A HOL model of the Hare-Clark voting system
Version 2.04

Michael Norrish

8 June 2004

Abstract

This document gives a formal, logical presentation of the Hare-Clark voting system used in the Australian Capital Territory, as specified in Schedule 4 of the Electoral Act 1992 (R12). The presentation is one that can be mechanically manipulated by the HOL proof assistant.

Contents

Chapter 1

Introduction

1.1 Notation

This document is a "literate program", which means that it is a combination of expository text, and actual HOL source code. This source code is always found in specially marked-up chunks. The following chunk, ⟨*Sample* 4a⟩ illustrates:

4a ⟨*Sample* 4a⟩≡
```
HOL code might apper here.

To elide details, code can also be
put into other chunks, and referred
to from here.  Thus:
  ⟨Sample Details 4b⟩
```

The reference to ⟨*Sample Details* 4b⟩ inside the chunk above should be taken to mean that the text of the chunk labelled ⟨*Sample Details* 4b⟩ is actually included in ⟨*Sample* 4a⟩ but has been elided for reasons of exposition. Those details are provided in a chunk of their own:

4b ⟨*Sample Details* 4b⟩≡ (4a) 5a▷
```
Material separated out from the
higher level chunk.
```

Note that all chunks are accompanied by references that appear in left margins to make it easy to find cross-referenced chunks. The parenthesised numbers on the right hand side specify which chunks, if any, this chunk is referred to by. The references identify the page-number of the chunk, possibly followed by additional roman letters if more than one chunk appears on that page.

If multiple chunks appear with the same name, then these should be understood as if the material within them is concatenated together. Thus, it is possible to extend the subsidiary chunk ⟨*Sample Details* 4b⟩:

5a ⟨*Sample Details* 4b⟩+≡ (4a) ◁4b

```
More information
```

The numbers with triangles that appear to the right of such chunks indicate that they are part of a continued chunk.

1.2 The HOL Formalisation

The following "chunk", which assembles the HOL source code, should now make sense: the theory consists of sections with names such as ⟨*Preliminaries* 56a⟩, ⟨*Types* 8a⟩ and ⟨*Algorithm Definition* 27⟩ that will be present in the output source in that order. This doesn't mean that the presentation in this document follows that order. The ⟨*Preliminaries* 56a⟩ section, for example, is less fascinating than the other material, and is relegated to an appendix.

5b ⟨*HOL Theory of the Electoral Act 1992* 5b⟩≡

⟨*Preliminaries* 56a⟩
⟨*Types* 8a⟩
⟨*Algorithm States* 19⟩
⟨*Algorithm Definition* 27⟩
⟨*Theorems* 40⟩
⟨*Closing* 64b⟩

Chapter 2

The Algorithm Introduced

The basic outline of the algorithm is given in Figure 2.1, where each of the boxes in the chart corresponds to a clause of Part 4.2 of the Act. The logical specification of the algorithm will reflect this structure, with one HOL function for each box, or phase of the algorithm. Each such function will have the name `clause`$\langle n \rangle$ for appropriate values of n.

The HOL model of this algorithm will take an "algorithm state" and pass it through each of the flow-chart boxes in Figure 2.1. So, the initial state will have a number of continuing candidates, no successful candidates and a pile of unallocated votes. The state will evolve as it passes through the various phases, with candidates becoming successful and others being excluded.

2.1 Counts, or the algorithm's "clock"

The algorithm proceeds in phases that the Act calls "counts". To a first approximation, each loop around the graph in Figure 2.1 is a count. Unfortunately, this simple picture is inaccurate because Clause 9 can cause multiple phases of vote transfers, and each of these phases consists of a single "count". This doesn't cause any significant problems.

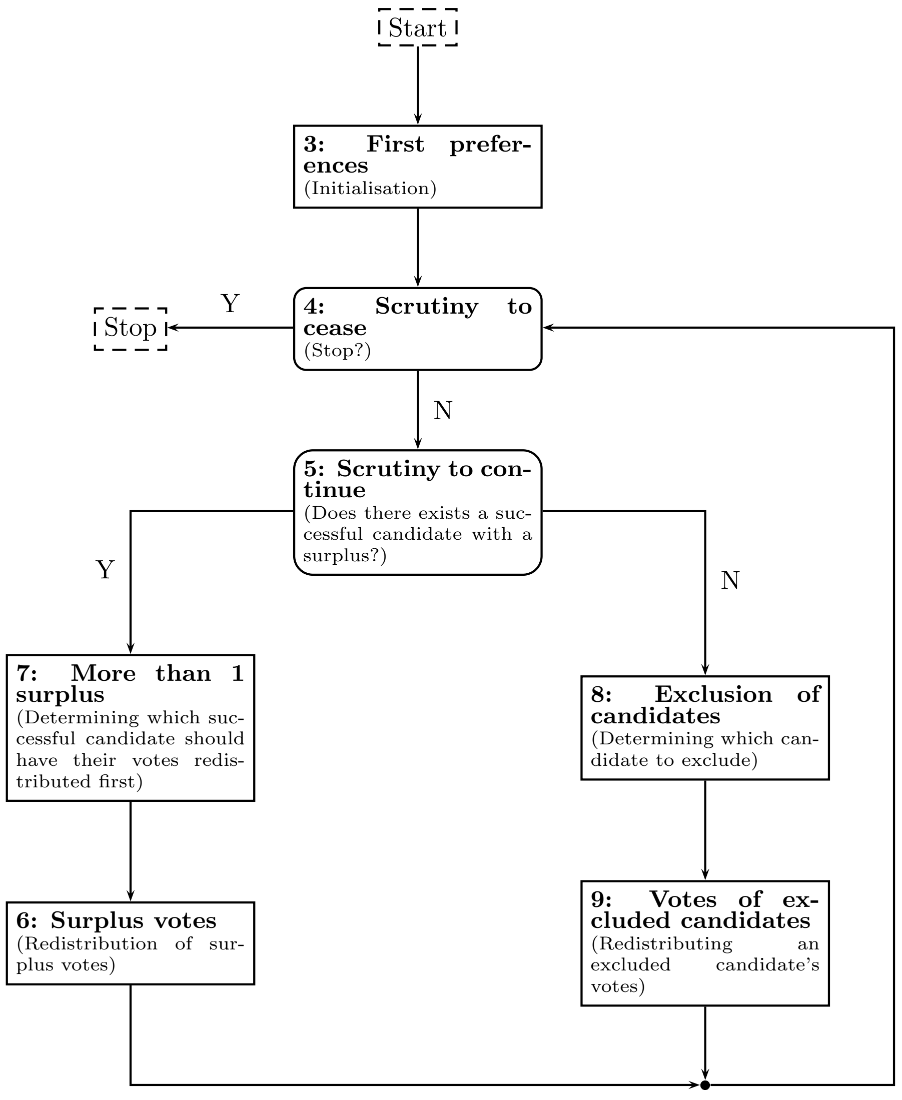
Start
3: First preferences
(Initialisation)
4: Scrutiny to cease
(Stop?)
Y
Stop
N
5: Scrutiny to continue
(Does there exists a successful candidate with a surplus?)
Y
N
7: More than 1 surplus
(Determining which successful candidate should have their votes redistributed first)
8: Exclusion of candidates
(Determining which candidate to exclude)
6: Surplus votes
(Redistribution of surplus votes)
9: Votes of excluded candidates
(Redistributing an excluded candidate's votes)

