QuickChecking Patricia Trees

Jan Midtgaard

DTU Compute, Technical University of Denmark

Abstract. Patricia trees are a space-efficient, purely functional data structure, useful for efficiently implementing both integer sets and dictionaries with integer keys. In this paper we illustrate how to build a QuickCheck model of the data structure for the purpose of testing a mature OCaml library implementing it. In doing so, we encounter a subtle bug, initially inherited from a paper by Okasaki and Gill, and since then flying under the radar for almost two decades.

1 Introduction

Obviously, a correct data structure is preferable to a buggy one. In this paper we illustrate how one can build a straightforward QuickCheck model for testing Patricia trees, a commonly used functional data structure. In doing so, we encounter a bug in a common Patricia tree library, inherited from a published paper [Okasaki and Gill, 1998]. Our paper thereby serves multiple purposes:

– as a pedagogical example of building a QuickCheck model,
– to document this error, and
– to illustrate the significance of generators for QuickChecking.

2 Background

We first recall the relevant background material on Patricia trees and QuickCheck.

2.1 Patricia Trees

A *Patricia tree* is a data structure for representing integer sets (and dictionaries) *compactly* and *functionally*. Historically Patricia trees were introduced 50 years ago by Morrison [1968]. Thirty years later they were recast as a functional data structure and popularized by Okasaki and Gill [1998]. The data structure works by inspecting and traversing the underlying representation of a set's numbers bit by bit (alphanumerically). Below we explain the little endian version that traverses the bits from the least to the most significant bit.

Elements in a Patricia tree are ordered similarly to a standard binary search tree. Specifically the order of elements is determined by a *branching bit* in all internal nodes: elements with a 0 in the branching bit belong in the internal node's left sub-tree, whereas elements with a 1 in the branching bit belong in

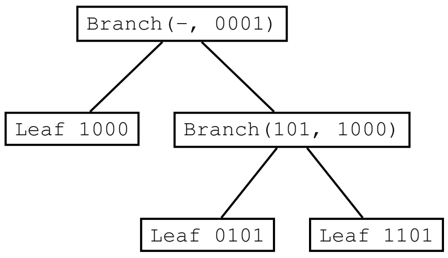


Fig. 1. The Patricia tree corresponding to the set $\{5, 8, 13\}$

the internal node's right sub-tree. For example, the branching bit of the root node in the Patricia tree in Fig. 1 is the least significant bit (the *parity* bit, `0001`, when we limit the presentation to only four bits). Therefore the even element 8 with bit representation (1000) belong to the left sub-tree, whereas the odd elements 5 (0101) and 13 (1101) belong to the right sub-tree.

To avoid needless branches the internal nodes of a Patricia tree also carry a shared *prefix* representing the string of bits that all elements in a given sub-tree have in common. For example, the elements 5 (0101) and 13 (1101) in Fig. 1 share the common prefix 101 but differ in the fourth bit (1000). The branching bit of the inner-most internal node thereby lets us distinguish the sub-trees' two leaf elements.

The `ptrees` library is a mature OCaml implementation of Patricia trees. For example, the Sawja library [Hubert et al., 2011] internally uses `ptrees` for efficient functional data structures, and Sawja is again used as the Java front-end in Facebook's static analyzer Infer [Calcagno and Distefano, 2011]. Like other data structures, such as red-black trees, we can use the Patricia tree structure to create both integer sets (by storing at each leaf only set-membership information) and to create dictionaries with integer keys (by storing at each leaf the entry associated with the given integer key). In `ptrees` this is realized by two sub-modules: One sub-module `Ptset` of `ptrees` implements integer sets whereas another sub-module `Ptmap` of `ptrees` represents dictionaries with integer keys.[1] In the following we will focus on the set implementation `Ptset`.[2]

Following Okasaki and Gill [1998], `Ptset` represents Patricia trees as an algebraic data type with three constructors:

```
type t =
  | Empty
  | Leaf of int
  | Branch of int * int * t * t
```

[1] The most recent version has simply split `ptrees` into separate packages `Ptset` and `Ptmap`, both of which are available through OCaml's package manager OPAM.

[2] The module `Ptset` also contains a sub-module implementing a big-endian version following the description of Okasaki and Gill [1998].

The first constructor `Empty` represents the empty set, the second constructor `Leaf` represents a singleton set, and the third constructor `Branch` splices together two sub-trees based on a shared prefix and a branching bit as explained above.

As an example operation, consider `mem : int -> t -> bool`, a membership predicate. The `mem` predicate can be implemented as a recursive function that pattern matches on the node type:

```
let zero_bit k bb = (k land bb) == 0
let rec mem k = function
  | Empty  -> false
  | Leaf j -> k == j
  | Branch (_, bb, l, r) -> mem k (if zero_bit k bb then l else r)
```

For empty trees and leaves `mem`'s code is straightforward: empty sets contain no members and a singleton set {`j`} contains only `j`. For internal nodes we test whether the branching bit `bb` is zero (after extracting it by a suitable logical *and'ing*), and continue the search recursively in the left (or right) sub-tree.

