Dictionary Abstractions and Implementations in Isabelle/HOL

Bryan Ford

Introduction

This case study examines a very basic object-oriented abstraction: the dictionary. Fundamentally, a dictionary is a collection of (name, value) associations, generally allowing only one value for a given name. The standard minimum set of operations that any dictionary can be expected to support is as follows:

- **Insert(n, v)**: enter a new (name, value) pair, or if the name already has an association, replace its existing association with a new value.
- **Remove(n)**: remove any association that may exist for a given name.
- **Lookup(n)**: find the value currently associated with the given name, if one exists.

Of course, in practice dictionaries often provide additional capabilities, such as to iterate over all the (name, value) pairs in the dictionary, to remove and return an arbitrary entry, or even to compose multiple dictionaries together or split them apart. However, in this case study, I only deal with the minimal set of methods above.

There are a tremendous number of ways in which dictionaries can be implemented, with a wide variety of performance characteristics; a huge body of fundamental computer science theory is devoted solely to implementing this one abstraction efficiently. In this case study I used the Isabelle/HOL theorem prover to formalize and prove the operational correctness of three different well-known methods of implementing dictionaries: lists, hash tables, and binary search trees. First I will describe how the dictionary abstraction itself is formalized, then how each of the implementations in turn is formalized and proven correct.

In this formalization, I currently *only* deal with correctness, and not with complexity (runtime and storage characteristics). I would like to do so at some point, and with its growing support library for real numbers I expect this to become practical in Isabelle before too long; however, at the moment Isabelle's library does not yet have the exponential/logarithmic machinery needed to formalize worst-case complexity properties of algorithms such as these, let alone the probabiility theory that would be needed to formalize average-case behavior.

The Dictionary Abstraction

The first task of course was to formalize the notion of "dictionary" as an abstraction. In Isabelle/HOL, the obvious choice seemed to be to model a dictionary as a function from names to values. More specifically, a dictionary is an object with the following type in Isabelle/HOL (declared in **Dict.thy**):

```
types ('a,'b) dict = "'a => 'b option"
```

In other words, **dict** is a function type parameterized by two type variables - **'a**, representing whatever type is used for names, and **'b**, representing the type used for values. The function representing a dictionary takes a name and returns the type **'b option**, which is a basic parameterized type supported by Isabelle's standard library which formalizes the notion of an "optional value": i.e., a value which can be either present (expressed '**Some v**') or absent (expressed as '**None**'). The function representing a dictionary returns the value associated with the requested name, if any, or returns **None** if there is no association for that name.

Given this type definition, the three fundamental methods for dictionary objects are naturally formalized in Isabelle/HOL as higher-order function definitions:

Given this type definition, the three fundamental methods for dictionary objects are naturally formalized in Isabelle/HOL as higher-order function definitions:

```
constdefs
        dict_cons :: "('a,'b) dict"
        "dict_cons == (%n. None)"

        dict_insert :: "['a, 'b, ('a,'b) dict] => ('a,'b) dict"
        "dict_insert n v D == (%n'. if n' = n then Some v else D n')"

        dict_remove :: "['a, ('a,'b) dict] => ('a,'b) dict"
        "dict_remove n D == (%n'. if n' = n then None else D n')"

        dict_lookup :: "['a, ('a,'b) dict] => 'b option"
        "dict_lookup n D == D n"
```

Note that the percent sign is the lambda operator in Isabelle/HOL. **dict_cons** is simply the empty dictionary - i.e., the dictionary that returns **None** regardless of the name queried for; **dict_insert** constructs a new dictionary in which queries for the new name **n** will return the given value **v** and all other queries will be passed along to the previous dictionary; **dict_remove** similarly constructs a new dictionary in which queries for the removed name will cause **None** to be returned. The **dict_lookup** method, of course, is trivial: it simply invokes the dictionary as a function with the appropriate name.