Figure 2.1: A flow-chart for the Hare-Clark algorithm. Bold-face numbers are those of the clauses in Part 4.2, accompanied by those clauses' titles.

Chapter 3

Basic Types

Before a formal description of the Hare-Clark algorithm is possible, all of the "objects" it manipulates need to be described. Types provide these descriptions. In this chapter, the simplest types are described, starting with candidates, and moving up to "vote piles". In Chapter 4, the type of the algorithm states is introduced.

8a ⟨*Types* 8a⟩≡ (5b)
```
  ⟨Candidates 8b⟩
  ⟨Ballot Papers 9a⟩
  ⟨Candidate Information 13a⟩
  ⟨Vote Piles 15b⟩
```

3.1 Candidates

It is not necessary to commit to a particular representation for candidates as the algorithm doesn't require anything more than being able to compare two candidates for equality. The following establishes such a type:[1]

8b ⟨*Candidates* 8b⟩≡ (8a)
```
  val _ = new_type("candidate", 0)
```
Defines:
  `candidate`, used in chunks 9a, 11b, 15, 16, 18, 20, 22b, 33, 35, 36, 38, 39, and 47.

There are no interesting functions possible a type such as this. In order to make the specification executable, the candidate type may be made an abbreviation for the type of strings.

[1]The zero in the definition means that the type is a nullary type constructor, alternatively, a type constant.

3.2 Ballot Papers

A ballot is represented as a list of candidates, where the first candidate in the list is the voter's first choice, where the second candidate in the list is the voter's second choice, and so on.

9a ⟨*Ballot Papers* 9a⟩≡ (8a) 9b ▷
```
  val _ = type_abbrev("ballot", ``:candidate list``)
```
Defines:
  `ballot`, used in chunks 9b, 10a, and 47.
Uses `candidate` 8b.

Ballot papers coupled with transfer values will be known as *transferable ballot papers* (`tbp`):

9b ⟨*Ballot Papers* 9a⟩+≡ (8a) ◁ 9a 9c ▷
```
  val _ = type_abbrev("tbp", ``:ballot # real``)
```
Defines:
  `tbp`, used in chunks 10–14, 16a, 18, 20, 21b, and 33.
Uses `ballot` 9a.

There are a number of important functions to be defined on the type of ballot papers.

9c ⟨*Ballot Papers* 9a⟩+≡ (8a) ◁ 9b 12b ▷
  ⟨*Ballot Paper Functions* 10a⟩

3.2.1 Well-formed ballots

Not all possible lists of candidates correspond to real ballots. In particular, a list of candidates that includes repeated candidates is not valid. Also, a ballot must list at least one candidate. Further, it is assumed that the provisions of Part 4.1, clause 2 have been applied to normalise ballot papers, so that

- ballot papers where "the same number has been marked in 2 or more candidate squares", have those numbers and any greater number removed (Part 4.1, clause 2(2)); and
- ballot papers where "a number is missing from the series of consecutive whole numbers marked in the candidate squares", have all numbers greater than the missing number removed (Part 4.1, clause 2(3)).

Because voters do not have to rank all possible candidates, and because of the above "normalisation provisions", ballots may be lists of different lengths. Further, votes must be for valid candidates (where validity is given in a set of candidates `ok_cands`, passed as a parameter). The following defines a new predicate to test whether or not a ballot is *well-formed*:

10a ⟨*Ballot Paper Functions* 10a⟩≡ (9c) 10b▷

```
val wf_ballot_def = Define`
  wf_ballot  ok_cands (b : ballot) =
    ALL_DISTINCT b /\ ~(b = []) /\ set(b) SUBSET ok_cands
`;
```

Defines:
  `wf_ballot`, used in chunks 10b, 42a, and 47.
Uses `ballot` 9a and `set` 57a.

Transferable ballot papers also need their own well-formedness predicate. The ballot component must itself be well-formed, and the transfer value must be positive and rational.

10b ⟨*Ballot Paper Functions* 10a⟩+≡ (9c) ◁10a 11a▷

```
val wf_tbp_def = Define`
  wf_tbp ok_cands ((b, tv) : tbp) =
    wf_ballot ok_cands b /\ 0 < tv /\ ?p q. tv = &p / &q
`;
```

Defines:
  `wf_tbp`, used in chunks 15a, 25, 45b, and 47.
Uses `tbp` 9b and `wf_ballot` 10a.

3.2.2 What is the value of a bag of transferrable ballots?

Bags of transferable ballot papers may need to be summed. This calculation is done by the function `sum_vote_bag`:

11a ⟨*Ballot Paper Functions* 10a⟩+≡ (9c) ◁10b 11b▷

```
  val sum_vote_bag_def = Define‘
    sum_vote_bag (bag : tbp bag) : real =
      ITBAG ((+) : real -> real -> real) (image SND bag) 0
  ‘;
```

Defines:
  `sum_vote_bag`, used in chunks 15a and 23c.
Uses `image` 57a and `tbp` 9b.