One interesting fact about Patricia trees is that they have a unique representation, meaning that identical sets will have identical structure. For now we will not concern ourselves with how Patricia trees are implemented under the hood but rather take a black-box view of the `Ptset` module for testing purposes. To this end we limit ourselves to the following subset of operations to keep things manageable:

```
val empty     : Ptset.t
val singleton : int -> Ptset.t
val mem       : int -> Ptset.t -> bool
val add       : int -> Ptset.t -> Ptset.t
val remove    : int -> Ptset.t -> Ptset.t
val union     : Ptset.t -> Ptset.t -> Ptset.t
val inter     : Ptset.t -> Ptset.t -> Ptset.t
```

All of these should be self-explanatory as operations over integer sets. The `add` operation for example expects an integer and a Patricia tree as arguments and returns a new Patricia tree representing the resulting, bigger set.

2.2 QuickCheck

QuickCheck [Claessen and Hughes, 2000] is also known as *(randomized) property-based testing*. As such, it builds on the idea of expressing a family of tests by a *property* (quantified over some input) and a *generator* of input. For the rest of this paper we will use OCaml's QCheck library.[3] As an example, consider McCarthy's 91 function:

```
let rec mc x =
  if x > 100 then x - 10 else mc (mc (x + 11))
```

[3] available at `https://github.com/c-cube/qcheck/`

This function is renown for being observably equivalent to the following simpler specification:

$$mc(n) = \begin{cases} 91 & n \leq 101 \\ n - 1 & n > 101 \end{cases}$$

(if we allow ourselves to ignore stack overflows due to the heavy use of recursion).

To test this property, we supply `Test.make` with the equivalence property and an input generator `small_signed_int` (a builtin generator of small signed integers from the QCheck library) to form a QuickCheck test:

```
let mc91_spec =
  Test.make ~name:"McCarthy␣91␣corr.␣spec" ~count:1000
    small_signed_int
    (fun n -> if n <= 101
              then mc n = 91
              else mc n = n - 10)
```

where we additionally specify the name of the tested property and the number of desired test runs (1000) as optional parameters `~name` and `~count`. We can subsequently run this QuickCheck test:

```
  QCheck_runner.run_tests ~verbose:true [mc91_spec]
```

and confirm the specification over the generated, small integer inputs:

```
  law McCarthy 91 corr. spec: 1000 relevant cases (1000 total)
  success (ran 1 tests)
```

Suppose we instead phrase a test of the incorrect property that McCarthy's 91 function is equivalent to the constant function always returning 91:

```
  let mc91_const =
    Test.make ~name:"McCarthy␣91␣constant" ~count:1000
      small_signed_int (fun n -> mc n = 91)
```

and run it, QCheck will immediately inform us of this failed property and print a minimal (*shrunk*) input for which it fails:

```
  law McCarthy 91 constant: 3 relevant cases (3 total)
    test `McCarthy 91 constant`
    failed on ≥ 1 cases:
    102 (after 30 shrink steps)
```

In this case it took the QCheck library 30 simplification steps to cut a failing input down to this minimal one, 102. Such shrinking is important in trying to understand the (often large) machine generated counterexamples on which a property fails. For example, if we disable the default, builtin shrinking over integers we may get a larger counterexample:

```
  law McCarthy 91 constant: 8 relevant cases (8 total)
    test `McCarthy 91 constant` failed on ≥ 1 cases: 4921
```

From the input 4921 it may be less clear what the underlying problem is.

In the Erlang community it is common to combine the randomized property-based testing approach with that of *model-based testing*. Concretely, this involves expressing an abstract model of the system (or module) under test and to test each of the available operations '*op*' for the property

the model and the implementation of 'op' agree

akin to how we have tested agreement between McCarthy's 91 function and its specification. For this reason the commercial QuickCheck implementation offered by QuviQ comes with a domain-specific language (DSL) for compactly expressing

- models,
- generators of arbitrary sequences of operations, and
- the above agreement property.

In the next section we will build an example model.

3 Building a Model

Following practice within the QuickCheck community [Claessen and Hughes, 2002, Hughes, 2010], we build a *model* of Patricia trees that distills their functionality to its core. Unlike the Erlang tradition we will explicitly express a model, a symbolic representation of operation sequences, a generator of arbitrary sequences of operations, and the agreement property. The following subsections are concerned with each of these.

3.1 A Model

A *model* serves as an executable specification of the intended meaning of a piece of software, similarly to how a *definitional interpreter* [Reynolds, 1972] specifies the intended meaning (the semantics) of a programming language. When Patricia trees are used to implement integer sets, we can easily model them using a list. For example, an empty set can be modeled with an empty list, a singleton set can be modeled with a singleton list, and the membership predicate can be delegated to `List.mem` from the standard library (assuming it has been thoroughly tested):

```
let empty_m = []
let singleton_m i = [i]
let mem_m i s = List.mem i s
```

where we suffix the operations with `_m` to underline that these operations belong to our model.

