

Robust, Generic, Modularly-Verified Map: A Software Verification Challenge Problem

Derek Bronish

Department of Computer Science and Engineering
The Ohio State University
bronish@cse.ohio-state.edu

Hampton Smith

School of Computing
Clemson University
hamptos@clemson.edu

Abstract

Maps are a fundamental component in the development of modern software. The ability to associate keys with values in the manner of a partial mathematical function is important for a wide range of applications, and also accommodates a broad variety of implementations with diverse performance profiles. The foundational-yet-sophisticated nature of this problem makes it an ideal benchmark for software verification efforts. A tension between modular reasoning and the usefulness of the component can be observed particularly in systems where the keys may be reference types. Criteria for modular verification of a robust map component are elaborated, and existing attempts to verify a map component are surveyed.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: Correctness proofs

General Terms Design, Documentation, Verification

1. Introduction

Maps (also sometimes referred to as “dictionaries”) are a well-studied, foundational software component. While their behavior is intuitive to describe, many implementations are sophisticated and involve complex linked data structures or hashing. These properties, which make maps ideal as a teaching tool, also make them a useful testbed for verification efforts. A well-conceived dictionary has the potential to reveal many properties of a given verification system, some of which we introduce here for discussion later.

The first and most obvious property of a verification system is whether or not it permits verification attempts that can succeed: given an implementation, can it be verified against its specification? If the specification captures only some of the behavior of the data structure—for example that it performs no null dereferences—we say it can be *partially verified*, whereas if the specification captures all of the behavior we say it can be *fully verified*. If a verification system also guarantees termination, then a verified implementation is called *totally correct*.

A second property of a verification system is whether or not its verification can be completed *automatically*. As we elaborate upon in [7], *push-button verification* is an important milestone if fully verified software is to become an accepted norm.

Another important property of a system is whether it permits *genericity* of verified components. Since the amount of resources (in time, money, or people) invested in a verified component is likely higher than its non-verified counterpart (due to the need for specifications, mathematical theory development, etc.), it is important that a component permit maximum reuse so that this cost can be amortized over time. Genericity allows the component to be reused in multiple scenarios with different instantiations.

A final and much-neglected property is *modularity*. If a verification system is to scale, it must not require that previously-verified components be re-verified when used in a new context. To accomplish this, the specification of a component must be self-contained: the specification system must be flexible enough to permit each new component to be specified without changing the specifications of existing components, as doing so would require those components to be re-verified against the new specification. Furthermore, verifying an implementation of a component must not require the examination of any other *code*, only specifications of components mentioned.

This paper proceeds as follows: in Section 2 we will discuss some of the difficulties inherent in a modular, generic dictionary as they relate to programming language features, mathematical abstraction, and automated proving. We present the challenge problem in Section 3, and then investigate some existing attempts to verify maps in Sections 4 and 5.

2. Discussion

Map implementations exist in standard libraries for nearly every programming language. Using these implementations as a starting point, we can determine the baseline behavior expected by clients of a map: the behavior which, in an ideal world, would be fully verified as totally correct.

Java’s Map interface [1], which provides Java’s modern abstract conception of a map, will serve as a specific example. Java Maps are parameterized by their key and value types, cannot contain duplicate keys, and map each key to a single value. Key equality is determined using the `equals()` method, which may be unique for each class—for some classes this may be reference equality, but for others it may be overridden to implement true equality based upon the abstract values of objects. A remark in the interface description explicitly notes that care must be taken if keys are mutable in ways that affect `equals()` comparisons. It should also be noted that the key type will provide or inherit a `hashCode()` method for hashing purposes and that, by contract, this method must be *consistent with equals*—i.e., objects that are equal under `equals()` should have identical `hashCode()`s.

As the Java documentation implies, the failure to draw a sharp distinction between reference and value equality can cause issues

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.

PLPV’11, January 29, 2011, Austin, Texas, USA.

Copyright © 2011 ACM 978-1-4503-0487-0/11/01...\$10.00

for attempts to verify maps, or even to just reason about them as clients. This paper concludes by investigating some real systems that evince such difficulties, but first it is useful to consider a simple example. In Figure 1 we consider a map from some user-defined type, which represents rational numbers as integer numerators and denominators,¹ to integers. The map `m` uses hashing with a hash table size of two, and the client holds references `f` and `g`, the latter of which is an alias to a key inside the map.

