us to develop *optimistic memoization*, a technique that aggressively enables local recomputations to reconstruct a global result.
- *Effectiveness.* The local properties that establish the global property of interest are typically mutually independent and recomputation of one does not necessitate recomputation of others. For example, sortedness of a list is established from checking that adjacent elements are ordered; if an element is inserted into the list, we need to check its order only with respect to its neighbors. Independence of local computations means that incremental computation can produce significant speedups.

The main contributions of this paper are:

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

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

2. Definitions and 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****, i.e., the values of its actual parameters; its* ***implicit arguments****, i.e., values accessed on the heap; and its* ***callee return values****, i.e., the results of function calls it makes.*

Note that implicit arguments are defined not to include locations that are read (only) by the callees of the function.

DEFINITION 2. *A* ***data structure invariant check*** *is a set of (potentially recursive) functions that are side-effect-free in the sense that they do not write to the heap, make system calls, or escape address of an object allocated in the invariant check. Furthermore, in each function, no loop conditional or function call can depend on any callee return values.*[1]

[1] This technical restriction, described further in Section 3.5, is required to ensure that the original functions and their incrementalized versions have the same termination properties in the presence of optimistic memoization. This restriction can be sidestepped but we have not found it to be an impediment in practice.

```
class OrderedIntList {
  IntListElem head;
  void insert(int n) {
    invariants();
    ...
    invariants();
  }
  void delete(int n) {
    invariants();
    ...
    invariants();
  }

  void invariants() {
    if (! isOrdered(head)) complain();
  }
  Boolean isOrdered(IntListElem e) {
    if (e == null || e.next == null)
      return true;
    if (e.value > e.next.value)
      return false;
    return isOrdered(e.next);
  }
}
```

Figure 1. The example class `OrderedIntList` and its invariant check `isOrdered`.

Throughout this paper, we will often assume that a check is a single recursive function. However, DITTO supports also checks composed of multiple recursive functions, such as the one in Figure 9. When the check contains multiple functions, we identify the check by the "entry-point" function that is invoked by the main program.

The incrementalizer memoizes the computation at the level of function invocations, so recursive checks are more efficient than iterative ones. Most iterative invariant checks can be rewritten without loss of clarity into recursive checks.

The main 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`, shown in Figure 1. The invariant verifies that the list maintains its elements in sorted order. The invariant is checked at method entries and exits. The former ensures that the invariant is maintained by modifications performed from outside the class. Such modifications could occur if, say, an `IntListElem` object was mistakenly exposed to users of the class. The latter ensures that the list operation itself maintains the invariant.

The invariant check is simple and readable, but it is inefficient. In common usage scenarios, the unoptimized `isOrdered` will dominate the performance of the program. However, the check is amenable to incrementalization under most common modifications to the list. For instance, if an element e is inserted into the middle of the list, `isOrdered` needs to be re-executed only on e and its predecessor; the success of the invocation of `isOrdered` preceding the change guarantees the checked property for the remaining elements in the list, as they have not changed since then. This incrementalization reduces the cost of the check from the time linear in the size of the list to constant time.

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

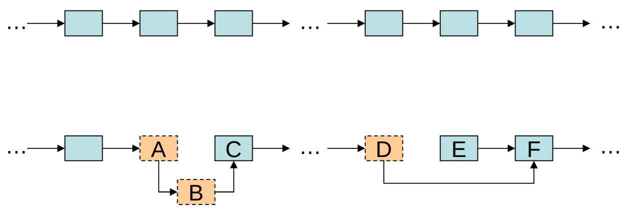


Figure 2. Before and after a list operation: an element is inserted, and another is deleted. The elements modified during the operation are dashed.

1. During the first execution of `invariants`, we record the sequence of recursive calls to `isOrdered`, their inputs, and their results.
2. During the subsequent execution of the main program, we track changes to memory locations that served as implicit inputs to a check. The tracking is performed with write barriers.
3. The next time `invariants` is invoked, we re-execute only the recursive invocations to `isOrdered` whose inputs have changed; we reuse memoized results for the remaining invocations. We update the memoized results so that further executions of the check can be incrementalized.

We describe these steps in detail in the context of a modification scenario, shown in Figure 2, where an element is inserted into the list and another element, further down the list, is deleted. Note that we assume that the invariant check is performed only after both modifications, and not in between the two modifications.

DITTO stores all inputs for each (recursive) invocation of `isOrdered`. The function `isOrdered` has five inputs: 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`.

The modification shown in Figure 2 updates two fields already in the list: `A.next` and `D.next`. Based on the inputs stored in the previous execution of the check, DITTO determines that these fields served as implicit inputs to invocations `isOrdered(A)` and `isOrdered(D)`. These two invocations must be re-executed on the new input values. Since `isOrdered(A)` occurred first in the previous execution, it is re-executed first.