Strictly speaking, the **dict** type defined in the **types** statement above is actually a supertype of the type of dictionaries we are actually interested in reasoning about: specifically, the **dict** type includes *all* possible functions from names to values, whereas we are actually only interested in reasoning about dictionaries that can be constructed from an empty dictionary from some sequence of insert and remove method calls - i.e., finite dictionaries. To express this constraint, I use Isabelle's inductive set definition mechanism to construct an explicit set representing all possible finite dictionaries:

```
consts
        dictset :: "('a,'b) dict set"
inductive "dictset"
    intros
        empty:  "dict_cons : dictset"
        insert: "[| D : dictset; dict_lookup n D = None |]
                 ==> dict_insert n v D : dictset"
```

Note that this set is built conservatively in the sense that it only builds dictionaries through sequences of **dict_insert** operations which insert values for names that weren't already in the dictionary. I later prove the stronger result that any dictionary resulting from any sequence of *arbitrary* insertions *and* removals is still in this set - i.e., that **dictset** indeed represents the closure of all possible finite dictionaries. Of course, the set could have been defined under the latter, stronger conditions in the first place. However, I found that defining the **dictset** more conservatively in this way makes it much easier subsequently to prove interesting properties about the elements of this set, because inductive proofs over the set only have to deal with insertions, and only non-colliding insertions at that.

One final step I could have taken, but didn't out of pure laziness (and lack of necessity for the purposes of this project), would be to use this **dictset** to declare a new Isabelle data type representing finite dictionaries. This way, the **dict** type could actually represent the *exact* abstraction we're trying to deal with, rather than an inclusive supertype.

Note that *all* of the above definitions, including the hypothetical "strict" finite dictionary data type if I had defined one, are purely conservative extensions: i.e., they are guaranteed by Isabelle to be logically sound and consistent as long as the basic HOL system is.

Based on these definitions, I proved a variety of basic properties of the dictionary abstraction, many of which are also useful as lemmas in future proofs about dictionaries and dictionary implementations. The vast majority of

Based on these definitions, I proved a variety of basic properties of the dictionary abstraction, many of which are also useful as lemmas in future proofs about dictionaries and dictionary implementations. The vast majority of the proofs are handled almost entirely automatically by Isabelle, simply by unfolding a few definitions and invoking the automatic reasoner (which uses a combination of simplification and natural deduction). A few examples of the types of properties of interest are:

- Result of lookups after various modification operations - e.g., looking up a name immediately after inserting a value for that name, or looking up a name after inserting a value for a *different* name.
- Equivalence of dictionaries constructed by consecutive modifications referring to the same name - e.g., the notion that inserting values for the same name twice causes the first value to be "forgotten."
- Order-independence of modification operations for different names.

Most of these lemmas naturally form simplification rules, which I feed to Isabelle's voracious simplifier by tagging them with '**[simp]**' in the source code; this causes them to be used automatically during future proofs.

The only proofs in the abstract dictionary theory that weren't completely trivial were the proofs mentioned earlier of the closedness of the **dictset**: i.e., that *all* possible sequences of arbitrary insertions and removals produce finite dictionaries that are still in **dictset**. These proofs still couldn't be called hard by any means, however; they mainly just involved invoking the appropriate induction rule initially (which Isabelle conveniently generated automatically when the **dictset** was defined in the first place) and leading the theorem prover through the appropriate sequence of case analyses.

The List Implementation

The first and simplest concrete, more-or-less realistic dictionary implementation is formalized in **ListDict.thy**. It defines another type alias, **listdict**, and a corresponding set of functions representing the method implementations, prefixed by **ld_** instead of **dict_** to distinguish them from the abstract dictionary's methods.

```
types ('a,'b) listdict = "('a * 'b) list"

constdefs
        ld_cons :: "('a,'b) listdict"
        "ld_cons == []"

        ld_remove :: "['a, ('a,'b) listdict] => ('a,'b) listdict"
        "ld_remove n D == filter (%(n',v). (n' ~= n)) D"

        ld_insert :: "['a, 'b, ('a,'b) listdict] => ('a,'b) listdict"
        "ld_insert n v D == (n,v) # ld_remove n D"

        ld_lookup :: "['a, ('a,'b) listdict] => 'b option"
        "ld_lookup n D ==
                (if (EX v. ((n,v) : set D))
                 then Some (SOME v. ((n,v) : set D))
                 else None)"
```