The distinguishing feature of sets, namely uniqueness of elements, surfaces when building a model for the remaining operations. For these we choose to maintain a sorted list representation. Based on this choice we can now implement a model straightforwardly. For example:

```
let add_m i s =
  if List.mem i s then s else List.sort compare (i::s)
```

where we rely on `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` which expects a comparison function as its first argument. The model for set union structurally recurses over its two argument lists, always putting the least element first, and thereby maintaining the sorted invariant:

```
let rec union_m s s' = match s,s' with
  | [], _ -> s'
  | _, [] -> s
  | i::is,j::js -> if i<j then i::(union_m is s') else
                     if i>j then j::(union_m s js) else
                       i::(union_m is js)
```

The remaining models for `remove` and `inter` are straightforward and therefore omitted here.

The critical reader may object that our model is no more than an inefficient implementation of the abstract data type of sets. We stress that this is a consequence of testing a *functional* data structure. The model-based QuickCheck approach was initially suggested (among others) for testing monadic code and has since been used successfully and repeatedly for locating defects in imperative code such as Google's LevelDB key-value data storage library[4] and the underlying AUTOSAR modules used in Volvo cars [Hughes, 2016].

3.2 Symbolic Operations

We first formulate a data type for symbolically representing calls to the `Ptset` API.

```
type instr_tree =
  | Empty
  | Singleton of int
  | Add of int * instr_tree
  | Remove of int * instr_tree
  | Union of instr_tree * instr_tree
  | Inter of instr_tree * instr_tree
```

Each of these constructors correspond to one of the operations listed earlier.

The alert reader may have noticed that we did not include a symbolic `Mem` constructor. The reason for this omission is simple: since a Patricia tree is a functional data structure, a query cannot affect it. For the purposes of representing and generating arbitrary Patricia trees a `mem`-query therefore has no effect. We will of course include a test for agreement between `mem` and `mem_m` in our forthcoming test suite to exercise the operation.

We can now write an interpreter for such instruction trees. Following the inductive definition the interpreter becomes a recursive function that interprets each symbolic operation as the corresponding Patricia tree operation:

[4] `http://www.quviq.com/google-leveldb/`

```
  (*  interpret : instr_tree -> Ptset.t  *)
 let rec interpret t = match t with
   | Empty        -> Ptset.empty
   | Singleton n  -> Ptset.singleton n
   | Add (n,t)    -> Ptset.add n (interpret t)
   | Remove (n,t) -> Ptset.remove n (interpret t)
   | Union (t,t') ->
     let s  = interpret t in
     let s' = interpret t' in
     Ptset.union s s'
   | Inter (t,t') ->
     let s  = interpret t in
     let s' = interpret t' in
     Ptset.inter s s'
```

For example, we interpret a `Singleton i` node as a call to `Ptset.singleton i` and we interpret a `Union` node by two recursive interpretations of the sub-trees and a `Ptset.union` of their results.

3.3 A Generator

In order to QuickCheck the above properties we need the ability to generate arbitrary trees of operations. Starting from the inside, the below expresses a recursive generator expressed using `QCheck`'s `Gen.fix` combinator:

```
  (*  tree_gen : int Gen.t -> instr_tree Gen.t  *)
 let tree_gen int_gen =
   Gen.sized
     (Gen.fix (fun rgen n -> match n with
       | 0 -> Gen.oneof [Gen.return Empty;
                         Gen.map (fun i -> Singleton i) int_gen]
       | _ ->
         Gen.frequency
           [(1,Gen.return Empty);
            (1,Gen.map  (fun i -> Singleton i) int_gen);
            (2,Gen.map2 (fun i t -> Add (i,t)) int_gen (rgen (n-1)));
            (2,Gen.map2 (fun i t -> Remove (i,t)) int_gen (rgen (n-1)));
            (2,Gen.map2 (fun l r -> Union (l,r)) (rgen (n/2)) (rgen (n/2)));
            (2,Gen.map2 (fun l r -> Inter (l,r)) (rgen (n/2)) (rgen (n/2)));
            ]))
```

Each invocation accepts a *fuel* parameter `n` to delimit the number of recursive generator calls. When we run out of fuel (`n = 0`), we hit the first branch of the pattern match and generate either a symbolic empty set or a singleton set. If there is still fuel left we choose between generating a list of things: empty sets, singletons, adds, removes, unions, or intersections. Since the latter four involves generating sub-trees we invoke the generator recursively, this time with a decreased amount of fuel. By the design of QCheck's fixed-point generator `Gen.fix` the recursive generator is passed as a parameter (above named `rgen`).

