The Boyer-Moore Majority-Vote Algorithm.

The problem we use to illustrate algorithm development in Martin-Löf's theory of types is called the majority-vote problem. It may briefly be described as determining whether or not one of the candidates in a ballot has received a majority of the votes. The solution on which our development is based is described in [MG] and is attributed by them to R. Boyer and J.S. Moore.

Let us suppose that the number of votes cast in a ballot is n and that the votes are recorded in an array a of length n. To be completely formal we further suppose that the candidates are drawn from the (non-empty) type A. Equality on A is necessarily decidable. Thus from now on we work within the following context.

$|[\quad A \in U_1$
$; \quad .eq. \in \forall(A, a, b.(a = b) \vee \neg(a = b))$ % We write $a.eq.b$
$; \quad n \in N$
$; \quad a \in Set(N, i.i < n) \longrightarrow A$
$; \quad x \in A$
$\triangleright$

In most ballots there are few candidates and many voters. Thus an obvious solution is to create a pigeon hole for each candidate and to put votes one-by-one into the pigeon holes. When all votes have been counted the number of votes of the candidate receiving most votes can be compared with n div 2. This means, however, that we must either assume known the number of candidates, allow for n candidates, or perform a preliminary analysis to determine the exact number of candidates. We obtain a much more elegant solution by doing none of these and abandoning the pigeon-hole solution altogether.

The specification in type theory of the program we require is the following theorem.

(0) $$Set(A, x.majority(x)) \vee \neg Set(A, x.majority(x))$$

where

$$majority(x) = \mathrm{N}(i : 0 \leqslant i < n : ai = x) > n \text{ div } 2.$$

Note that (0) is trivially true in classical mathematics; in constructive mathematics it is only true if one can provide a proof of either the proposition $Set(A, x.majority(x))$ — i.e. exhibit a candidate receiving a majority of votes — or the proposition $\neg Set(A, x.majority(x))$. Note also that an object in the right summand of (0) carries no computational content. What is significant is that the specification is deterministic: any two objects that achieve the specification must be equal.

In searching problems such as this a common strategy is to replace a proposition that may or may not be satisfiable by one that is always satisfiable but in such a way that a simple test on a satisfying instance determines whether the original proposition is satisfiable. This, for example, is the strategy adopted when a sentinel is added to the end of an array during a linear search for an element x. In this case we recognise that an easily solved problem is that of determining whether or not a given candidate x occurs a majority of times in a. This problem has specification:

(1) $$\forall(A, x.majority(x) \vee \neg majority(x).)$$

We leave it as an exercise for the reader to construct an object of (1).

Our solution to the majority-vote problem is based on combining a solution to (1) with a solution to the following:

(2) $$Set(A, x.pm(n, x)),$$

where the predicate *pm* has yet to be defined. (The parameter n occurs in *pm* in anticipation of later developments.)

Comparing (2) with (0) immediately suggests a definition of *pm*.

(3) $$pm(n, x) = majority(x) \vee \neg Set(A, x.majority(x)).$$

Of course a pair of objects of types (1) and (2) is not the same as an object of (0). However such an object can be easily recovered. Specifically, the function

$$\lambda q.\mathbf{split}(q, (f, p).\mathbf{when}(fp, a.\mathbf{inl}\ p, b.\mathbf{inr}\ (\lambda x.x))$$