The formalization of this implementation takes advantage of Isabelle's powerful **list** data type, which is parameterized to support finite lists of any desired type. In this case, a **listdict** is actually a list of (name, value) pairs. The **ld_remove** method uses **filter**, a library function that filters a list using an arbitrary boolean function, to remove any elements referring to the name in question. The **ld_insert** method first removes any existing element(s) with the same name from the list, and then prepends a new element to the list with the new name and value. The **ld_lookup** implementation presented here is, strictly speaking, not very practical from an implementation perspective: it uses Isabelle's **set** function, which takes a list and returns the set of all elements it contains (ignoring any duplicates), and chooses an arbitrary element with a matching name from that set, on the hope that there's at most one. I don't have any great excuse for defining this method this way: it would have been more realistic, and probably not much harder, if any, to definite it in a more standard functional

contains (ignoring any duplicates), and chooses an arbitrary element with a matching name from that set, on the hope that there's at most one. I don't have any great excuse for defining this method this way: it would have been more realistic, and probably not much harder, if any, to definite it in a more standard functional programming style using a primitive-recursive function definition. But this way seemed a bit easier when I was first starting to work on this theory, and I never got around to going back and fixing it.

Given these definitions, it is just as trivial to prove most of the basic properties we would expect as it was for the abstract dictionary. The main exception is that the order-independence results that apply to the abstract dictionary type do not carry over to list-based dictionaries: i.e., inserting values for two different names in two different orders *do not* produce identical **listdict**s, since the resulting lists will have their elements in a different order. However, this is OK because what we're really interested in verifying is that the implementation *behaves* correctly for all possible sequences of method calls, and order-independence is not necessary in order to do that.

So finally we reach the Big Kahuna: proving that the list-based dictionary is behaviorally equivalent to the dictionary abstraction. The goal, as I formulated it, is to prove that after all possible sequences of modification operations, the result of any query will be the same for **dict** and **listdict**. Isabelle's support for inductive sets serves this purpose very well; I simply used it to define a simulation relation associating each possible **listdict** resulting from a given sequence of operations to a corresponding abstract **dict**:

```
consts
        dict_ld_rel :: "(('a,'b) dict * ('a,'b) listdict) set"
inductive "dict_ld_rel"
    intros
        init:   "(dict_cons, ld_cons) : dict_ld_rel"
        insert: "(D,L) : dict_ld_rel
                 ==> (dict_insert n v D, ld_insert n v L) : dict_ld_rel"
        remove: "(D,L) : dict_ld_rel
                 ==> (dict_remove n D, ld_remove n L) : dict_ld_rel"
```

Note that the single colon (**:**) operator is Isabelle's set membership operator. Once this relation is defined, proving behavioral equivalence is simply a matter of showing that **lookup** operations on corresponding elements in this relation always yield the same result:

```
theorem "(D,L) : dict_ld_rel ==> dict_lookup n D = ld_lookup n L"
 apply (erule dict_ld_rel.induct)
   apply (simp)
  apply (case_tac "n = na")
  apply (simp_all)
 apply (case_tac "n = na")
 apply (simp_all)
```

It is convenient that the dictionary abstraction, as formulated here, is completely deterministic: i.e., after any given set of modifications ("input actions"), there is one and only one valid result for each query method (the "output actions"). If, on the other hand, the dictionary abstraction was enhanced to support iteration over the contents of the dictionary, then the abstraction would presumably allow the contents to be returned in any order, and then the implementation-abstraction correspondence would not be one-to-one, but the possible outputs of the implementation would have to be shown to be a subset of the possible outputs of the abstraction: i.e., a full-fledged I/O Automata-style simulation relation. Implementing iteration support should not be too hard given Isabelle's facilities; doing so is left as an exercise for the reader. :-)