For flexibility we have parameterized the tree generator over the integer generator `int_gen` used in the singleton, add, and remove cases. We thereby avoid having to rewrite the tree generator to experiment with integer generation.

To increase the chance of generating adds, removes, unions, or intersections we assign them a higher weight (2), meaning that each of them will be chosen with probability $\frac{2}{1+1+2+2+2+2} = \frac{1}{5}$ whereas an empty set or a singleton is only generated with probability only $\frac{1}{10}$. For each of the recursive invocations we decrease the amount of fuel passed. Finally we wrap the size-bounded, recursive generator in a call to `QCheck`'s `Gen.sized` combinator, which first generates an arbitrary (small) integer and subsequently passes it as the fuel parameter to the size-bounded generator.

With the tree generator in place we can generate arbitrary trees from the top level. For example:

```
# Gen.generate1 (tree_gen Gen.int);;
- : Qctest.instr_tree =
Union
 (Union
   (Union (Add (1247377935267464492, Singleton (-344203684848058197)),
     Remove (788172988455234350, Empty)),
   Add (3495994339175018836, Singleton (-3950939914241702626))),
 Add (1460909625285095467,
  Inter (Singleton 3576840527525220675,
   Union (Empty, Singleton (-534074627919219807)))))
```

where we pass `Gen.int` as integer generator (a uniform generator of `int`).

Since OCaml does not supply a generic printer for use outside the top level, QCheck cannot print our trees in case it should find a counterexample. It is however straightforward to write (yet another) structural, recursive function `to_string` that serializes a symbolic instruction tree into a string. We can now express our generator with printing capability as follows:

```
(*  arb_tree : instr_tree arbitrary *)
let arb_tree = make ~print:to_string (tree_gen Gen.int)
```

where we make use of QCheck's `make` operation for combining the pure generator resulting from `tree_gen` with our pretty-printer `to_string` into a full generator (these are denoted by the parameterized type `'a arbitrary` in QCheck).

3.4 Expressing agreement

To express agreement between the implementation and our abstract model we need a final piece of the puzzle: the ability to relate one to the other. Following Claessen and Hughes [2002], we can do so with an abstraction function `abstract : Ptset.t -> int list`. We can simply implement `abstract` as an alias for the `elements` operation from the earlier versions of `ptrees`'s set API. In the recent API versions however, `elements` has been removed. In this case we can easily implement it as a fold, followed by a subsequent sorting:

```
let abstract s =
  List.sort compare (Ptset.fold (fun i a -> i::a) s [])
```

At last we are in position to test! For example we can write a test that expresses agreement between the singleton operation over both Patricia trees and our model:

```
let singleton_test =
Test.make ~name:"singleton␣test" ~count:2500
  arb_int
  (fun n -> abstract (Ptset.singleton n) = singleton_m n)
```

This expresses that creating a singleton set as a Patricia tree and abstracting the result as an ordered list should agree with our model interpretation over lists.

Similarly we can express agreement for the `union` operation:

```
let union_test =
  Test.make ~name:"union␣test" ~count:2500
    (pair arb_tree arb_tree)
    (fun (t,t') ->
      let s  = interpret t in
      let s' = interpret t' in
      abstract (Ptset.union s s') = union_m (abstract s) (abstract s'))
```

This expresses that the elements of two joined Patricia trees should give the same as taking the union of the elements for each tree.

3.5 Shrinking Trees

A sometimes neglected advantage of QuickCheck is *shrinking*: the ability to systematically cut down large machine-generated counterexamples to small ones that are easier for humans to understand. This mirrors the working routine of a software engineer: first recreate a run with an input exhibiting a bug, then systematically cut down the input (if possible) to a minimum in order to get to the heart of the error.

In QCheck shrinkers are implemented as iterators: a lazy stream of values. For example, `Iter.empty` creates the empty stream, `Iter.return v` creates the singleton stream containing only `v`, `Iter.of_list vs` creates a stream from a list `vs`, and `Iter.append` composes two iterator streams sequentially.

We can now express our shrinker as follows:

```
(*  tshrink : instr_tree -> instr_tree Iter.t  *)
let rec tshrink t = match t with
  | Empty -> Iter.empty
  | Singleton i ->
    (Iter.return Empty)
    <+> (Iter.map (fun i' -> Singleton i') (Shrink.int i))
  | Add (i,t) ->
    (Iter.of_list [Empty; t; Singleton i])
    <+> (Iter.map (fun t' -> Add (i,t')) (tshrink t))
    <+> (Iter.map (fun i' -> Add (i',t)) (Shrink.int i))
```

```
| Remove (i,t) ->
  (Iter.of_list [Empty; t])
  <+> (Iter.map (fun t' -> Remove (i,t')) (tshrink t))
  <+> (Iter.map (fun i' -> Remove (i',t)) (Shrink.int i))
| Union (t0,t1) ->
  (Iter.of_list [Empty;t0;t1])
  <+> (Iter.map (fun t0' -> Union (t0',t1)) (tshrink t0))
  <+> (Iter.map (fun t1' -> Union (t0,t1')) (tshrink t1))
| Inter (t0,t1) ->
  (Iter.of_list [Empty;t0;t1])
  <+> (Iter.map (fun t0' -> Inter (t0',t1)) (tshrink t0))
  <+> (Iter.map (fun t1' -> Inter (t0,t1')) (tshrink t1))
```