3.2.3 Which candidates are in a bag of transferrable ballots?

This function returns (the set of) candidates mentioned in a bag of transferrable ballots. This is done by calculating the image of the bag under `set o FST`. The `FST` function strips the transfer value of each ballot, and then the `set` function turns a list of candidates into a set of candidates. The result is a bag of sets of candidates. The call to `ITBAG` then calculates the union of all these sets.

11b ⟨*Ballot Paper Functions* 10a⟩+≡ (9c) ◁11a 12a▷

```
  val tbp_bag_candidates_def = Define‘
    tbp_bag_candidates (tb : tbp bag) : candidate set =
      ITBAG (UNION) (image (set o FST) tb) {}
  ‘;
```

Defines:
  `tbp_bag_candidates`, used in chunks 14b and 22a.
Uses `candidate` 8b, `image` 57a, `set` 57a, and `tbp` 9b.

3.2.4 Is a ballot paper for a candidate?

The predicate `ballot_is_for` takes three parameters: `ignore`, a set of candidates to ignore; `c`, the candidate whose ballot this might be; and `b`, the (transferable) ballot to be checked. If, ignoring those candidates in the set `ignore`, the ballot `b` is "for" candidate `c` (i.e., there is at least one candidate remaining on the ballot, and the first such candidate is `c`), then this predicate is true, otherwise false. The ignored candidates correspond to those that have already been excluded from consideration for one reason or another. They may have been dead to start with, they may have already become successful candidates, or they may have been excluded.

In more detail, the `filter` function in the definition filters a list of candidates so that only those satisfying its first argument remain. The predicate in this case is `\c'. ~(c' IN ignore)`, which is true of those candidates that don't appear in the `ignore` set.

12a ⟨*Ballot Paper Functions* 10a⟩+≡ (9c) ◁11b

```
val ballot_is_for_def = Define`
  ballot_is_for ignore c ((b, tv) : tbp) =
    let b' = filter (\c'. ~(c' IN ignore)) b
    in
        ~(b' = []) /\ (HD b' = c)
`;
```

Defines:
`ballot_is_for`, used in chunks 17, 33, 39, and 47.
Uses `filter` 57a and `tbp` 9b.

Further properties of ballot papers are stated and proved in an appendix:

12b ⟨*Ballot Papers* 9a⟩+≡ (8a) ◁9c
⟨*Properties of ballot papers* 45b⟩

3.3 Candidate Information

Each candidate in the vote-counting system is associated with three pieces of information:

- a natural number (the vote value of the candidate's votes);
- a bag of ballot papers; and
- another bag, representing the last batch of votes transferred to this candidate.

The number of votes allotted to a candidate is not simply the sum of the votes' transfer values, nor even this sum rounded down. Instead, when a packet of papers is transferred to a candidate, the number of votes these represent is rounded down, before being added to the candidate's count. This is as per the definition of *count votes*. Therefore, in addition to each pile of votes, the system must track the number of votes this pile represents. Finally, the last packet of votes transferred to a candidate is significant when a candidate becomes successful. It is only this packet of votes that may be liable for redistribution.

It seems simpler to avoid having the same votes recorded twice, so the total votes for a candidate will be the union of the second and third components. Another option would be to have transferred votes move into both the second and third components simultaneously. This type is introduced below as a `cinfo` (standing for "candidate information"):

13a ⟨*Candidate Information* 13a⟩≡ (8a) 13b▷

```
val _ = Hol_datatype`
  cinfo = <| count : num ;
             old_votes : tbp bag ;
             latest_votes : tbp bag |>
`;
```

Defines:
  `cinfo`, used in chunks 14–16, 18, and 20.
Uses `tbp` 9b.

There are a number of functions to be defined on candidate information records.

13b ⟨*Candidate Information* 13a⟩+≡ (8a) ◁13a
  ⟨*Candidate Information Functions* 14a⟩

3.3.1 What are all the ballots in a `cinfo`?

Though a candidate information record needs to track the most recently transfered votes separately from the other votes a candidate has received, it is also frequently necessary to consider all of a candidate's votes together. The name of this function makes it clear that a bag of transferrable ballot papers is returned.

14a ⟨*Candidate Information Functions* 14a⟩≡ (13b) 14b ▷

```
val cinfo_votebag_def = Define‘
  cinfo_votebag (ci : cinfo) : tbp bag =
    BAG_UNION ci.old_votes ci.latest_votes
‘;
```

Defines:
`cinfo_votebag`, used in chunks 14–17 and 23c.
Uses `cinfo` 13a and `tbp` 9b.

3.3.2 What are the candidates in a `cinfo`?

This function returns (the set of) all of the candidates mentioned in a candidate information record:

14b ⟨*Candidate Information Functions* 14a⟩+≡ (13b) ◁ 14a 15a ▷

```
val cinfo_candidates_def = Define‘
  cinfo_candidates pt = tbp_bag_candidates (cinfo_votebag pt)
‘;
```

Defines:
`cinfo_candidates`, used in chunk 16b.
Uses `cinfo_votebag` 14a and `tbp_bag_candidates` 11b.

3.3.3 Well-formedness for `cinfo` values

A candidate information record is well-formed if

- the two bags of ballots stored for that candidate are finite;
- at least one of the bags is non-empty;
- the stored count for the candidate is no greater than the sum of the transfer values on all the stored ballots; and
- each stored ballot is well-formed.

A ballot is only well-formed with respect to a set of valid candidates, so this set is given as a parameter to the `wf_cinfo` function defined below.

15a ⟨*Candidate Information Functions* 14a⟩+≡ (13b) ◁14b

```
val wf_cinfo_def = Define`
  wf_cinfo okcands ci =
    finite (cinfo_votebag ci) /\ ~(cinfo_votebag ci = {||}) /\
    &ci.count <= sum_vote_bag (cinfo_votebag ci) /\
    !b. BAG_IN b (cinfo_votebag ci) ==> wf_tbp okcands b
`;
```

Defines:
`wf_cinfo`, used in chunk 17.
Uses `cinfo_votebag` 14a, `finite` 57a, `sum_vote_bag` 11a, and `wf_tbp` 10b.