The re-running of `isOrdered(A)` uses the new implicit arguments, specifically the new value of `A.next`, which now points to `B`. The execution thus continues to the invocation `isOrdered(B)`. Since DITTO has not yet encountered `isOrdered` with the explicit argument `B`, it adds this new invocation to its memoization table and continues executing, reaching the recursive call to `isOrdered(C)`.

At this point, DITTO determines that (i) `isOrdered(C)` has been memoized and (ii) the implicit arguments to `isOrdered(C)` have not changed since the previous execution of the check. However, this is not sufficient to safely reuse `isOrdered(C)` because there is no guarantee that the last input to `isOrdered(C)`—the callee return value from `isOrdered(C.next)`—will return the same value as in the previous execution of the check. The danger is quite real: There is a modification to an implicit input of `isOrdered(D)` further down the list; if `isOrdered(D)` returned a different value, this value could ultimately affect the value returned by `isOrdered(C.next)`. So, a straightforward memoization algorithm cannot safely reuse `isOrdered(C)`; instead, it must continue the re-execution until it is sure that no further callee return values might change. In our example, it would have to re-execute past `C` all the way to `D`, re-executing also all the intervening function invocations.

DITTO follows a more sophisticated algorithm. To deal with the uncertainty of callee return values, DITTO optimistically assumes that the recursive call to `isOrdered(C)` will return the same value as it did previously. This is a sensible assumption since recursive invariant check often do return the same value. (Typically, this is the "success" value.) This *optimistic memoization* strategy allows DITTO to reuse the cached result for `isOrdered(C)`, which successfully terminates the re-execution. The execution eventually returns back up to `isOrdered(A)`, which returns `true` — the same value it returned last time. Thus, the function that invoked `isOrdered(A)` need not be re-executed since all of its inputs are the same as last time; we are now done with the re-execution of the modified portion of the data structure around nodes `A` and `B`.

DITTO then re-executes `isOrdered(D)`, the second call whose implicit inputs have changed. The incrementalization then continues to `isOrdered(F)`, which is successfully reused, terminating the recursive calls. The invocation of `isOrdered(D)` evaluates to `true`, matching its previous result, so the entire recomputation ends. The invocation `isOrdered(E)` is no longer reachable in the computation and is ignored.

DITTO now returns the cached result of the entire invariant check, `true`, to the caller, `invariants()`.

Consider now the case when `isOrdered(D)` returns a value different than it did previously. (Note that our optimistic assumption is not necessarily wrong yet, as we assumed only that `isOrdered(C)`, but not `isOrdered(D)`, returns the same value as it did previously.) The new return value would be propagated from `isOrdered(D)` back up to its caller, which would be re-executed. This process would continue until either (a) a caller is reached that returns the same result that it did previously; or (b) the execution reaches the first caller, `isOrdered(head)`, and the new overall result is cached and returned. Note that if this upward propagation reaches `isOrdered(B)`, the optimistic memoization decision made when reusing `isOrdered(C)` is shown to be incorrect. In this case, `isOrdered(B)` is re-executed like the other calls during this propagation phase.

Whether the optimistic assumptions turned out wrong or not, the incrementalizer stores the new inputs and the result for each re-executed call; the memoization data for `isOrdered(E)` is garbage collected. This maintenance ensures that DITTO will be able to incrementalize the invariant check during its next execution.

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

3. Incrementalization Algorithm

This section presents details of our incrementalization algorithm. We start by describing the memoization cache and continue with a straightforward incrementalization algorithm. The inefficiency of this algorithm will motivate our optimistic incrementalizer, presented next. We will conclude by explaining the steps taken when optimistic assumptions fail.

3.1 Computation graph

On the first invocation of an invariant check, DITTO caches the computation of the check in a *computation graph*, which records the computation at the granularity of function invocations. Between invocations of the invariant check, the graph is used to track how the main program changes the check's implicit arguments. On the subsequent invocation of the invariant check, the graph is used to identify memoized function invocations whose inputs have been changed. These function invocations are re-executed and the graph is updated; the remaining function invocations are reused from the graph. The incrementally updated graph is equivalent to re-running the invariant check from scratch on the current program state.

The computation graph contains a node for each (dynamic) function invocation performed during the execution of the check. Directed edges connect a caller with its callees. DITTO stores the graph in memory in the form of a table. A table entry, shown below, represents one node of the graph, i.e., one function invocation. We will use the terms *function invocation* and *computation node* (or node) interchangeably as appropriate.

f	explicit_args	implicit_args	calls	return_val	dirty

The entry contains six fields: *f* is the invoked function; *explicit_args* is a list of values passed as actual arguments to *f*; *implicit_args* is a list static and heap locations read by the invocation; *calls* is a list of function invocations made by this function invocation, represented as links to other entries in the table; *return_val* is the return value of this invocation; and *dirty* is used during the incremental computation to mark invocations whose implicit inputs have been modified. Recall that *implicit_args* includes only the locations read by this invocation, not by its callees. The table is indexed by a pair $(f, explicit\_args)$.