where the infix operation `<+>` is an alias for `Iter.append`. This shrinker codifies a systematic reduction: (a) We cannot reduce empty trees further. (b) We attempt to first replace a singleton with an empty tree, and otherwise shrink the singleton element itself. (c) We attempt to first replace addition and removal nodes with an empty tree, by dropping the node and keeping only the sub-tree, by replacing an addition node with a singleton node, by shrinking the sub-tree recursively, or by reducing the added or removed element. (d) In both the remove, union, and intersection cases, we first attempt to replace them with an empty tree, we then attempt to keep only a sub-tree, and finally we attempt to reduce sub-trees recursively.

With `tshrink` for shrinking trees, we enhance our generator with this ability:

```
(*  arb_tree : instr_tree arbitrary *)
let arb_tree =
  make ~print:to_string ~shrink:tshrink (tree_gen arb_int.gen)
```

where `arb_int` is some integer generator.

3.6 Refining The Integer Generator

We have expressed all tests in terms of `arb_int`, an (unspecified) integer generator. If we run our tests with `arb_int` implemented as a uniform generator `Gen.int` everything appears to work as intended:

```
random seed: 85075455
law empty: 1 relevant cases (1 total)
law singleton test: 2500 relevant cases (2500 total)
law mem test: 2500 relevant cases (2500 total)
law add test: 2500 relevant cases (2500 total)
law remove test: 2500 relevant cases (2500 total)
law union test: 2500 relevant cases (2500 total)
law inter test: 2500 relevant cases (2500 total)
success (ran 7 tests)
```

Here we have tested the agreement property between the model and `Ptset` across the 7 operations, each on 2500 arbitrary inputs, with the exception of `empty` which we only need to test once. Repeating this run (with different seeds for each run) does not change our perception. For example, if we repeat these 15.001 tests 10 times, totaling 150.010 tests the Patricia tree implementation still appears to function correctly.

The strategy of generating integers uniformly for our test cases may however be questioned. First, the chance of generating a corner case such as `0`, `min_int`, or `max_int` is 2^{63} each with OCaml's 63-bit integers (on a 64-bit machine). Yet the past decades of software engineering tells us precisely to remember to test such corner cases! How can we do so?

One way to adjust the integer generator to include such corner cases is to compose multiple different generators. For example, we can choose to either generate a `small_signed_int` (which includes 0), generate an integer uniformly (as above), or generate one of the two extremal corner cases:

```
let arb_int = frequency [(5,small_signed_int);
                         (3,int);
                         (1, oneofl [min_int;max_int])]
```

Here we have weighted each of these choices, by generating a `small_signed_int` with chance $\frac{5}{9}$, by generating an integer uniformly with chance $\frac{3}{9} = \frac{1}{3}$, and by generating `min_int` or `max_int` with chance $\frac{1}{9}$. Overall with this alternative integer generator we still have *some* chance of generating *all* integers, but the resulting distribution is skewed towards smaller numbers and corner cases.

Our `arb_int` is by no means the final word on integer generation. For some situations, e.g., our testing of McCarthy's 91 function, we would prefer to avoid generating duplicate numbers, as these represent redundant tests. In other situations (as we shall see shortly) we would precisely want a generator to emit duplicates. An orthogonal aspect is size: the builtin generators of QuviQ's commercial QuickCheck implementation is based on *generations*. The distribution of their integer generator `int()` thus initially generates smaller numbers but its output varies towards greater numbers as a property is repeatedly tested (generations goes by).[5] Testing and potentially catching errors over small inputs first will again reflect in time saved shrinking a needlessly big counterexample.

3.7 The Bug and Some Potential Fixes

If we try to run the test suite with the refined integer generator the framework quickly locates a problem:

```
law union test: 537 relevant cases (537 total)
  test `union test`
  failed on ≥ 1 cases:
  (Add (-4611686018427387904, Singleton 0),
   Add (-4611686018427387904, Singleton 1)) (after 10 shrink steps)
```

We identify the number `-4611686018427387904` as `min_int`, the least representable integer in 64-bit OCaml. With this in mind, the counterexample illustrates that a set union of the sets {`min_int`, 0} and {`min_int`, 1} does not yield