The Hash Table Implementation

The second dictionary implementation I formalized in Isabelle is the classic hash table (**Hash.thy**). This implementation assumes that the hash table size is fixed for the lifetime of the dictionary (i.e., it does not support dynamically growing and shrinking hash tables). However, it is parameterized by both the hash table size and the hash function, so it is guaranteed to be operationally correct for all possible hash table sizes and hash

dynamically growing and shrinking hash tables). However, it is parameterized by both the hash table size and the hash function, so it is guaranteed to be operationally correct for all possible hash table sizes and hash functions. The basic types and methods are defined as follows:

```
types 'a hashfunc = "'a => nat";

types ('a,'b) hashtab = "('a,'b) dict list";

constdefs
        ht_cons :: "nat => ('a,'b) hashtab"
        "ht_cons l == SOME H. (length H = l &
                                (ALL i. (i < l --> H!i = dict_cons)))"

        ht_remove :: "['a, ('a,'b) hashtab, 'a hashfunc] => ('a,'b) hashtab"
        "ht_remove n H F == list_update H (F n) (dict_remove n (H!(F n)))"

        ht_insert :: "['a, 'b, ('a,'b) hashtab, 'a hashfunc] => ('a,'b) hashtab"
        "ht_insert n v H F == list_update H (F n) (dict_insert n v (H!(F n)))"

        ht_lookup :: "['a, ('a,'b) hashtab, 'a hashfunc] => 'b option"
        "ht_lookup n H F == dict_lookup n (H!(F n))"
```

There are a couple oddities to this formalization worth noting. First, each method takes the hash function as a separate argument, whereas strictly speaking it would probably be conceptually cleaner to encapsulate it in the hash table data type. The only reason I didn't do it that way was because this way seemed slightly easier to work with since it doesn't require building and taking apart pairs during every operation; in other words, sheer laziness.

The other interesting thing to notice is that the hash table "implementation" actually uses the abstract dictionary: specifically, each entry in the hash table is actually an abstract dictionary, into which the hash table implementation simply "blindly tosses" all elements that happen to map to the same table entry. This formalization essentially skirts around the issue of handling hash collisions by punting the problem *back* up to the original high-level abstraction. It can certainly be argued that in doing this I haven't *completely* implemented a hash table, since in practice some concrete method will have to be used to deal with hash table collisions. However, I believe that this is in fact an excellent way to formalize the hash table "implementation" because *fully* implementing a hash table merely involves instantiating the hash table code as defined above, then recursively implementing the abstract dictionaries used in the hash table implementation by instantiating some other dictionary imlpementation such as the simple **listdict**. However, since the exact implementation for these sub-dictionaries isn't specified here, *any* such implementation can be instantiated as long as it satisfies the basic properties that dictionaries must have: for example, hash collisions could instead be handled by building colliding elements into a binary tree. Thus, this formulation provides a good separation of concerns and in that way leverages the abstraction/implementation distinction to yield greater flexibility.

The definition of **ht_cons** above, which simply constructs a hash table (list) of the appropriate length in which every entry is the empty dictionary, was probably not the easiest way to do this; it probably would have both been simpler and had a closer correspondence to a practical implementation if I had defined it as a primitive recursive function using Isabelle's **primrec** facility. As it was, it turned out that the most complicated proof required in the whole theory was the rather mundane proof that such a list as **ht_cons** tries to create actually exists. Ah, the virtues of hindsight.

Not too surprisingly, many of the proofs for the standard dictionary properties depend on a couple key invariants, which are encoded in those theorems as assumptions:

- The same hash function is used for all consecutive operations on the hash table.
- The hash function only returns values in the correct range given the size of the hash table.

After the standard properties are proved, I again prove behavioral equivalence by defining an inductive set that functions more or less as a simulation relation. The definition of the simulation relation is restricted according to

After the standard properties are proved, I again prove behavioral equivalence by defining an inductive set that functions more or less as a simulation relation. The definition of the simulation relation is restricted according to the invariants above, so that proofs using the relation automatically have those invariants available as premises. In addition, since as I mentioned earlier the hash function is passed to all of the manipulation methods as a separate parameter, the simulation "relation" actually consists of triples rather than pairs, so that the correct hash function is always "carried along" with a given hash table. Given this definition, the actual proof of behavioral equivalence was entirely identical to the proof used for list-based dictionaries.

