École polytechnique fédérale de Lausanne

Optionnal Semester Project

Verified double-hashing hash map

Martin Vassor

supervised by
G. Candea, Pr.
K. Argyraki, Pr.
A. Zaostrovnykh, Ph. D. Student

January 11, 2017

1 Introduction

2 Implementation

3 Verification

3.1 Provided proof

3.1.1 Requirement (R)

Figure 1 present the relevant part of the proof of the original loop in `find_key`. The last statement (`no_key_found(ks, k)`) requires the property `not_my_key(k)` to be verified for all current keys in the mapping, ensured by the `up_to(nat_of_int(length(ks)), ...(not_my_key)(k)...)` statement.

`up_to(nat, prop)` verifies that `prop` is ensured for all `i` below `nat`:
`up_to(0, prop) = true`
`up_to(n, prop) = prop(n-1) && up_to(n-1, prop)`

This `up_to` statement is proved by the *for*-loop invariant: at each round, `not_my_key(k, nth(index, ks))` is ensured, either because the cell is empty (`no_busy_no_key` lemma), either because the hash does not match (`no_hash_no_key` lemma), either because the key does not match (hence inferred by Verifast).

Finally, the *for*-loop only proves that the `up_to` statement holds when starting from `index = start` and looping. The lemma `by_loop_for_all` prove that this loop access is equivalent to a continuous access from `0` to `length`.

3.1.2 Impact of the modifications

TODO

3.2 The `stripe_l_fp` fixpoint

3.2.1 Definition

First, a fixpoint is defined, which returns the index to be updated after n iterations with an offset of *step*, starting from *start* with capacity *capa*.

A lemma ensuring that `stripe` $= start + n * step \% capa$ is proved.

Definition 1: stripe(int start, int step, nat n, int capa)

```
fixpoint int stripe(int start, int step, nat n, int capa) {
  switch(n) {
    case zero: return start;
    case succ(m): return
      (stripe(start, step, m, capa) + step) % capa;
  }
}
```

The `stripe_l_fp` fixpoint builds a `list<option<nat>>` given a starting point, an offset, a number of accesses and a capacity. The base case of this fixpoint is to generate a `list` containing only `nones` (fixpoint `gen_none`), if `zero` accesses are performed. The recursive case is to update the $start + n * offset\%capa$ cell, using the above `stripe` fixpoint.

Definition 2: stripe_l_fp(int start, int step, nat bound, int capa)

```
fixpoint list<option<nat> > stripe_l_fp(int start,
  int step, nat n, int capa)
{
  switch(n) {
    case zero: return gen_none(nat_of_int(capa));
    case succ(m): return update(stripe(start, step, n, capa),
      some(n), stripe_l_fp(start, step, m, capa));
  }
}
```

Exampl 1: `stripe_l_fp(0, 2, 5, 7)`

Calling `stripe_l_fp(0, 2, 5, 7)` produces the following list. Notice that the base case returns a list containing only `nones`, not a list containing a `some(0)`.

`none`
`some(4)`
`some(1)`
`some(5)`
`some(2)`
`none`
`some(3)`

3.2.2 Properties

The main property required is that the number of cell containing `some(i)` (for any `i`) is equal to the number of steps done. This property will later be used to ensures that all cells are eventually reached (see Subsection 2.2.4). The function that count the number of such cells is named `count_some(list<option<nat>> list)`.

There are also other properties which are used internally to prove the `count_some` property. The main lemma is named `stripe_l`.

Definition 3: Prototype of `stripe_l`

```
lemma list<option<nat> > stripe_l(int start, int step, nat n, int capa)
requires 0 <= start &*& start < capa &*& step > 0 &*& int_of_nat(n) <= capa &*& coprin
ensures count_some(result) == n
&*& length(result) == capa
&*& true == up_to(nat_of_int(capa), (list_contains_stripes)(result, start, step))
&*& true == up_to(nat_of_int(capa), (lst_opt_less_than_n)(result, n))
&*& true == forall(result, opt_not_zero)
&*& result == stripe_l_fp(start, step, n, capa)
&*& coprime(step, capa);
```

3.2.3 Proof of `stripe_l`

The proof of these properies relies on the fact that the same cell is not updated twice. Once this is ensured, the construction of the fixpoint ensures the validity of the properties.

Algorithm 1 shows the main steps of the proof. In the base case, all trivially holds. In the inductive case, if the `stripe(start, step, n, capa)`-th cell (i.e. the one hit at n-th iteration) already contains `some(i)`, the `list_contains_stripes` property ensures that `stripe(start, step, i, capa)`-th cell is the one we are hitting. Hence, $start + step \times i\%capa = start + step \times n\%capa$, with $n - i < capa$. Then the Chinese remainder theorem leads to a contradiction.

3.2.4 From `stripe` fixpoint to `R`

3.3 Proof of the *Chinese remainder theorem*

3.3.1 Properties

3.3.2 Computation of *gcd*

3.3.3 *gcd* properties

3.3.4 Proof by contradiction

4 Conclusion

4.1 Validity of benchmark

4.2 Forthcoming work

Input: int start, int step, nat n, int capa

```
switch n do
  case zero
    // All hold by construction
  end
  case succ(m)
    // Recursive call
    list lst ⟵ stripe_l(start, step, m, capa)
    // Now, we want to update the stripe(start, step, n, capa)-th to
       some(n)
    // Proof by contradiction that the cell contains none
    switch nth(stripe(start, step, n, capa), lst do
      case some(i)
        assert start + i × step%capa = start + n × step%capa;
        assert (n − i) × step%capa = 0;
        assert n − i < capa;
        // The chinese remainder theorem applies and shows a
           contradiction
        chinese_remainder_theorem(step, capa, (n − i) × step);
      end
      case none
      end
    endsw
    // We now that the stripe(start, step, n, capa)-th cell contains
       a none, which we update to some(n), so the properties hold
       for the updated list.
    return update(stripe(start, step, n, capa), some(n), lst)
  end
endsw
```

Algorithm 1: Proof of stripe_l

The `update(index, elem, list)` fixpoint returns `list` with the `index`-th element updated to `elem`.

```
int i = 0;
for (; i < capacity; ++i)
/*@ invariant ... &*&
  true == up_to(nat_of_int(i),
    (byLoopNthProp)(ks, (not_my_key)(k),
      capacity, start));
@*/
//@ decreases capacity - i;
{
  int index = loop(start + i, capacity);
  int bb = busybits[index];
  int kh = k_hashes[index];
  void* kp = keyps[index];
  if (bb != 0 && kh == key_hash) {
    if (eq(kp, keyp)) {
      //@ hmap_find_this_key(hm, index, k);
      return index;
    }
  } else {
    //@ if (bb != 0) no_hash_no_key(ks, khs, k, index, hsh);
    //@ if (bb == 0) no_bb_no_key(ks, bbs, index);
  }
  //@ assert(true == not_my_key(k, nth(index, ks)));
}
/*@ by_loop_for_all(ks, (not_my_key)(k),
  start, capacity, nat_of_int(capacity));
@*/
/*@ assert true == up_to(nat_of_int(length(ks)),
  (nthProp)(ks, (not_my_key)(k)));
@*/
//@ no_key_found(ks, k);
```

Figure 1: Original *for*-loop for searching a key