[5] `http://quviq.com/documentation/eqc/eqc_gen.html`

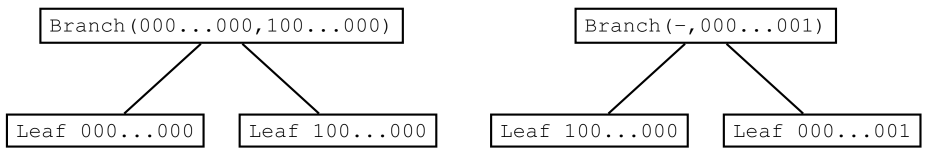


Fig. 2. The tree shapes resulting from `add min_int (singleton 0)` and `add min_int (singleton 1)`

{`min_int`, 0, 1}! What does it yield then? If one calls `abstract` on the resulting data structure it actually yields

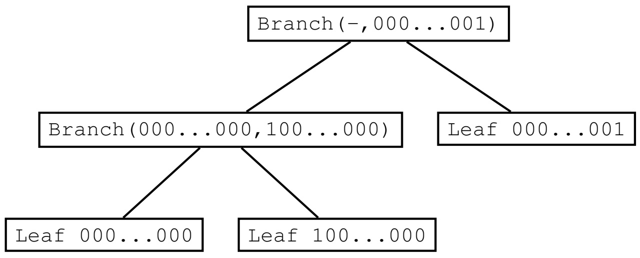
```
[-4611686018427387904; -4611686018427387904; 0; 1]
```

with a duplicate `min_int` entry!

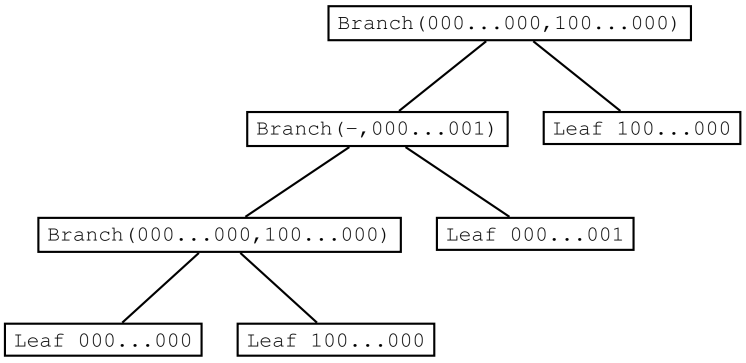
To understand the problem we must reopen the black box of `Ptset`'s implementation. First, since `min_int` is represented in 2-complement representation as a string of 0's with only a 1 in the sign bit, the left sub-tree resulting from `add min_int (singleton 0)` has the shape displayed on the left in Fig. 2. Similarly the right sub-tree resulting from `add min_int (singleton 1)` has the shape displayed on the right in Fig. 2. Now, the `union` operator simply performs a call to the internal `merge` operation, which is a recursive procedure for merging two Patricia trees:

```
let rec merge = function
  | t1,t2 when t1==t2 -> t1
  | Empty, t  -> t
  | t, Empty  -> t
  | Leaf k, t -> add k t
  | t, Leaf k -> add k t
  | (Branch (p,m,s0,s1) as s), (Branch (q,n,t0,t1) as t) ->
      if m == n && match_prefix q p m then
        (* The trees have the same prefix. Merge the subtrees. *)
        Branch (p, m, merge (s0,t0), merge (s1,t1))
      else if m < n && match_prefix q p m then
        (* [q] contains [p]. Merge [t] with a subtree of [s]. *)
        if zero_bit q m then
          Branch (p, m, merge (s0,t), s1)
        else
          Branch (p, m, s0, merge (s1,t))
      else if m > n && match_prefix p q n then
        (* [p] contains [q]. Merge [s] with a subtree of [t]. *)
        if zero_bit p n then
          Branch (q, n, merge (s,t0), t1)
        else
          Branch (q, n, t0, merge (s,t1))
      else
        (* The prefixes disagree. *)
        join (p, s, q, t)
```

The gist of the code is that it handles merging with empty sub-trees and leafs as separate cases (lines 3–6). In our situation above we hit the case of merging two branching nodes (line 7). This proceeds by a case analysis of the least significant branching bit: are the branching bits identical (and do the prefixes agree) (line 8–10), is one branching bit less than the other (and do the prefixes agree) (line 11–16 and 17–22), or is there some disagreement (line 23–25)? In our case the branching bits are `min_int` and `1` and comparing them with a signed comparison (line 11) yields true contrasting the intention of taking the *least significant bit*. From here on it is downhill. The empty prefix (`q`, represented as all 0's) of the right tree also has a zero sign bit (line 13), thereby causing `Leaf 000...000` to be merged recursively with the right tree (line 14). This boils down to invoking `add` (line 5) and results in a structure of the form:



which is in turn placed as the left sub-tree in the overall result by line 14:



and thereby explains the duplicate entry of `min_int` in the result and the disagreement between the implementation and our model.