DITTO constructs the computation graph by instrumenting the invariant check. The offline instrumentation diverts all invocations of an invariant check c — i.e., all calls to a functions in c from a function not in c — to the catch-all `incrementalize` runtime library function, described in detail later in this section (see Figure 7). For instance, the call to `isOrdered(head)` in `invariants()` in Figure 1 is rewritten to invoke `incrementalize()` instead.

DITTO instruments each function *f* in the invariant check c to record the data necessary to construct a memoization table entry. The instrumented version of the `isOrdered` function in Figure 1 is shown in Figure 3. The transformation inserts code at the beginning of *f* to check if this invocation has been memoized. If a table entry with the same explicit arguments already exists, the function returns with the cached result value; if not, a new entry is created and implicit arguments and the return value are recorded. The `try` and `catch` are required by optimistic memoization; their purpose is described later in this section.

In addition to recording the implicit arguments used by each function invocation, a reverse map, from heap locations (implicit arguments) to table entries, is created. This reverse map is used to determine which function invocations depend on modified heap values. See Figure 4 for an example initial computation graph.

Instrumentation is also used to track updates to implicit inputs. These updates can occur anywhere in the main program, so DITTO places write barriers into statements that might write those locations. The write barriers are described in further detail in Section 4. When an update to an implicit location is detected, function invocations whose implicit arguments have been modified are marked as *dirty*, which prevents reuse of their memoized results (see Figure 5).

3.2 Naive incrementalizer

DITTO can reuse the cached result of a function invocation if the function is invoked with identical inputs as the cached invocation. However, checking whether all inputs are identical is non-trivial. Recall that, for the purpose of memoization, a function has three kinds of inputs: explicit inputs (i.e., actual arguments); implicit inputs (i.e., values read by the function from the heap and static variables); and return values from its callees. Ideally, we want to reuse the cached result at the time when the function is invoked; but at this point we know only that the explicit arguments are identical. We may also know that the values of implicit input locations have not changed since the last invocation, but is the function going to read the same set of locations? The answer depends on the return values from the function's callees; if they differ from the previous return values, the function may read different locations and clearly cannot be reused.

```
Boolean isOrdered(IntListElem e) {
  try {
    // creates a new entry if one doesn't exist
    MemoEntry n = getMemoEntry(isOrderedId, [e]);
    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.value));
    n.addImplicit(addressOf(e.next.value));
    if (e.value > e.next.value) {
      n.setResult(false);
      return false;
    }
    n.addCall(isOrderedId, e.next);
    n.setResult(isOrdered(e.next));
    return n.result;
  } catch (Exception e) {
    throw new OptimisticMemoizationException();
  }
}
```

Figure 3. The instrumented version of `isOrdered()`.

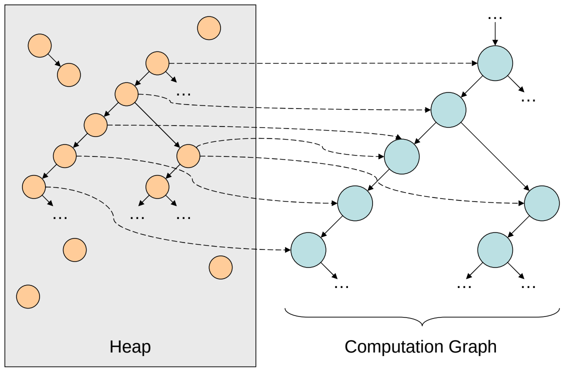


Figure 4. Part of the computation graph after an initial run. The dotted lines from items on the heap to computation nodes (function invocations) indicate the implicit arguments used by those nodes. Not all dotted lines are shown.