is of type (1) $\wedge$ (2) $\Rightarrow$ (0). This can be seen by the following derivation (which the reader may choose to skip).

```
                % The abbreviation S is used throughout for          %
                  Set(A, x.majority(x)) ∨ ¬Set(A, x.majority(x))
0.0       ||[   f ∈ ∀(A, x.majority(x) ∨ ¬majority(x))
0.1       ;     p ∈ Set(A, x.pm(n, x))
          ▷       % pm(n, x) ⇒ (¬majority(x) ⇒ ¬Set(A, x.majority(x)),exercise 1.13 (b) %
0.2               p ∈ Set(A, x.¬majority(x) ⇒ ¬Set(A, x.majority(x)))
0.3.0             ||[   x ∈ A
0.3.1             ;     g ∈ ¬majority(x) ⇒ ¬Set(A, x.majority(x))
                  ▷       % 0.0,0.3.0, ∀-elim %
0.3.2                     fx ∈ majority(x) ∨ ¬majority(x)
0.3.3.0                   ||[   a ∈ majority(x)
                          ▷       % 0.3.0, 0.3.3.0, Subtype-intro, inl-intro %
0.3.3.1                         inl x ∈ S
                          ]||
0.3.4.0                   ||[   b ∈ ¬majority(x)
                          ▷       % 0.3.1,0.3.4.0, ⇒-elim %
0.3.4.1                         gb ∈ ¬Set(A, x.majority(x))
                                  % 0.3.4.1, example 2.1, inr-intro %
0.3.4.2                         inr(λx.x) ∈ S
                          ]||
                             % 0.3.2,0.3.3,0.3.4, ∨-elim %
0.3.5                     when(fx, a.inl x, b.inr(λx.x)) ∈ S
                  ]||
                     % 0.2,0.3, Subtype-elim %
0.4               when(fp, a.inl p, b.inr(λx.x)) ∈ S
          ]||
```

The identifier "*pm*" has been chosen as an abbreviation for "possible-majority candidate". From (3) we observe that an object $x \in Set(A, x.pm(n, x))$ satisfies the property

(4) $$\neg majority(x) \Rightarrow \neg Set(A, x.majority(x))$$

(since the right side of (3) formally implies (4)) Because a candidate obtaining a majority of votes is always unique, if one exists, (4) may be read as the statement that x excludes all other candidates from being in the majority.

We choose to prove (2) by elimination on n (i.e. by induction over the natural numbers). The basis is trivial since no candidate can occur a majority of times among 0 votes: thus we can straightforwardly exhibit an object of the right summand of (3) and any object will do as our possible-majority candidate. Problems occur when we try to perform the induction step. Suppose that $x \in Set(A, x.pm(k, x))$ for some $k \in \mathbb{N}$. How does one construct an object $y \in Set(A, x.pm(k+1, x))$? It is clear that more information is needed about the object x — we must strengthen our induction hypothesis.

In programming terms our aim is simply to construct a loop of the form **for** $i := 1$ **to** n **do** that exhibits a possible-majority candidate at each iteration. The notion of inductive hypothesis corresponds directly to the notion of invariant property. Strengthening the inductive hypothesis corresponds to introducing additional auxilliary variables into the computation.

Too strong a hypothesis would be the conjunction of (0) and

$$\forall(A, x.majority(x) \vee \forall(x, \neg majority(x)) \Rightarrow pm(n, x))$$

since it defeats the purpose of introducing the predicate pm. (Such a hypothesis states that x is a possible-majority candidate if either it is a majority candidate or no value is a majority candidate. It is a hypothesis likely to be proposed by a mathematician with no regard for the computational efficiency of the proof.) Instead we wish to strengthen the induction hypothesis as little as possible.

Another hypothesis we might consider is that along with the possible majority candidate is known its number of occurrences in the array segment. This is both too strong and too weak. It is too weak to stand alone as an inductive hypothesis. It is too strong because if we do try to prove it inductively we are obliged to consider a hypothesis in which the number of occurrences of every candidate is known, i.e. we have to revert to the pigeon-hole method.

The hypothesis we actually make is that not only is there a possible-majority candidate x but there is also an "estimate" of its number of occurrences in the array segment.

The information that the estimate must convey is when to discard one value and replace it by another. Suppose x is a possible-majority candidate for the first k votes. Then it is necessary to discard x as a possible-majority value for the first $(k+1)$ votes if the number of occurrences of x among these votes does not guarantee that no other value is a possible-majority value. This will be so when the number of occurrences of x is at most $(k+1)$ div 2. This suggests that the estimate we maintain is an upper bound on the number of occurrences of x. Denoting the estimate by e we require:

$$no\text{-}of\text{-}occurrences(k, x) < e \tag{5.1}$$

(where $no\text{-}of\text{-}occurrences(k, x) = \mathrm{N}(i : 0 \leqslant i < k : ai = x)$).