3.4 Vote Piles

A "vote pile" is a finite map from candidates to candidate information records (written in HOL as `:candidate |-> cinfo`). The notation with the arrow will generally be preferred to the abbreviation.

15b ⟨*Vote Piles* 15b⟩≡ (8a)

```
val _ = type_abbrev("vote_pile", ``:candidate |-> cinfo``)
⟨Vote Pile Functions 16a⟩
```

Uses `candidate` 8b and `cinfo` 13a.

There are a great many functions on vote piles as this type is the core component of the algorithm state.

3.4.1 What are a pile's ballot papers?

This function returns all of the ballot papers stored within a pile (i.e., a finite-map from candidates to pile-triples):

16a ⟨*Vote Pile Functions* 16a⟩≡ (15b) 16b▷

```
val pile_ballots_def = Define`
  pile_ballots (p : candidate |-> cinfo) : tbp bag =
    ranb (cinfo_votebag o_f p)
`;
```

Defines:
  `pile_ballots`, used in chunks 21b and 25.
Uses `candidate` 8b, `cinfo` 13a, `cinfo_votebag` 14a, `ranb` 60a, and `tbp` 9b.

3.4.2 What are the candidates in a pile?

Composing the `cinfo_candidates` function with `p` generates a finite map from candidates to set of candidates. (This is the map from candidates to the *sets* of candidates mentioned on their ballot papers.) Taking the range of this map returns a set of sets.

16b ⟨*Vote Pile Functions* 16a⟩+≡ (15b) ◁16a 17▷

```
val pile_candidates_def = Define`
  pile_candidates (p : candidate |-> cinfo) =
    BIGUNION (ran (cinfo_candidates o_f p))
`;
```

Defines:
  `pile_candidates`, used in chunks 22a and 42b.
Uses `candidate` 8b, `cinfo` 13a, and `cinfo_candidates` 14b.

3.4.3 Well-formedness for vote piles

A vote pile is well-formed if for every candidate in the domain of the pile each stored ballot in the candidate's record is actually for the candidate, and if each record is itself well-formed. This function is again parameterised by a set of acceptable candidates. In addition, determining who a ballot is for requires a knowledge of which candidates on a ballot should be ignored, so this must also be a parameter.

17 ⟨*Vote Pile Functions* 16a⟩+≡ (15b) ◁16b 18▷

```
val wf_pile_def = Define`
  wf_pile okcands ignores p =
    !c. c IN dom p ==>
        wf_cinfo okcands (p ' c) /\
        !b. b IN set (cinfo_votebag (p ' c)) ==>
            ballot_is_for ignores c b
`;
```

Defines:
  wf_pile, used in chunks 25, 47, and 55.
Uses ballot_is_for 12a, cinfo_votebag 14a, dom 57a, set 57a, and wf_cinfo 15a.

3.4.4 Transferring a bundle of votes

The transfer of bundles of votes is done in Clauses 6 and 9 (Sections 5.4 and 5.7), so it makes sense to specify a function that both implementations can use. The function `transfer_bundle` takes a bundle of votes to be transferred. All of the ballots are assumed to have the same transfer value, but they may be for different candidates. The `bfor` parameter is used to determine which candidate a ballot is for, and the `pile` parameter is the pile of votes which is to be updated as the votes are transferred.

18 ⟨*Vote Pile Functions* 16a⟩+≡ (15b) ◁17

```
val transfer_bundle_def = Define‘
  transfer_bundle (bundle : tbp bag)
                  (bfor : tbp -> candidate)
                  (pile : candidate |-> cinfo)
  =
    let brel (b1 : tbp) (b2 : tbp) = (bfor b1 = bfor b2) in
    let grouped_votes = partitionb brel bundle in
    let transfer_group gp pile =
         let (b, tv) = BAG_CHOICE gp in
         let b_is_for = bfor (b, tv) in
         let count = trunc (tv * &(card gp)) in
         let (old_count,old_ballots,old_lastballots) =
               if b_is_for IN dom pile then
                  let pt = pile ’ b_is_for in
                     (pt.count, pt.old_votes, pt.latest_votes)
               else
                  (0, {||}, {||})
         in
         let merged = BAG_UNION old_ballots old_lastballots
         in
             pile |+ (b_is_for,
                      <| count := old_count + count ;
                         old_votes := merged ;
                         latest_votes := gp |>)
    in
         ITSET transfer_group grouped_votes pile
‘;
```

Defines:
  `transfer_bundle`, used in chunks 33, 39, and 42b.
Uses `candidate` 8b, `card` 57a, `cinfo` 13a, `dom` 57a, `partitionb` 61a, and `tbp` 9b.

Chapter 4

Algorithm State

19 ⟨*Algorithm States* 19⟩≡ (5b)
  ⟨*Algorithm State Type* 20⟩
  ⟨*Functions over Algorithm States* 21b⟩
  ⟨*Algorithm State Well-formedness (Invariant)* 25⟩

The state manipulated by the algorithm must include

- A vote pile, keyed by the current first candidate of each ballot paper.
- A bag of votes corresponding to those votes that have been set aside. Votes can be set aside if they no longer mention any continuing candidates, or once a candidate is deemed successful and elected. In the latter case, all of that's candidate's votes, except the last packet of votes transferred to that candidate, have no further role to play, and mustn't be considered by further stages of the algorithm, even if they do mention other continuing candidates.
- A characterisation of the candidates not in the domain of `vote_pile` into three disjoint sets: `dead`, `excluded`, and `done_successes`. The dead set will not change as algorithm proceeds. The `excluded` candidates are those who have been excluded through the action of Clause 9. The `done_successes` candidates will be those candidates who are both successful, and have had any surplus votes redistributed. In other words, they will be candidates who do not need any further work done on them. The Act's category of *successful* candidates will be the union of those candidates with `vote_pile` who have sufficient votes with `done_successes`. The Act's category of *continuing* candidates will be the subset of the candidates with `vote_pile` who are not successful.
- The election's *quota*. The quota will not change as the algorithm proceeds.
- The number of candidates to elect, given as N in Schedule 4. This number is another that will not change as the algorithm proceeds.