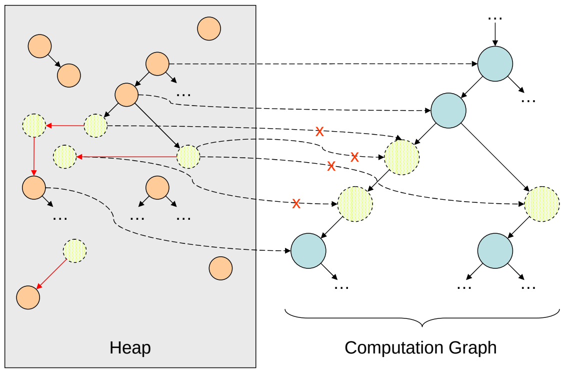


Figure 5. Memory locations with dashed outlines have been modified since the last execution of the invariant. All computation nodes that used these memory locations are marked as dirty.

```
function incrementalize(f, initial_args)
  return memo(f, initial_args)

function memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].hasModifiedImplicitArgs())
    return exec(f, x)
  foreach (c in t[f,x].calls)
    // did the call return the same value as last time?
    old_return_val = c.return_val
    if (memo(c.f, c.explicit_args) != old_return_val)
      // memo lookup failed somewhere in c.f's call tree
      return exec(f, x)
  // conditions described in Lemma 1 hold; reuse allowed
  return t[f,x].return_val

function exec(f, x)
  // invoke f', the instrumented version of f
  return f'(x)
```

Figure 6. The naive incrementalizer.

A conservative rule for reuse of memoized results is to ensure that (i) the explicit arguments are identical and that there has been no change to (ii) implicit input values as well as to (iii) callee return values. (To confirm that the return values are identical, the naive incrementalizer will incrementally execute the calls, meaning that it will try to reuse as much of the calls as possible.)

LEMMA 1. *Consider an invocation of function f that (i) has explicit arguments e, (ii) accesses the set of heap locations I, and (iii) invokes functions $g_1(a_1), \ldots, g_n(a_n)$, which return values $r_1, \ldots, r_n$. The cached result for this invocation is identical to the value of $f(x)$ invoked in the current program state if the following conditions hold: (1) $x = e$; (2) the locations in I have not been modified since the memoized invocation was executed; (3) $g_1(a_1), \ldots, g_n(a_n)$, if invoked on the current program state, would return the same values $r_1, \ldots, r_n$ as at the time of the previous invocation of $f(e)$.*

The proof involves showing that the current function invocation (1) accesses the same set of locations as the cached invocation; and (2) makes identical function invocations as the cached invocation. It is easy to show that if the previous implicit locations have not been changed, the first call made by the function is identical to the first call made by the cached invocation. If this call returns the same value as previously, the function will continue accessing the same implicit input locations. Since their values have not been changed, the second call made by the function will be identical to the second call in the cached invocation. The proof then proceeds by induction.

The naive incrementalization algorithm is shown in Figure 6. In this code, `initial_args` are the arguments provided to the first, entry-point function call of the invariant check. `t[f,x]` represents a lookup in the memoization table of an entry with function f and explicit arguments x.

The naive incrementalizer is simple: starting from the first function invocation of the invariant check, it recursively follows the path of the computation, reusing memoized results where appropriate. However, it is very costly: in order to ascertain that child calls do in fact return the same values as in the previous execution, it requires a memoization table lookup for every function invocation in the computation, even those that are unaffected by any input modifications.

3.3 Optimistic incrementalizer

Ideally, the incrementalizer should recompute only function invocations whose inputs have changed. But how do we determine that calls made by f would return the same values if executed in the current program state? The naive incrementalizer does so by "replaying" the sequence of all calls indirectly made by f and ensuring that all these transitive callees of f can be memoized. This process is expensive. A constant-time memoization check can be performed by time-stamping the invocation of each function and checking if any transitive callee of f has had its implicit arguments modified. Such a time interval mechanism was used by by Acar et al. [1] to aid in identifying relevant functions with changed inputs.

DITTO develops what we think is a simpler mechanism based on the common property that invariant checks usually succeed. When a check succeeds, it returns a success code; the same is true for all recursive function invocations made by the check. Our optimistic assumption thus is that *a function invocation in an invariant check typically returns the same value*. This observation holds even when some of the transitive callees had their implicit inputs changed because invariants usually hold even after the data structure is modified.

The *optimistic memoization* employed by DITTO simplifies the naive incrementalizer: we optimistically reuse a cached invocation if the explicit and implicit arguments are the same; the callee return values are assumed to return the same values. The optimistic `memo()` function is shown here:

```
function optimistic_memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].hasModifiedImplicitArgs())
    return exec(f, x)
  // optimistically assume that conditions
  // described in Lemma 1 hold and allow reuse
  return t[f,x].return_val
```

Optimistic memoization breaks dependencies of an invocation on its callees, whose return values are not yet known at the time when reuse of the invocation is attempted. This frees DITTO from having to perform the memoization lookup on many function invocations whose inputs have not changed. In other words, the benefit of optimistic memoization is that, in the common case of a successful check, we recompute only the local properties of those data structure nodes that have changed.

DITTO must of course handle the case of an incorrectly predicted optimistic value. The steps for doing so are detailed in Section 3.5.

3.4 The complete algorithm

The complete incrementalization algorithm, shown in Figure 7, needs to handle two more issues: *pruning of unreachable computations* and *recomputation in response to changed return values*.