The problem of reference equality is evident when we consider a client attempting to perform a lookup in `m` using `f`. Although the abstract value “one half” does indeed appear in the map, reference equality will not determine this because `f` points to an object not in the map at all. Furthermore, this example demonstrates that testing equality of *class members* is still not sufficient, because the same abstract value can be realized by multiple concrete values of the members. The problem of holding aliases to map contents is also illustrated in this figure: changes to a key inside the map are possible via `g`, and this can result in broken invariants on the data structure (e.g., a value may wind up in the wrong hash bucket). While Java’s `Map` leaves the behavior of the map undefined when keys are changed in a manner that affects value equality (as implemented by `equals()`) and informally requires `hashCode()` to respect `equals()`, verified maps will either have to rigorously and modularly specify these restrictions, or else deal very carefully with aliases like `g`.

As it is a core abstraction in a mature library, we feel Java’s `Map` interface specifies (albeit informally) a reasonable expectation of the functionality a dictionary should offer—functionality sophisticated enough that issues like those of Figure 1 must be foreseen and accounted for by prospective verifiers. Nuances such as the issue of reference equality could certainly have been specified away in Java’s `Map` and yet were not, indicating that they are reasonable expectations for the use of maps in practical programs.

3. Statement of New Challenge Problem

In an effort to precisely describe a software verification challenge problem that identifies common pitfalls along the lines discussed so far, we now elaborate the components and criteria that a verified map should contain and satisfy.

To begin, notice that the criteria can be organized into a natural hierarchy. There are strict requirements on the code and the verification system’s behavior that must be met in order for the problem to be considered “solved” in any meaningful sense, but there are also non-functional requirements that an optimal solution should meet to the greatest extent possible. Additionally, the criteria of this challenge problem can be divided into those which are intrinsic to the particular problem of verifying a map, and those which speak more broadly to the issue of scalable and practical software verification in general.

- Strict requirements for practical verification in general:
 - Verification must be modular, in that only *specifications* of components and procedures mentioned should be used to conduct proofs of code correctness, *not* implementations
 - Clients must be able to reason modularly about their own code, e.g., holding aliases to contents of another object must not interfere with the proper functioning of that object
- Strict requirements for the map challenge in particular:
 - Ability to define and undefine mappings
 - Ability to access a value via a key, based on *value equality*

¹The rational number example is inspired by a discussion of `equals()` in [6]

- Ability to check whether the map is empty
- Ability to iterate over mappings
- The possibility of key aliasing must be soundly dealt with. The requirement of immutable keys is not desired, but if necessary it should be specified in a modular way.
- Desired properties of practical verification in general:
 - The client-view behavior of components should be specified in a way that is both logically and textually divorced from any particular implementation
 - The amount of annotation that does not directly describe the client-view behavior should be minimized
 - The proof process should proceed in a “push-button” manner with minimal human guidance and programmer-supplied hints
- Desired properties of a map verification solution:
 - The map should be fully generic in keys and values, meaning that any type can serve in either role
 - Multiple qualitatively different implementations of the map should be verified (e.g., hashing, binary search tree, etc.)
 - Verify a client operation which creates a deep copy of a map, given the ability to copy key and value types
 - Verify a boolean client function that tests for value equality of two map objects, using value equality on the keys and values
 - Demonstrate the ability to disprove incorrect client code and broken implementations of map

There are also requirements on the information that actually constitutes a “solution” to a verification challenge problem. Clearly the specifications and code are core pieces, but in addition it is beneficial to readers and reviewers if output from intermediate stages of the proof process is included (verification conditions, automated translations to intermediate languages, etc.). Above and beyond these data, which pertain directly to the program proof, supplemental programs and their proofs help paint a more complete picture of the verification attempt. For example, client code that uses the verified component can aid in illustrating the extent of the system’s modularity, and incorrect component implementations that are refuted by the prover help indicate the soundness of the verification strategy. A full solution to the challenge should include all of these, and anything else that might be deemed useful to the verification community at large.

We conclude with an analysis of two existing verified maps in light of these criteria. Neither is a full solution to the problem proposed here, most notably due to their use of reference rather than value equality.

4. Map in Dafny

In [5], the authors present a map component that is specified, implemented, and verified using Dafny. Dafny verification [4] uses a custom, Java-like programming language with reference semantics for user-defined types (which can be generic), and program proofs are based on translation to the intermediate language Boogie [2].