But this property alone is insufficient. We need to know that e is not a gross overestimate of the number of occurrences of x (the value $e = k$ satisfies (5.1)). The value e must also

represent some limit on the number of occurrences of other candidates which precludes their being majority values. We propose therefore that e should also have the property:

$$\forall(A, y.(y = x) \vee no{-}of{-}occurrences(k, y) \leq k - e). \tag{5.2}$$

The property (5.2) will imply the property $pm(k, x)$ if we also add the requirement:

$$k \leq 2 * e. \tag{5.3}$$

In summary, the property we require to prove is

$$Set(A \times N, (x, e).ind{-}hypo(n, x, e)) \tag{6}$$

where $ind{-}hypo(k, x, e)$ is the conjunction of properties (5.1), (5.2) and (5.3).

We now have two tasks. The first is to prove (6) by induction. An object of (6) will therefore take the form $\mathbb{N}elim(n, basis, (m, h)induction{-}step)$. The second task is to verify that the conjunction of properties (5) is indeed a stronger property than $pm(k, x)$. This is needed in order to show that the function **fst** maps an object of type (6) into the required object of type (2).

The second task is a problem of integer arithmetic and is one that we do not tackle here. We assume therefore that (the reader will verify for himself that)

$$\mathbf{fst} \in \forall((6), (x, e).pm(n, x)).$$

Let us now turn to the inductive proof of (6). The basis is a trivial problem of integer arithmetic since we can exhibit an arbitrary candidate, x say, as possible-majority candidate and 0 as the estimate of its number of occurrences.

For the induction step we assume that x and e satisfy properties (5) and show how to construct new values x' and e' satisfying $(5)[k := k+1]$. (To be completely formal we should assume that z, say, is an object of (6) and then split z into its components x and e.) Consider now the effect of the inclusion of the kth vote, ak, in the votes cast.

If $ak = x$ the estimate e of its occurrences increases by one and x is retained as a possible-majority value. That is $(x, e + 1) \in (6)[k := k + 1]$.

If, however, $ak \neq x$ the estimate e of x's occurrences remains constant and the situation is more complicated. One possibility is that $k = 2 * e$ and hence $k + 1 > 2 * e$. This clearly indicates that x is not a possible-majority value and there is the possiblility that some other value occurs a majority of times. The only value this could be is ak since by the induction hypothesis no value y different from x is a majority value in the first k votes and ak is the only value whose number of occurrences has increased in the process of extending the array segment. An upper bound on the number of occurences of ak is $k - e + 1$ since it occurs, by (5.2), at most $k - e$ times among the first k votes and, trivially, once more among the first $(k + 1)$ votes. Thus if $ak \neq x$ and $k = 2 * e$ the new possible majority value is $x' = ak$ and the new estimate is $e' = k - e + 1$ (which we observe equals $e + 1$).

The final possiblility is that although $ak \neq x$ the bound $k + 1 \leq 2 * e$ remains true. In this case $k + 1 - e$ is exactly one more than than $k - e$ and the number of occurrences of

each element y in the first $(k+1)$ votes is at most one more than its number of occurrences is the first k votes. Thus $(5)[k := k+1]$ is true of $x' = x$ and $e' = e$.

Summarising the inductive step takes the form:

if $ak = x \longrightarrow (x, e+1)$
[] $(ak \neq x) \wedge (2 * e = k) \longrightarrow (ak, e+1)$
[] $(ak \neq x) \wedge (2 * e > k) \longrightarrow (x, e)$
fi

and the complete (including a solution to (1)) takes the form:

```
split(
    INelim(n
        ,< x0, 0 >
        ,(k,p).split(p
                ,(x,e).if  ak = x ⟶< x, e+1 >
                        [] (ak ≠ x) ∧ (2 * e = k) ⟶< ak, e+1 >
                        [] (ak ≠ x) ∧ (2 * e > k) ⟶< x, e >
                        fi
                )
        )
    ,(x,e).
        (λx.when(
                INelim(n, 0, (k,c)if ak = x ⟶ c+1 [] ak ≠ x ⟶ c fi)
                    .gt.n div 2
                ,a.inl x
                ,b.inr b
                )
        x
    )
||
```

Reference

[MG] J.Misra and D.Gries
"Finding repeated elements," Science of Computer Programming, *2*, 143-152 (1982).