The Binary Search Tree Implementation

The final dictionary implementation I formalized is the classic binary search tree. In the implementation I formalized, no balancing of any kind is done; extending this formalization to some of the popular balancing variants such as treaps, red-black trees, or splay trees should make for fun and interesting "future work." The formalization of the basic tree structure uses of Isabelle's ML-style discriminated union data type facility:

```
datatype ('a,'b) bt = Node "('a,'b) bt" "'a" "'b" "('a,'b) bt" | NoNode
```

In other words, a binary tree is either an empty leaf node (**NoNode**), or consists of a left subtree, a name, a value, and a right subtree, respectively.

Most of the method implementations use Isabelle's built-in support for well-founded primitive recursive function definition in order to operate recursively over this data type, which requires types to be declared separately from the definitions themselves:

```
consts
        bt_cons         :: "('a::linorder,'b) bt"
        bt_lookup       :: "['a::linorder, ('a,'b) bt] => 'b option"
        bt_insert       :: "['a::linorder, 'b, ('a,'b) bt] => ('a,'b) bt"
        bt_remove       :: "['a::linorder, ('a,'b) bt] => ('a,'b) bt"
        bt_scan         :: "('a::linorder,'b) bt => ('a * 'b) set"

        (* Internal helpers *)
        bt_merge        :: "[('a::linorder,'b) bt, ('a,'b) bt] => ('a,'b) bt"
        bt_ltall        :: "['a::linorder, ('a,'b) bt] => bool"
        bt_gtall        :: "['a::linorder, ('a,'b) bt] => bool"
        bt_sorted       :: "('a::linorder,'b) bt => bool"

defs
        bt_cons_def:
                "bt_cons == NoNode"

primrec
        "bt_lookup n' (Node l n v r) =
                (if n' = n      then Some v
                 else if n' < n then bt_lookup n' l
                                else bt_lookup n' r)"
        "bt_lookup n' (NoNode) = None"

primrec
        "bt_insert n' v' (Node l n v r) =
                (if n' = n      then Node l n v' r
                 else if n' < n then Node (bt_insert n' v' l) n v r
                                else Node l n v (bt_insert n' v' r))"
        "bt_insert n' v' (NoNode) = Node NoNode n' v' NoNode"

primrec
```

```
        "bt_insert n' v' (NoNode) = Node NoNode n' v' NoNode"

primrec
        bt_remove_Node:
        "bt_remove n' (Node l n v r) =
                (if n' = n      then bt_merge l r
                 else if n' < n then Node (bt_remove n' l) n v r
                                else Node l n v (bt_remove n' r))"
        "bt_remove n' (NoNode) = NoNode"
```

In addition, **bt_remove** requires a helper function, **bt_merge**, which merges the two disjoint subtrees "orphaned" when a node is removed into a single tree, in this case iby inserting the right subtree into the left:

```
primrec
        "bt_merge (Node l n v r) y = Node l n v (bt_merge r y)"
        "bt_merge NoNode y = y"
```

Finally, although these are all the methods that would theoretically be needed in a practical implementation of a basic binary search tree, a few additional helper functions are very useful for proving stuff:

```
primrec
        "bt_ltall n' (Node l n v r) =
                (n' < n & bt_ltall n' l & bt_ltall n' r)"
        "bt_ltall n' (NoNode) = True"
primrec
        "bt_gtall n' (Node l n v r) =
                (n < n' & bt_gtall n' l & bt_gtall n' r)"
        "bt_gtall n' (NoNode) = True"
primrec
        "bt_sorted (Node l n v r) =
                (bt_gtall n l & bt_sorted l &
                 bt_ltall n r & bt_sorted r)"
        "bt_sorted (NoNode) = True"
```

Essentially these functions define predicates which are used in the following correctness proofs to express the critical invariants that allow the binary search tree algorithm to function properly. The most important is obviously the last one, which builds on the first two: it expresses the property of a binary tree being properly sorted for searching.