Pruning. If a computation of a check has two function invocations with modified inputs, $f(x)$ and $g(y)$, and $g(y)$ is a transitive callee of $f(x)$, then we should recompute $f(x)$ before $g(y)$. The reason is that the new computation of $f(x)$ may or may not lead to an invocation of $g(y)$. Invoking $g(y)$ first could result in an exception, or it could be costly (for example, node y could have moved to a different data structure on which the evaluation of the invariant check could be expensive). Thus, DITTO re-executes dirty nodes (i.e., nodes with modified implicit inputs) in a breadth-first search order (i.e., nodes closest to the root are executed first). After each node is re-executed, the incrementalizer prunes nodes that are no longer in the computation graph; these nodes will not be re-executed.

Changed return values. When a re-execution of an invocation evaluates to a return value that differs from the cached return value, the changed value must be propagated to the caller of the recomputed invocation. DITTO tracks all nodes with differing return values and re-executes their callers in reverse breadth-first-search order, which ensures that a node is re-executed only after all its children have been re-executed (if that was necessary). This re-execution along a path continues up the graph until either (i) the return value of a re-

```
function incrementalize(f, initial_args)
  to_propagate = {}
  // identify memoized executions that have modified
  // implicit arguments (detected by write barriers)
  changed = get_changed_implicit_locations()
  changed_fns = map_locs_to_memo_table_entries(changed)
  // need to re-run root if arguments have changed
  if (t[f,initial_args] == null)
    changed_fns.add((f, initial_args))
  changed_fns.sort_bfs_order()
  foreach((f,x) in changed_fns)
    t[f,x].dirty = true
  foreach ((f,x) in changed_fns)
    // only re-execute if still in graph (not pruned)
    // and dirty (hasn't already been re-executed)
    if (t[f,x] != null && t[f,x].dirty)
      exec(f, x)
  propagate_return_vals()
  return t[f,initial_args].return_val

function memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].dirty)      // changed implicit_args
    return exec(f, x)
  // thanks to optimistic memoization, don't
  // need to check callee return values
  return t[f,x].return_val

function get_callers(f, x)
  // returns nodes that call f(x)

function exec(f, x)
  oldentry = t[f,x]
  // f' is the instrumented version of f
  newresult = f'(x)
  if (newresult != oldentry.return_val)
    to_propagate.add((f,x))
  foreach (c in oldentry.calls)
    if (get_callers(c.f, c.explicit_args).size() == 0)
      prune(c.f, c.explicit_args)
  return newresult

function propagate_return_vals()
  to_propagate.sort_reverse_bfs_order()
  while (to_propagate.size() > 0)
    e = to_propagate.remove(0)
    f, x, oldval = e.f, e.explicit_args. e.return_val
    newval = f'(x)
    if (oldval != newval)
      to_propagate.insert_reverse_bfs_order(
         get_callers(f,x))

function prune(f, x)
  var calls = t[f,x].calls
  t[f,x] = null
  foreach(c in calls)
    if (get_callers(c.f, c.explicit_args).size() == 0)
      prune(c.f, c.explicit_args)
```

Figure 7. Pseudo-code for the main incrementalizing algorithm.

executed ancestor evaluates to the cached value; or (ii) the root node is reached, which changes the overall result of the invariant check.

The DITTO incrementalizer is shown in Figure 7. In the implementation, the graph is not traversed using BFS; instead, the nodes are kept ordered using the order maintenance algorithm due to Bender, et al. [5].

An example of the algorithm in action (with pruning, optimistic memoization, and return value propagation) is shown in Figure 8.

3.5 Optimistic mispredictions

Recall that when the optimistic incrementalizer encounters a call to (a non-dirty) invocation $g(y)$, the incrementalizer reuses its old return value without first ensuring that the $g(y)$ would return the same value in the current program state. When this optimistic assumption is wrong, the re-execution of $f(x)$, the caller of $g(y)$, may go wrong in one of three ways:

The invocation $f(x)$ finishes evaluation but yields an incorrect result. In this scenario, no remedial action is needed. The correct return value will reach $f(x)$ during the propagation described in Section 3.4 and $f(x)$ will thus be re-executed with the correct return value and will produce the correct result.

The incorrect return value causes $f(x)$ to throw an exception. For example, $g(y)$ may return an object that is no longer in the data structure and may thus have invalid field values. When $f(x)$ reads those values, it may throw a divide-by-zero error or a null-pointer deference. Since this exception would not be raised in the non-incremental check, it must be prevented from reaching the main program. The code transformation described in Section 3.1 encloses the entire function in a try-catch block. If an exception is thrown due to a wrong optimistic assumption, the exception is caught and the execution of the function is stopped. The function will eventually be re-executed with correct inputs, as in the previous scenario. If an exception still occurs at this stage, the exception is forwarded on to the main program.