In retrospect, we now realize that our shrinker constructed a minimal counter example: we need at least two branching nodes to hit line 7 and recreate the bug. In the current setting the error is also limited to a case with `min_int` occurring twice in order to be erroneously duplicated in the resulting list of elements.

One potential fix is to change the representation of the branching bit. After all we need only represent 64 different branching bit values on a 64-bit architecture which can be done with only 6 bits.[6] This fix is however a more invasive change throughout the module.

An elegant and less invasive patch was suggested by Jean-Christophe Filliâtre. The essence of the fix is to compare the two OCaml `ints` (a signed integer data type) albeit using an unsigned comparison. Since the only members we compare are branching bits on the binary form `0001`, `0010`, `0100`, ..., we can do so as follows:

```
let unsigned_lt n m = n >= 0 && (m < 0 || n < m)
```

which boils down to `n < m` for all non-sign-bit cases, yields false when `n` is a sign-bit (as desired), and yields true when `m` is a sign-bit (as desired). All sign bit comparisons in the code (incl. line 11 and 17) should thus be patched to call `unsigned_lt` instead. This fix furthermore has the advantage of costing only a few more comparisons in the common cases (assuming the call is inlined by the OCaml compiler).

The sub-module implementing the big endian version of sets and the `Ptmap` module implementing dictionaries contain the same problematic comparisons. They have all been fixed subsequently.

3.8 The bug and the research paper

The identified bug is not only relevant to users of `ptrees`, but to the functional programming community at large. Compare the listed OCaml `merge` function to the following SML merge function from Okasaki and Gill [1998, Fig.5]:

```
fun merge c (s,t) =
  let fun mrg (Empty, t) = t
        | mrg (t, Empty) = t
        | mrg (Lf (k,x), t) = insert c (k,x,t)
        | mrg (t, Lf (k,x)) = (c o swap) (k,x,t)
        | mrg (s as Br (p,m,s0,s1), t as Br (q,n,t0,t1)) =
            if m=n andalso p=q then
                (* The trees have the same prefix. Merge the subtrees. *)
                Br (p,m,mrg (s0,t0),mrg (s1,t1))
            else if m<n andalso matchPrefix (q,p,m) then
                (* q contains p. Merge t with a subtree of s. *)
                if zeroBit (q,m) then Br (p,m,mrg (s0,t),s1)
                                 else Br (p,m,s0,mrg (s1,t))
            else if m>n andalso matchPrefix (p,q,n) then
                (* p contains q. Merge s with a subtree of t. *)
                if zeroBit (p,n) then Br (q,n,mrg (s,t0),s1)
                                 else Br (q,n,t0,mrg (s,t1))
            else (* The prefixes disagree. *)
                join (p,s,q,t)
  in mrg (s,t) end
```

[6] Since OCaml's garbage collector reserves 1 *tag bit* in integers to distinguish them from heap-allocated data, in OCaml we need only 63 different values.

where the parameter `c : 'a * 'a -> 'a` is a *combining function* for resolving key collisions (useful when Patricia trees are used to represent dictionaries in general).

The comments and the structure of this code are the same as in the OCaml version: lines 2–3 handle merging with empty trees, lines 4–5 handle merging with singletons, and lines 6–19 handle the merging of two internal nodes with a 4-branch case analysis like the OCaml version: are the branching bits identical (and do the prefixes agree) (line 7–9), is one branching bit less than the other (and do the prefixes agree) (line 10–13 and 14–17), or is there some disagreement (line 18–19)?

The branching bit in the data type underlying the above operation is declared as SML's `int` type (also a signed integer data type):

```
datatype 'a Dict =
    Empty
  | Lf of int * 'a
  | Br of int * int * 'a Dict * 'a Dict
```

Since Okasaki and Gill's `merge` function contains comparisons `m<n` and `m>n` written using the signed integer comparison of SML it thereby exhibits the same problematic behavior as the OCaml version.

4 Related Work

Over the past 17 years QuickCheck has evolved from a Haskell library [Claessen and Hughes, 2000] to the present situation where ports have been made to many of the most popular programming languages.[7] In the process the approach has been extended to test imperative code [Claessen and Hughes, 2002] and a commercial port for Erlang has been developed by the company QuviQ. QuviQ's commercial port includes a compact state-machine DSL for easily specifying and testing such code with *abstract models* [Hughes, 2010] akin to the current paper. One notable difference between QuviQ's state-machine DSL and the model in this paper is that

– the state-machine approach is sequence-centric: it can be used to generate API call sequences (at its core describing a *regular language*) and test agreement between a model and an implementation's output and behaviour, whereas
– the example model we have presented is tree-centric (describing a *context-free language*).