Here follows the definition of a new HOL type, `state`, a record of seven fields, corresponding to the state discussed above.

20 ⟨*Algorithm State Type* 20⟩≡ (19) 21a▷
```
val _ = Hol_datatype`
  state = <| vote_pile      : candidate |-> cinfo ;
             set_aside       : tbp bag ;
             dead            : candidate set ;
             excluded        : candidate set ;
             done_successes : candidate set ;
             quota           : num ;
             num_to_elect    : num |>
`;
```
Defines:
`state`, used in chunks 21–24, 29–31, 37, and 39.
Uses `candidate` 8b, `cinfo` 13a, `set` 57a, and `tbp` 9b.

It is clear from Part 4.2, clauses 7 & 8, that the algorithm must have access to previous stages of the vote-counting. That is, the algorithm must have access to a list of previous states from previous iterations of the process. This will be done by providing a list of previous states as another argument to each function implementing the various stages of the algorithm. Also, each such function will return both a new state, and a potentially new history. In this sense, the full state being manipulated by the algorithm is actually a pairing of a state with another list of states, the latter being the history. A `state` coupled with another list of `state`s will be a `fullstate`:

21a ⟨*Algorithm State Type* 20⟩+≡ (19) ◁20

```
val _ = type_abbrev("fullstate", ``:state # state list``)
```

Defines:
  `fullstate`, used in chunks 28, 33, 35, 36, and 38.
Uses `state` 20.

4.1 Functions over Algorithm States

In this section, necessary auxiliary functions over ballot papers and algorithm states are described.

4.1.1 Which are all the ballot papers?

This function returns all of the ballot papers stored within the given state:

21b ⟨*Functions over Algorithm States* 21b⟩≡ (19) 22a▷

```
val state_ballots_def = Define`
  state_ballots (s : state) : tbp bag =
    BAG_UNION (pile_ballots s.vote_pile) s.set_aside
`;
```

Defines:
  `state_ballots`, never used.
Uses `pile_ballots` 16a, `state` 20, and `tbp` 9b.

4.1.2 Which are all the candidates?

This function returns the set of all the candidates mentioned in a state. As candidates only move from one set to another, and are never dropped, this set should stay the same through all of the algorithm's phases. It is possible that a valid candidate should have no first round votes at all. This possibility explains the complicated definition of `pile_candidates` below; it's not enough to simply take the domain of the `vote_pile` map as this set only includes those candidates who are at the top of a ballot paper.

22a ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁21b 22b▷

```
  val all_candidates_def = Define`
    all_candidates (s:state) =
        s.dead UNION s.excluded UNION
        pile_candidates s.vote_pile UNION
        s.done_successes UNION
        tbp_bag_candidates s.set_aside
  `;
```

Defines:
  `all_candidates`, used in chunks 23–25, 42c, 47, and 55.
Uses `pile_candidates` 16b, `state` 20, and `tbp_bag_candidates` 11b.

4.1.3 How many votes does a (continuing) candidate have?

This is easy to calculate: if the candidate is in the vote pile, return the `count` field of that candidate's information record. Otherwise the candidate has no votes.

22b ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁22a 23a▷

```
  val candidate_votes_def = Define`
    candidate_votes s (c : candidate) : num =
        if c IN dom(s.vote_pile) then (s.vote_pile ' c).count else 0
  `;
```

Defines:
  `candidate_votes`, used in chunks 23a, 29, 31, 35, 37, 42c, 47, and 55.
Uses `candidate` 8b and `dom` 57a.

4.1.4 What is the set of *successful* candidates?

Given an algorithm state, return the set of candidates that are successful, in the sense of Schedule 4.

23a ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁22b 23b▷

```
  val successful_candidates_def = Define‘
    successful_candidates s  =
      s.done_successes UNION
      (all_candidates(s) INTER {c | s.quota <= candidate_votes s c})
  ‘;
```

Defines:
  successful_candidates, used in chunks 23b, 25, 29–31, and 34.
Uses all_candidates 22a and candidate_votes 22b.

4.1.5 Which candidates can be ignored?

Given a state, this function returns the set of candidates that can now be ignored. An ignored candidate is one whose presence on a ballot is ignored when deciding where a ballot is to be transferred in a redistribution.

23b ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁23a 23c▷

```
  val ignored_candidates_def = Define‘
    ignored_candidates (s:state) =
      s.dead UNION s.excluded UNION successful_candidates(s)
  ‘;
```

Defines:
  ignored_candidates, used in chunks 24, 25, 33, 39, 47, and 55.
Uses state 20 and successful_candidates 23a.

4.1.6 What is the total value of remaining, "live" votes?

Given a set of candidates to ignore, sum the values of the ballot papers that have not been "set aside" (because they only mention candidates in the set to ignore). This sum is the "genuine" sum of all the ballots, computed by summing the ballots' transfer values.

23c ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁23b 24▷

```
  val total_live_votes_def = Define‘
    total_live_votes s =
        let total ci = sum_vote_bag (cinfo_votebag ci)
        in
            ITBAG (+) (FRANGEB (total o_f s.vote_pile)) 0
  ‘;
```

Defines:
  total_live_votes, used in chunks 25 and 47.
Uses cinfo_votebag 14a and sum_vote_bag 11a.

4.1.7 What is the set of *continuing* candidates?

Given an algorithm state, return the set of candidates that are continuing, in the sense of Schedule 4.

24 ⟨*Functions over Algorithm States* 21b⟩+≡ (19) ◁23c

```
val continuing_candidates_def = Define‘
  continuing_candidates (s:state) =
    all_candidates(s) DIFF ignored_candidates(s)
‘;
```

Defines:
`continuing_candidates`, used in chunks 25, 29, 30, and 37.
Uses `all_candidates` 22a, `ignored_candidates` 23b, and `state` 20.