The incorrect value causes non-termination. Similar to the previous case, an incorrect return value may cause a loop or a recursion to iterate forever. The return value did not cause non-termination when $f(x)$ was executed previously because the explicit or implicit inputs were different. We offer two alternative remedial actions.

The first one, currently used by DITTO, imposes a restriction on the way DITTO-incrementalizable invariant checks must be written: *No loop conditional or function call can depend on a callee return value.* Here, *depends* includes both control- and data-dependence. Under this restriction, each loop and each call in the re-executed $f(x)$ uses only (correct) values from the current state, and thus will not cause a spurious non-termination.

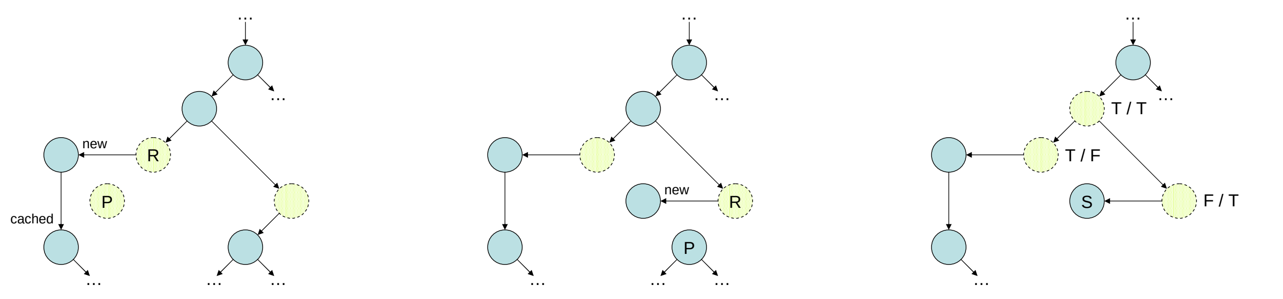
Our practical experience is that this restriction is more of a technicality than a real burden. We have yet to write a loop of any sort inside an invariant check function, and we have found it easy to overcome the function call restriction by avoiding short-circuit boolean evaluation.

To ensure that programmers are unable to violate this restriction, we have written a simple static analysis that checks for such a violation. The analysis is fairly trivial because aliasing is impossible in a side-effect-free function.

The second solution for this situation is to implement a timeout that would trigger when an optimistic execution takes far longer than it has taken historically. In this case, the invariant check would be re-executed from scratch. A benefit of this approach is that no programming restrictions are made on the function, though a cost is that its behavior may be unpredictable.

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.



(a) Re-execution and pruning (b) Re-execution and pruning (c) Changed return values

Figure 8. Re-execution after modification to the data structure shown in Figure 5. **(a)** The first dirty node, R, is re-executed. The re-execution in the node execution encounters (i) a new node, which is added to the graph, and (ii) a non-dirty node with a valid memoized value, which stops recursion early thanks to optimistic memoization. The dirty node P is pruned from the graph and will not be re-executed. **(b)** The second dirty node is re-executed. A new node is added and the non-dirty node marked 'P' and its children are pruned. Though not shown in the figure, memoization table entries are added or modified for the functions invoked in this step. The resulting computation graph reflects the changes made to the heap in Figure 5. **(c)** The results of re-executed nodes are compared with their old cached values. If they differ, the new results are propagated up through the graph. In this example, let the invariant check be a test for the presence of a special object S. Assume that S has moved from the left branch of the tree to the right; as a result, some node results differ ("F/T" indicates an old result of *false* and a new result of *true*), and are propagated up the graph. However, the propagation stops soon because an ancestor node's new result matches its old one.

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 [7].

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 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 entry that represents a function call on those arguments, is implemented as a hash table. 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 primitive types like `Integer` or `Boolean`. This strategy conservatively preserves 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 can cause significant overhead. To combat this phenomenon, the runtime portion of DITTO keeps a reference count of dependent invariant checks in the header of each object. The write barriers are constructed to first check that the reference count is greater than zero, and only then to add its field to the list of mutated ones. The reference count for a particular object is decremented when an invariant's hash table lookup is done and the dirty nodes identified. This way, if any of its dirty nodes accesses the object again, its reference count will be incremented. If not, since the dirty nodes are the only ones that accessed it beforehand, it is no longer relevant to that invariant check and does not need to be monitored further.

In practice, the inclusion of a 'header' reference count is implemented by creating a new class `IncObject` that inherits from `java.lang.Object`, and contains an integer field corresponding to the reference count. DITTO then sets the penultimate class in the class hierarchy of each object type used by invariant checks to inherit from this class instead of `java.lang.Object`.