Note that in the type definitions in the **consts** section above, the **'a** type for each method, which represents the type used for names, is constrained to the **linorder** axiomatic type class (second-order type). This Isabelle feature is analogous to Haskell's type classes, except that in this case type classes can be specified to have arbitrary mathematical properties: in this case, **linorder** is the class of all types for which an ordering relationship is defined on all members via the '**<**' operator, and the ordering relationship satisfies the standard properties expected for total orders (reflexivity, transitivity, etc.). The same effect could have been accomplished without the use of type classes by parameterizing the binary tree implementation with an explicit ordering function assumed to have the desired properties (analogous to the hash function used in the hash table implementation); however, given Isabelle's direct support for axiomatic type classes, doing it this way was naturally the simplest approach, not to mention the most cool. :-)

Unsurprisingly, proving that all of the desired dictionary-like properties hold for this binary tree implementation was not nearly as trivial as it was for the list and hash table implementations, but it was not that difficult either; mostly it involved figuring out the right intermediate lemmas to prove in the right order. Naturally, a good chunk of lemmas had to be proved right up front about the basic sortedness predicates above before anything interesting could be proven about the actual dictionary operations. However, once that was done, most of the proofs came fairly readily.

As usual, I finally defined a simulation relation between binary trees and abstract dictionaries using an inductive

proofs came fairly readily.

As usual, I finally defined a simulation relation between binary trees and abstract dictionaries using an inductive set and proved that for all corresponding modification sequences the result of a lookup in the two models is identical. This proof itself turned out to be no more difficult than it was for the previous implementations; all the real work happened in proving the basic properties of binary trees.

Isabelle does have one niggly problem that got in the way a few times: its automatic reasoning sometimes does not deal too well with fairly trivial inequalities. For example, it never could seem to figure out automatically that a proof goal with "a < b; b < a" as its premises is trivially valid. This is perhaps a likely area for future improvement in the tool. :-)

Undoable Dictionaries

Finally, in order to extend the basic dictionary abstraction a bit in a way that seemed interesting at the time (though admittedly may not be particularly realistic in this precise formulation), I created an extended "undoable document" abstraction (**Undo.thy**) and a corresponding implementation (**LogDoc.thy**). Essentially, a "document" in this model is simply a dictionary with one additional method: **undo**. Each call to a modification method (**insert** or **remove**) not only performs the appropriate modification operation but also saves a record of the modification so that any previous version of the "document" can be recovered at any time by invoking the **undo** method one or more times.

The difference between the abstraction and the implementation is simply in how this logging is done. In the abstract implementation, a "document" is simply a list of complete versions of the dictionary, so that after a valid sequence of modifications the head of the list is always the latest version and the tail is always the empty dictionary (the "original version"). The **undo** method in this case simply pops the first item off the list.

In the more interesting implementation of this abstraction, a document consists of only the current (head) version of the dictionary, in conjunction with an RCS-style "reverse change log" containing all of the information needed to get back to any previous version of the document. Each change log entry is represented by this discriminated union data type:

```
datatype ('a,'b) logentry =
          LogInsert 'a          (* Insert, not present *)
        | LogUpdate 'a 'b       (* Insert, already present *)
        | LogRemove 'a 'b       (* Remove, present *)
        | LogNull               (* Remove, not present *)
```

The first two entry types are created by **insert** operations and the other two are created by **remove** operations. The appropriate type of entry is selected in each case according to whether an association already existed for the given name before the modification operation: if so, then the previous value is saved along with the affected name. The **LogNull** entry type, created when a name is removed that was not already in the dictionary anyway, reflects the design philosophy that each "requested change" should create a new version, regardless of whether the operation really caused any *effective* change. Of course, this is one of those endlessly debatable fine points.

As with the basic dictionary abstraction and implementations, I proved behavioral equivalence between these models by first proving a number of basic critical properties for each model individually, and then defining a simulation relation between them and inductively proving that the result of a query is the same after any possible sequence of modifications. Since it's all just more of the same, I won't bore you with details.

Conclusion

Formalized a couple marginally interesting object-oriented data abstractions and implementation algorithms. Had fun. Isabelle is cool.

As far as making this case study available to others, I plan to submit it (with all the code) to the Isabelle folks in

[illegible]

As far as making this case study available to others, I plan to submit it (with all the code) to the Isabelle folks in case they want to include it in the Isabelle distribution as another example theory.

Cheers,
Bryan