The authors of [5] explain the map that they fully verify as totally correct uses “built-in equality” to compare keys. For primitive types like Dafny’s `int`, this approach works fine, but for user-defined types the built-in equality is reference-based. As discussed in Section 2, this is a problem for the usefulness of the Dafny `Map` component. In many applications one wishes the keys in a map to

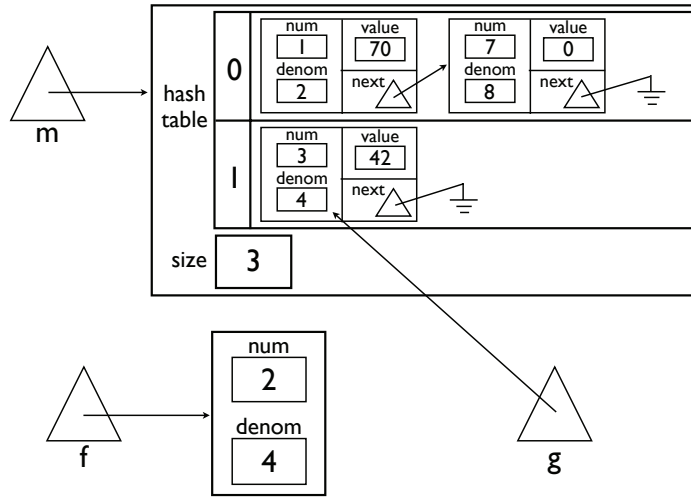


Figure 1. A hashing implementation of map using singly-linked lists for hash buckets. This scenario illustrates the problem of using reference- or class-member equality for key lookup (a client holding *f* will not be able to look up the mapping to 70 even though she holds the same abstract value as its key). Aliases to the map’s contents like *g* can also be problematic, for example if the map’s specification allows keys or their hashcodes to change.

be objects of some user-defined type, and furthermore one wishes map lookup to be based on *value* equality of said objects.

The Dafny specification of Map is both explicitly and implicitly committed to the reference-based built-in equality. For example, the contract for Add, which is shown in Figure 2, not only uses “==” between two keys, but it also involves set membership (denoted “in”), which is based upon this same notion of equality. Indeed, there appears to be a fundamental bias in the Dafny specification language toward built-in equality: it is not clear how the contract for Add could be written if value-equality of keys were desired. As we see in Section 5, one solution might be for all objects to inherit an equals() method that can be overwritten to provide useful weaker notions of equality, although the question of how to express both forms of equality (value and reference) in specifications is still an interesting one.

The issue of modularity is interesting to consider in light of Dafny’s specification style. Dafny proofs are claimed in [5] to be modular in the sense that components need not be re-verified each time they are used, but to understand a Dafny component’s specification seems to require some sort of breach of modularity. For instance, the specification of Map’s behavior is written in terms of ghost variables Keys and Values, which are explicitly updated inside of the implementation to track the “abstract value” of the map. However, there is no syntactic distinction between class members that are used for these specification purposes, and those that are pieces of the concrete representation.² One can imagine a second implementation of Map using different data structures, and see that either some subset of the existing Map’s specification would have to be copy/pasted into this new component, or else the same behavior

²The distinction between real and ghost variables is not sufficient for these purposes, because for example a ghost set is used to keep track of the allocated objects owned by the Map, though this is not a part of the map’s abstract value. If this set were part of the client’s view, there would still be a problem because the way in which its value is updated is underspecified by the ensures clause of Add, and can only be discovered by inspection of the code, again breaking abstraction.

(from a client’s perspective) would have to be re-specified in terms of the new class’ members. Either option is a problem for readability and the ability of clients to reason about the component without the need for breaking an abstraction boundary.

Dafny’s Map was written and verified in response to a challenge problem that did not make any demands on efficiency [8], but for the sake of discussion here we note that the implementation chosen has an unoptimized linear performance profile. Specifically, the map is represented by two “parallel” sequences, one containing the keys and the other the values. Faster implementations, e.g., using hashing or balanced binary search trees, raise interesting issues of specification, and can also potentially threaten modularity in ways that a straightforward slow implementation does not.

5. Map in Jahob

Jahob [3] is a system built on top of Java that attempts to capture all Java complexity in its specification language. It can generate VCs in multiple formats and has a focus on permitting multiple backend provers, along with the ability of the programmer to augment her code with hints [10] for these provers.

In [9], the authors present an overview of the Jahob verification system as applied to the full functional verification of a number of linked data structures. A few of the examples mentioned qualify as maps. We examine two (the primary example: an association list, and another: a hashmap) in some detail and evaluate them vis-à-vis the criteria of Section 3.

The primary example in [9] is an association list, implemented as a linked list of key/value pairs. From the perspective of performance and sophistication of design, it is similar to the example discussed in Section 4. Put() (for defining a new mapping) runs in constant time, and contains() and get() are linear. As in the Dafny example, only reference equality is used for matching keys, limiting the usefulness of the component for actual map applications.