Optimizing leaf calls. If a function f is invoked with arguments a that do not lead to recursion, it is often faster to compute $f(a)$ outright than to memoize it. This situation commonly occurs at the ends of data structures, when a final `null` value is reached. Thus, if all the non-primitive arguments to a function call are `null`, DITTO does not perform any cache lookups and instead runs $f(a)$ to determine the return value. In addition, small commonly used non-recursive functions, such as `hashCode()` and `size()`, are special-cased as well. In all cases, the implicit arguments to these functions, if any, are still recorded.

5. Evaluation

All measurements were performed on a Pentium M 1.6 GHz computer with 1 gigabyte 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

```
Boolean checkHashBuckets(int i) {
  if (i >= buckets.length) return true;
  boolean b1 = checkHashElements(buckets[i], i),
          b2 = checkHashBuckets(i+1);
  return b1 && b2;
}
Boolean checkHashElements(HashElement e, int i) {
  if (e == null) return true;
  return (e.key.hashCode() % buckets.length == i) &&
         checkHashElements(e.next, i);
}
```

Figure 9. Invariant for the hash table. The invariant is invoked as `checkHashBuckets(0)`.

```
void invariants() {
  if (!isRedBlack(root) || checkBlackDepth(root) == -1 ||
  ! isOrdered(root, Integer.MIN_VALUE, Integer.MAX_VALUE))
    complain();
}
Boolean isOrdered(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;
  boolean b1 = isOrdered(e.left, lower, n.key),
       b2 = isOrdered(e.right, n.key, upper);
  return b1 && b2;
}
Boolean isRedBlack(Node n) {
  if (n == nil) return true;
  Node l = n.left, r = n.right;
  if (n.color != BLACK && n.color != RED)
    return false;
  if ((l != nil && l.parent != n) ||
      (r != nil && r.parent != n))
    return false;
  if (n.color == RED && (l.color != BLACK ||
                         r.color != BLACK))
    return false;
  boolean b1 = isRedBlack(l), b2 = isRedBlack(r);
  return b1 && b2;
}
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 -1;
  return left + (n.color == BLACK ? 1 : 0);
}
```

Figure 10. Invariants for the red-black tree. `nil` is a special dummy node in the implementation that is always black.

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).

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 9, 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 deletions.

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 10 for the code. The modifications consisted of 50% random insertions and 50% random deletions.

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

1. It is a 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 enforce 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 various sizes are presented in Figure 11. In each case DITTO successfully incrementalized the invariant, producing an asymptotic speedup over the unincrementalized version. The average speedup at 3200 elements 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.[2]

	Crossover size
Ordered list	≈ 250
Hash table	≈ 100
Red-black tree	≈ 200

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 squares below the highest spot in each column, and there are no bejeweled squares about it; see Figure 12 for the code.

[2] 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 measuring a different phenomenon, perhaps a theoretical crossover point without runtime overheads.

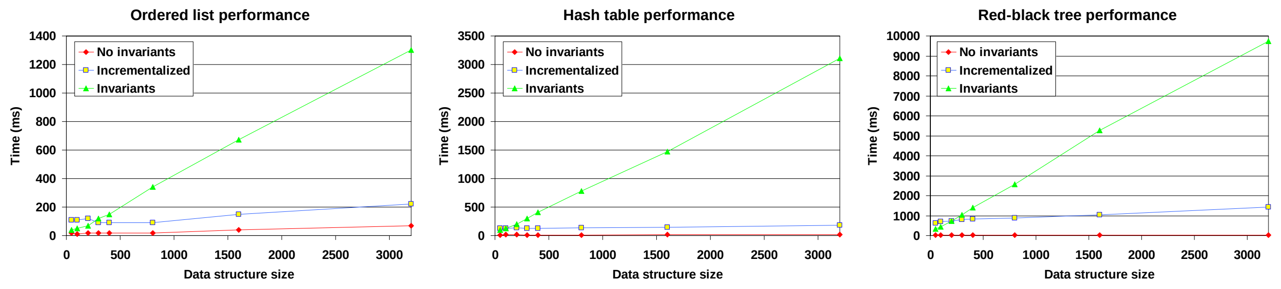


Figure 11. Results for data structure benchmarks. Each graph compares the performance of code with (i) no invariant checks (ii) standard invariant checks (iii) incrementalized invariant checks on different sizes of the data structure.

```
Boolean checkTop(int col) {
  if (col == width) return;
  boolean b1 = checkEmpty(col, top[col]),
          b2 = checkFull(col, top[col]-1),
          b3 = checkTop(col+1);
  return b1 && b2 && b3;
}

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 12. The invariant check that verifies that a `netcols` grid has no floating jewels.