The API of QuviQ's state-machine DSL has since been mimicked in Erlang's open source QuickCheck libraries PropEr[8] and Triq[9]. One can read this paper

[7] The Wikipedia page `https://en.wikipedia.org/wiki/QuickCheck` lists ports to 33 languages as of May 2017.
[8] `http://proper.softlab.ntua.gr/`
[9] `http://krestenkrab.github.io/triq/`

as an encouragement to enhance the many existing QuickCheck ports to other languages with state-machine frameworks to enable compact and powerful state-based QuickCheck models for Scala, OCaml, F#, Haskell, ...

Since its introduction property-based testing has successfully been applied to test and locate errors in a broad class of software: formal semantics [Felleisen et al., 2009], optimizing compilers [Pałka et al., 2011], type environments [St-Amour and Toronto, 2013], dynamic analyzers [Hriţcu et al., 2013], type systems [Fetscher et al., 2015], static analyzers [Midtgaard and Møller, 2015], and computational geometry [Sergey, 2016]. Common to many of these are that they are not model-based. For each particular domain, the involved operations are instead tested to satisfy domain-specific properties, e.g., non-interference [Hriţcu et al., 2013], lattice axioms [Midtgaard and Møller, 2015], or geometric properties [Sergey, 2016].

Recently there has been a trend towards letting a QuickCheck framework generalize the found counterexamples. SmartCheck [Pike, 2014] is a QuickCheck extension that can perform such generalization with the goal of explaining the general erroneous behaviour to the user. MoreBugs [Hughes et al., 2016] is another QuickCheck extension performing such generalization with the goal of avoiding repeated rediscovery of the same bugs. In practice this becomes a concern if a tester does not want to pause the testing process until the first round of errors is fixed or adjust his model specification to reflect the code's buggy behaviour [Hughes, 2016].

5 Conclusion and Perspectives

We have demonstrated how QuickCheck can locate a subtle bug in a published data structure paper after almost two decades — a bug which was also present in an influential library implementation.

For the purpose of bug-finding, the quality of a QuickCheck library's built-in generators is of utmost importance. Simple uniform generators are unlikely to exercise the corner cases that one would typically test by hand. As a consequence, a passing QuickCheck test suite based on such generators may give users a false sense of certainty in an implementation. Furthermore, for a QuickCheck library to be successful, the ability to efficiently shrink counterexamples is essential. Otherwise, the machine generated counterexamples simply get too big to be comprehensible for a human being. For both of these aspects, the commercial QuviQ QuickCheck implementation has a clear advantage, with several year's of effort in refining and engineering its generators and shrinkers.

The full source code of our developed tests is available for download at

`https://github.com/jmid/qc-ptrees`

Acknowledgments We thank Jesper Louis Andersen for sharing his expertise with alternative integer generators and Jean-Christophe Filliâtre for his Patricia tree library implementation and for promptly providing an elegant fix.

Bibliography

C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *LNCS*, pages 459–465. Springer-Verlag, 2011.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In P. Wadler, editor, *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 53–64, Montréal, Canada, Sept. 2000.

K. Claessen and J. Hughes. Testing monadic code with QuickCheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In J. Vitek, editor, *Programming Languages and Systems, 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *LNCS*, pages 383–405. Springer-Verlag, 2015.

C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In Morrisett and Uustalu, pages 455–468.

L. Hubert, N. Barré, F. Besson, D. Demange, T. P. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static analysis workshop for Java. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *LNCS*, pages 92–106. Springer-Verlag, 2011.

J. Hughes. Software testing with QuickCheck. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, volume 6299 of *LNCS*, pages 183–223. Springer-Verlag, 2010.

J. Hughes. Experiences with QuickCheck: Testing the hard stuff and staying sane. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *LNCS*, pages 169–186. Springer-Verlag, 2016.

J. Hughes, U. Norell, N. Smallbone, and T. Arts. Find more bugs with QuickCheck! In C. J. Budnik, G. Fraser, and F. Lonetti, editors, *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, pages 71–77. ACM, 2016.

J. Midtgaard and A. Møller. Quickchecking static analysis properties. In G. Fraser and D. Marinov, editors, *8th IEEE International Conference on Software Testing, Verification and Validation, ICST'15*, pages 1–10, Graz, Austria, Apr. 2015. IEEE Computer Society.

G. Morrisett and T. Uustalu, editors. *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, Boston, MA, Sep 2013.

D. R. Morrison. Patricia&mdash;practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

C. Okasaki and A. Gill. Fast mergeable integer maps. In G. Morrisett, editor, *ML'98: Proc. of the 1998 ACM SIGPLAN workshop on ML*, pages 77–86, Sept. 1998.

M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proc. of the 6th International Workshop on Automation of Software Test, AST 2011*, pages 91–97, 2011.

L. Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In W. Swierstra, editor, *Proc. of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64, 2014.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword Reynolds [1998].

J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

I. Sergey. Experience report: growing and shrinking polygons for random testing of computational geometry algorithms. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proc. of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 193–199, 2016.

V. St-Amour and N. Toronto. Experience report: Applying random testing to a base type environment. In Morrisett and Uustalu, pages 351–356.