4.1.8 Algorithm State Well-formedness

Not all states are well-formed. The predicate `wf_state` requires that:

- the `quota` is positive;
- the `vote_pile` is well-formed;
- the `set_aside` is of finite size;
- all of the votes in the `set_aside` are well-formed;
- the ballots in the `set_aside` pile do not overlap with those in the `vote_pile`;
- there are only finitely many votes;
- there are only finitely many successful candidates;
- the continuing candidates, the dead candidates, the excluded candidates and the successful candidates are mutually disjoint;
- the cardinality of `done_successes` is always no more than the `num_to_elect` field; and
- there are never too many "live votes": this constraint is expressed by treating every candidate in `done_successes` as equivalent to `quota` many votes, adding this number to the value of the remaining votes, and insisting that this be less than the `quota` multiplied by one more than `num_to_elect`.

25 ⟨*Algorithm State Well-formedness (Invariant)* 25⟩≡ (19)

```
val wf_state_def = Define`
  wf_state s =
    0 < s.quota /\

    wf_pile (all_candidates s) (ignored_candidates s) s.vote_pile /\

    finite s.set_aside /\
    (!b. BAG_IN b s.set_aside ==> wf_tbp (all_candidates s) b) /\
    disjoint (set s.set_aside) (set (pile_ballots s.vote_pile)) /\

    finite (successful_candidates s) /\
    card (successful_candidates s) <= s.num_to_elect /\
    disjoint_sets {| continuing_candidates(s); s.dead;
                     s.excluded; successful_candidates(s) |} /\
      total_live_votes s + &(s.quota * card (s.done_successes))
    <
      &(s.quota * (s.num_to_elect + 1))
`;
```

Defines:
  `wf_state`, used in chunk 42.
Uses `all_candidates` 22a, `card` 57a, `continuing_candidates` 24, `disjoint` 57a, `disjoint_sets` 58a, `finite` 57a, `ignored_candidates` 23b, `pile_ballots` 16a, `set` 57a, `successful_candidates` 23a, `total_live_votes` 23c, `wf_pile` 17, and `wf_tbp` 10b.

Chapter 5

The Algorithm Defined

3: First preferences: This phase of the algorithm must take a bag of votes, a set of candidates divided into dead and alive subsets, and the number of candidates to be elected. Given these inputs, the action of Clause 3 is to create an algorithm state. It must therefore be a function of type

$$\texttt{ballot bag} \times \texttt{num} \times \texttt{candidate set} \times \texttt{candidate set} \;\rightarrow\; \texttt{fullstate}$$

4: Scrutiny to cease: This phase must examine a state and determine whether or not the algorithm is to finish. It will be a function of type

$$\texttt{state} \rightarrow \texttt{bool}$$

where `bool` is the type of boolean values, true or false.

5: Scrutiny to continue: This phase must examine a state and determine whether or not there are any successful candidates with surplus votes. It is also a function of type

$$\texttt{state} \rightarrow \texttt{bool}$$

6: Surplus votes: This phase redistributes the surplus votes of a successful candidate. In addition to the current state, it will also take the selected candidate as an input, so its type will be

$$\texttt{fullstate} \times \texttt{candidate} \;\rightarrow\; \texttt{fullstate}$$

7: More than 1 surplus: This phase determines which successful candidate with surplus votes should have their surplus votes redistributed. Because circumstances may require the electoral commissioner to make a choice by lot, this function needs to allow for this possibility. The most straightforward way of modelling this is to have the function take a candidate as input, represeting the commissioner's choice, when needed. The constraint that the commissioner's choice must be one from the relevant set will be expressed elsewhere. The type will be

$$\texttt{fullstate} \times \texttt{candidate} \rightarrow \texttt{candidate}$$

In those situations, where the commissioner doesn't have to make a choice by lot, the extra parameter to this parameter will be ignored.

8: Exclusion of candidates: This phase determines which candidate to exclude in situations where there are no successful candidates with surpluses. Again, there are circumstances when the commissioner has to choose a candidate by lot, and this is represented by an extra `candidate` input to the function, whose type is

$$\texttt{fullstate} \times \texttt{candidate} \rightarrow \texttt{candidate}$$

9: Votes of excluded candidates: This phase redistributes the votes of an excluded candidate. As in the phase implementing clause 6, this function will take the excluded candidate as an additional input, so that its type will be

$$\texttt{fullstate} \times \texttt{candidate} \rightarrow \texttt{fullstate}$$

Components constructed in direct reflection of the flow-chart, combine to create the final algorithm:

27 ⟨*Algorithm Definition* 27⟩≡ (5b)
  ⟨*Clause 3: First preferences* 28⟩
  ⟨*Clause 4: Scrutiny to cease* 29⟩
  ⟨*Clause 5: Scrutiny to continue* 31⟩
  ⟨*Clause 6: Surplus votes* 33⟩
  ⟨*Clause 7: More than 1 surplus* 34⟩
  ⟨*Clause 8: Exclusion of candidates* 37⟩
  ⟨*Clause 9: Votes of excluded candidates* 39⟩

5.1 Clause 3: First preferences

The action of Clause 3 is simply one of creating an initial state. This is the HOL function `create_initial_state`. The calculation of the `quota` field uses the `DIV` function, which takes two natural numbers, and computes their quotient, rounding down. Thus, the model computes

$$\left\lfloor \frac{\mathrm{BP}}{\mathrm{N}+1} \right\rfloor + 1$$

where the Act calls for

$$\left\lfloor \frac{\mathrm{BP}}{\mathrm{N}+1} + 1 \right\rfloor$$

but it's clear that these two are the same.

3 First preferences

(1) For each ballot paper recording a first preference for a continuing candidate, 1 vote shall be allotted to the candidate.

(2) For subclause (1), a ballot paper on which a first preference for a candidate who died before polling day is recorded shall be taken to record a first preference for the candidate for whom the next available preference is recorded.

(3) After the allotment of votes under subclause (1), each continuing candidate's total votes shall be calculated and, if the votes equal or exceed the quota, the candidate is successful.