```
Boolean goodMapping(JList names) {
  if (names == null) return true;
  String s = (String) names.value;
  if (Character.isUpperCase(s.charAt(0)) ||
      Character.isDigit(s.charAt(0)))
    return false;
  boolean b1 = ! inReserved(s, 0),
          b2 = goodMapping(names.next);
  return b1 && b2;
}

Boolean inReserved(String s, int off) {
  if (off == reserved_names.length) return false;
  return s.equals(reserved_names[off]) || inReserved(s, off+1);
}
```

Figure 13. Invariant check for `JSO` that ensures that a protected function is not renamed.

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 [13] 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 13. To enable this invariant, we maintain an auxiliary list of map keys, `names`.

Figure 14 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 [14] and Spec# [4] provide motivation for this work. These languages enable the user to write data structure invariant 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 overhead followed by a moderate runtime overhead and verification for testing inputs, as opposed to a larger offline overhead, no runtime overhead, and verification for all inputs. On the other hand, the cases where the checks must be verified at runtime are perfectly suited to DITTO.

Software model checking [3, 10, 22] 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. [8] 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 considerable research [9, 17, 18, 19, 12]; see [21] for a comprehensive bibliography of early work. Initial research often focused on hand-incrementalizing particular algorithms [20].

Liu et al. began to devise a systematic approach to incrementalization [16], culminating with recent work [15] that presented a semi-automated incrementalizer for object-oriented languages. This work differs from DITTO in two respects. First, it incrementalizes

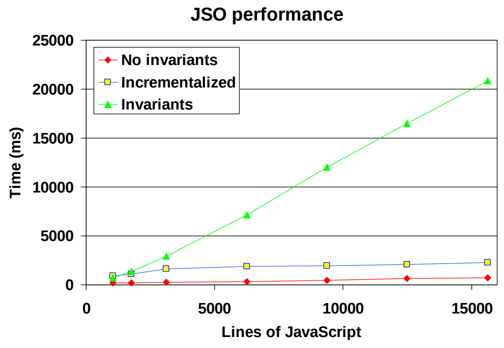


Figure 14. Performance numbers for `JSO`.

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 offers a number of library functions with which a programmer can incrementalize functional code functions and achieve considerable speedups. Acar's work and DITTO differ in several respects.

Acar's incrementalizer operates in the context of a purely functional program in ML. Input changes and computation dependences must be specified explicitly by the programmer. The framework is general and, thanks to the functional environment, can incrementalize computations that return new objects. Dependencies are tracked at the statement level, which allows for very precise change propagation. However, to achieve this granularity, functions must be statically split into several components, so that individual statements can be executed directly. These sub-functions must then be converted to continuation-passing style.

In contrast, DITTO operates in Java. Incrementalization is done automatically via write barriers and automatic instrumentation. DITTO operates on the domain of data structure invariant checks: recursive, side-effect-free functions. Because the rest of the program may be arbitrarily imperative, functions that return new objects are not allowed (such objects may be modified and thus are unsuitable for memoization). However, many common invariant checks can be written despite this restriction. Dependencies are tracked at the function level, which obviates the need for function splitting and CPS conversion (as well as optimizations required to elicit good CPS performance from Java VMs). The suitability of optimistic memoization for invariant checks further enables a simple implementation. Though the function-level granularity can require more code to be re-executed than necessary, invariant check functions tend to be small, and executing an entire function is often nearly as fast as identifying the few statements in that function that actually have modified dependences and rerunning just those.

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

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 96–107, New York, NY, USA, 2006. ACM Press.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[3] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec-sharp programming system: An overview. `http://research.microsoft.com/specsharp/papers/krml136.pdf`.

[5] M. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, 2002.

[6] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[7] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany*, volume 2830 of *LNCS*, pages 364–376, September 2003.

[8] Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. *SIGPLAN Not.*, 41(10):363–382, 2006.

[9] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–116, New York, NY, USA, 1981. ACM Press.

[10] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and W Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[11] Hotjava 1.0 signature bug, 1997. `http://www.cs.princeton.edu/sip/news/april29.html`.

[12] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. *ACM SIGPLAN Notices*, 35(5):311–320, 2000.

[13] Jso. `http://shaneng.awardspace.com/#jso_description`.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[15] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 473–486, New York, NY, USA, 2005. ACM Press.

[16] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.

[17] Bob Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 58–71, New York, NY, USA, 1977. ACM Press.

[18] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

[19] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328, New York, NY, USA, 1989. ACM Press.

[20] G. Ramalingam. Bounded incremental computation. Technical Report 1172, Univ. of Wisconsin, Madison, Computer Sciences Dept., 1210 West Dayton St., Madison, WI 53706, USA, 1993.

[21] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, New York, NY, USA, 1993. ACM Press.

[22] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.