As was the case with Dafny, this implementation is fully and automatically verified. With some reservation it could also be deemed

```

method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures (forall i :: 0 <= i && i < |old(Keys)| && old(Keys)[i] == key ==>
    |Keys| == |old(Keys)| &&
    Keys[i] == key && Values[i] == val &&
    (forall j :: 0 <= j && j < |Values| && i != j ==>
      Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]));
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);

```

Figure 2. The specification of Dafny Map’s Add procedure, which is used to define a new mapping

as modular, since keys and values of arbitrary type are permitted without impacting the correctness of the code, and linked list nodes are not permitted to leak from the list. There is still an issue with clients being able to hold references to keys and values inside of the map, and thus change the contents of the map without mentioning it (breaking the usual “frame property” that modular reasoning about client code relies upon), but if clients are always reasoning about the heap due to the reference-based nature of the programming language, this perhaps should not be considered an irreconcilable breach of modularity.

Another example of a fully verified dictionary mentioned in [9] is a `HashMap`. In terms of performance, this class is more similar to a map component that a client might expect: all operations run in constant expected time.³ Unfortunately, reference equality is again used for matching keys.

In considering section 3’s criteria regarding minimization of annotation as applied Jahob, we found that the association list verification uses seven lines of mathematical annotation (including hints to the back-end provers) for every ten lines of code.⁴ The hashing implementation uses twenty-two lines of mathematical annotation for every ten lines of code—an arguably prohibitive ratio.

The Jahob `HashMap` has a modularity issue in its handling of hashing, which arises from the ability of clients to retain references to keys once they are `Put` into the map. As was explained in Figure 1, changing a key via an alias can affect *which bucket* the mapping should be housed in. Since in Java (upon which Jahob is built), `hashCode()` is defined in the interface to the top-level `Object` class and its contract does not guarantee immutability, there is a real danger that an alias may change the bucket in which a key belongs. Though we find no mention of it in their paper, through personal correspondence with the authors (February 2009) we discovered that to guard against this *the interface of Object was changed to guarantee hash code immutability*.

This poses two problems: first, this implementation further restricts the legal key types, making it less generic than the association list example. Keys are only permitted if their hashcodes are immutable. Second, while immutable hashcodes may be a reasonable restriction (the Java `Map` indicates as much), the verification of this hash map required the modification of an external class’ specification and thus was not fully modular. It is interesting to consider how this problem could be solved while maintaining modularity: one idea might be to allow generics to place formal restrictions on their parameters, thus restricting the legal instantiations beyond just those allowed by the type signature.

³ For reasonable inputs. This implementation does not, for example, dynamically increase the number of buckets.

⁴ This ratio is a rough estimate, as it depends on formatting, but we use the authors’ original line breaks.

Acknowledgments

The authors would like to thank the following individuals for their input and feedback: Bruce Adcock, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Dustin Hoffman, Jason Kirschenbaum, Anirban Nandi, Bill Ogden, Murali Sitaraman, Paul Sivilotti, Aditi Tagore, Bruce Weide, and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. ECCS-0931669, DMS-0701260, DMS-0701187, CCF-0811737 and CCF-0811748.

References

- [1] *Java Platform Standard Ed. 6 Map Interface*, <http://download.oracle.com/javase/6/docs/api/java/util/Map.html>
- [2] M. Barnett, B. Chang, R. DeLine, B. Jacobs and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In F. de Boer, M. Bonsangue, S. Graf and W. de Roever, editors, *Formal Methods for Components and Objects*, number 4111 in Lecture Notes in Computer Science, pages 364-387. Springer Berlin/Heidelberg, 2006.
- [3] V. Kuncak and M. Rinard. An overview of the Jahob analysis system: project goals and current status. In *Proc. 20th IEEE International Parallel & Distributed Processing Symposium*, page 323. 2006.
- [4] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. To appear in *Proc. 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 2010.
- [5] K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. Leavens, P. O’Hearn and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, number 6217 in Lecture Notes in Computer Science, pages 112-126. Springer Berlin/Heidelberg, 2010.
- [6] H. Roumani. *Java By Abstraction*. Pearson Education Canada Inc., Toronto, Ontario, 2006.
- [7] M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith and B. W. Weide. Building a push-button RESOLVE verifier: progress and challenges. To appear in *Formal Aspects of Computing*, 2010.
- [8] B. W. Weide, M. Sitaraman, H. K. Harton, B. M. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, and Experiments (VSTTE 2008)*, pages 84-98. 2008.
- [9] K. Zee, V. Kuncak and M. Rinard. Full functional verification of linked data structures. In *Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 349-361. 2008.
- [10] K. Zee, V. Kuncak and M. C. Rinard. An integrated proof language for imperative programs. In *Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 338-351. 2009.