Figure 5.1: Part 4.2, clause 3

Conforming to the Act The provision of Clause 3(1) is handled by valuing all votes from the input with the value 1. The provision in Clause 3(2) is handled by the way the function `ballot_is_for` can be told to ignore certain candidates. In particular, candidates in the `dead` set are always ignored. Finally, Clause 3(3) is a calculation that the model allows to be performed at any time through the use of the `successful_candidates` function.

28 ⟨*Clause 3: First preferences* 28⟩≡ (27)

```
val clause3_def = Define`
  clause3 (bbag, n, dead) : fullstate =
    let add_to_pile b fm =
          let c = HD (FILTER (\c. ~(c IN dead)) b) in
          let oldv = if c IN dom fm then fm ' c else {||}
          in
              fm |+ (c, BAG_INSERT (b, 1) oldv) in
    let initial_fm = ITBAG add_to_pile bbag FEMPTY in
    let transform b = <| count := card b; old_votes := {||};
                         latest_votes := b |>
    in
      (<| vote_pile     := transform o_f initial_fm ;
          set_aside      := {||} ;
          dead           := dead ;
          excluded       := {};
          done_successes := {};
          num_to_elect   := n;
          quota          := (card(bbag) DIV (n+1)) + 1 |>,
       [])
`;
```

Defines:
`clause3`, used in chunk 42a.
Uses `card` 57a, `dom` 57a, and `fullstate` 21a.

5.2 Clause 4: Scrutiny to cease

4 Scrutiny to cease

(1) If, after a calculation under clause 3 (3), 6 (4), or 9 (2)(d), the number of successful candidates is equal to the number of positions to be filled, the scrutiny shall cease.

(2) If, after a calculation under clause 3 (3) or 6 (4) or after all the ballot papers counted for an excluded candidate have been dealt with under clause 9—

(a) the number of continuing candidates is equal to the number of positions remaining to be filled; and

(b) no successful candidate has a surplus not already dealt with under clause 6;

each of those continuing candidates is successful and the scrutiny shall cease.

Figure 5.2: Part 4.2, clause 4

The text of Clause 4 is presented in Figure 5.2. As previously dicussed, the formalisation of this clause is as a function that determines whether or not the algorithm is finished. This makes it a function from the state type to either true or false. The definition admits two ways in which the algorithm might be ready to terminate. These correspond to the two sub-clauses ((1) and (2)) of Clause 4. The first disjunct checks to see if the number of successful candidates is equal to the number of candidates to elect. If this is true, the algorithm can finish. Alternatively, there are as many continuing candidates as there are positions still needing to be filled, and no successful candidate has a surplus. These two sub-conditions correspond to sub-clauses (2)(a) and (2)(b). Recall that the done_successes component of the state records precisely those successful candidates who have no surplus. The assertion that there are no successful candidates with a surplus is thus the same as stating that the successful candidates are the same as those recorded in done_successes.

29 ⟨*Clause 4: Scrutiny to cease* 29⟩≡ (27) 30▷

```
val clause4_def = Define‘
  clause4 (s : state) =
    (card (successful_candidates(s)) = s.num_to_elect) \/
    ((card (continuing_candidates(s)) = s.num_to_elect) /\
     ~?c. c IN successful_candidates(s) /\
          s.quota < candidate_votes s c)
‘;
```

Defines:
clause4, never used.
Uses candidate_votes 22b, card 57a, continuing_candidates 24, state 20, and successful_candidates 23a.

By making the formalisation of Clause 4 simply determine whether or not to stop the algorithm, it omits the "movement" of continuing candidates to *successful* status that is required in Clause 4(2). This is handled by defining a special function, `winners`, which determines the winners of an election when given a final state.

30 ⟨*Clause 4: Scrutiny to cease* 29⟩+≡ (27) ◁29

```
val winners_def = Define‘
  winners (s : state) =
    if card(successful_candidates(s)) = s.num_to_elect then
      successful_candidates(s)
    else
      successful_candidates(s) UNION continuing_candidates(s)
‘;
```

Defines:
`winners`, never used.

Uses `card` 57a, `continuing_candidates` 24, `state` 20, and `successful_candidates` 23a.

5.3 Clause 5: Scrutiny to continue

5 Scrutiny to continue

If the scrutiny has not ceased in accordance with clause 4 and—

(a) 1 or more successful candidates have a surplus not already dealt with under clause 6—subject to clause 4, each surplus shall be dealt with in accordance with clause 6; or

(b) there are not successful candidates with such a surplus—1 continuing candidate shall be excluded in accordance with clause 8 and the ballot papers counted for him or her shall be dealt with in accordance with clause 9.

Figure 5.3: Part 4.2, clause 5

The text of Clause 5 is presented in Figure 5.3. As previously discussed, the formalisation of this clause is as a function which determines whether or not there exists a successful candidate with a surplus. If this function returns true, then the next stage of the algorithm will be to deal with a successful candidate, according to clauses 6 and 7. If it returns false, then the algorithm will exclude a candidate.

31 ⟨*Clause 5: Scrutiny to continue* 31⟩≡ (27)

```
val clause5_def = Define‘
  clause5 (s : state) =
    ?c. c IN successful_candidates(s) /\
        s.quota < candidate_votes s c
‘;
```

Defines:
  clause5, never used.
Uses candidate_votes 22b, state 20, and successful_candidates 23a.

6 Surplus votes

(1) Subject to clause 7, this clause applies in relation to the surplus of a successful candidate.

(2) Each ballot paper counted for the purpose of allotting votes to the successful candidate at the count at which the candidate became successful shall be dealt with as follows:

(a) if it does not specify a next available preference—it shall be set aside as finally dealt with for this part;

(b) if it specifies a next available preference—it shall be grouped according to the candidate for whom that preference is recorded.

(3) The count votes for each continuing candidate shall be determined and allotted to him or her.

(4) After the allotment under subclause (3), the continuing candidates' total votes shall be calculated and, if the total votes of a candidate equal or exceed the quota, the candidate is successful.

Figure 5.4: Part 4.2, clause 6

5.4 Clause 6: Surplus votes

The text of Clause 6 is presented in Figure 5.4. The formalisation makes Clause 6 a function that takes a state and a candidate as input, and returns a new state as output. This new state is one in which the given candidate has moved from `open` to `done_successes`, and where the last batch of votes received by that candidate have been transferred. This function is quite complicated. In particular, there is a great deal of complexity due to the fact that votes are transferred in "bundles".

Now the definition of `clause6` can be made. The first two definitions (`ignores` and `successes`) are just abbreviations for standard components of the state. The values `c's_ballots` and `c's_lastballots` are together the multi-set of votes that were cast for the successful candidate, and `c's_count` is the count total of those votes. Those votes in `c's_lastballots` are subject to transfer. The `surplus` is the candidate's surplus. The value `with_next_cand` are those of `c`'s last ballots that have a next available candidate. This is assessed by ignoring the usual "ignores", which will include the candidate `c`. All of the other ballots are included in the bag, `to_set_aside`. The value `CP` is the "number of ballot papers counted for the candidate at the count at which he or she became successful and that specify a next available preference". The value `base_tv` is the base transfer value that will be used to value all transferred ballots, calculated in accordance with Clause 4.1(1). The local function `bfor` returns the candidate that a ballot paper is for by taking the first name on the list of candidates that is not to be ignored.

33 ⟨*Clause 6: Surplus votes* 33⟩≡ (27)

```
val clause6_def = Define‘
  clause6 ((s, history) : fullstate, c : candidate) =
    let ignores = ignored_candidates s in
    let ci = s.vote_pile ’ c in
    let surplus = ci.count - s.quota in
    let with_next_cand : tbp bag =
          filter (\b. ?c. ballot_is_for ignores c b)
                 ci.latest_votes in
    let to_set_aside =
          BAG_UNION ci.old_votes
                    (BAG_DIFF ci.latest_votes with_next_cand) in
    let CP = card(with_next_cand) in
    let base_tv = &surplus / &CP in
    let adjust_tv (b, tv) = if tv < base_tv then (b, tv)
                            else (b, base_tv) in
    let bundle = image adjust_tv with_next_cand in
    let bfor (b,tv) = HD (filter (\c. ~(c IN ignores)) b) in
    let new_pile = transfer_bundle bundle bfor s.vote_pile
    in
       (s with <| done_successes := c INSERT s.done_successes ;
                  set_aside := BAG_UNION s.set_aside to_set_aside ;
                  vote_pile := new_pile \\ c |>,
        s :: history)
‘;
```

Defines:
  `clause6`, used in chunk 42c.
Uses `ballot_is_for` 12a, `candidate` 8b, `card` 57a, `filter` 57a, `fullstate` 21a, `ignored_candidates` 23b, `image` 57a, `tbp` 9b, and `transfer_bundle` 18.

5.5 Clause 7: More than 1 surplus

The mechanisation's `clause7` is required to be a function which is given a state and a candidate, and returns a candidate. The candidate returned must be one that has a surplus, and satisfies the constraints of Clause 7. The input candidate is the mechanisation's representation of the commissioner's choice by lot of the "relevant candidate", needed in Clause 7(3)(c)(ii).

One important aspect of Clause 7 is its use of the notion of "earliest count". The `time_of_success` scans a history list to determine when a candidate became successful (remembering that history lists have most recent states ahead of older ones). The greater the index returned, the earlier in time the candidate became successful.

34 ⟨*Clause 7: More than 1 surplus* 34⟩≡ (27) 35 ▷

```
val time_of_success_def = Define`
  (time_of_success c t [] = t) /\
  (time_of_success c t (s::ss) =
     if c IN successful_candidates s then
        time_of_success c (t + 1) ss
     else t)
`;
```

Defines:
  `time_of_success`, used in chunk 35.
Uses `successful_candidates` 23a.

Next is the definition of a function called clause7_set. This function examines the algorithm state and returns a set of candidates eligible to be selected to be the candidate whose surplus votes are to be redistributed. In most cases, this set will contain just one candidate, but it is possible that the commissioner has to make a decision by lot. In this case, clause7_set returns a set of multiple candidates. The code is a series of if-then-else tests, with various branches labelled with comments (in (*-*) pairs) containing the sub-clause numbers that this test corresponds to. Thus the first test is to see whether or not there is a successful candidate with a surplus who became successful first.

35 ⟨*Clause 7: More than 1 surplus* 34⟩+≡ (27) ◁34 36▷

```
  val clause7_set_def = Define‘
    clause7_set ((s,h) : fullstate) : candidate set =
      let possibles = { c | c IN dom(s.vote_pile) /\
                            s.quota < candidate_votes s c }
      in
        if card (possibles) = 1 then possibles
        else
          let earliest =
                { c | c IN possibles /\
                      (time_of_success c 0 h =
                       MAX_SET (IMAGE (\c. time_of_success c 0 h)
                                      possibles)) }
          in
            if card earliest = 1 then earliest (* 3(a) *)
            else
              let greatest_surplus =
                    {c | c IN earliest /\
                         (candidate_votes s c =
                          MAX_SET (IMAGE (candidate_votes s)
                                         earliest))}
              in
                if card greatest_surplus = 1 then
                  greatest_surplus (* 3(b) *)
                else if ?t. INJ (candidate_votes (nth t h))
                                greatest_surplus UNIV
                then (* 3(c)(i) *)
                  let t = LEAST t. INJ (candidate_votes (nth t h))
                                       greatest_surplus UNIV in
                  let s0 = nth t h
                  in
                      { c | c IN greatest_surplus /\
                            (candidate_votes s0 c =
                             MAX_SET (IMAGE (candidate_votes s0)
                                            greatest_surplus)) }
                else greatest_surplus (* 3(c)(ii) *)
  ‘;
```

Defines:
  clause7_set, used in chunk 36.
Uses candidate 8b, candidate_votes 22b, card 57a, dom 57a, fullstate 21a, nth 57a,

`set` 57a, and `time_of_success` 34.

Finally, the `clause7` function itself can be defined. This function assumes that the commissioner makes their choice from among the candidates that `clause7_set` decides are possible. The `CHOICE` function takes a set and returns an arbitrary element from it (because we know the set has just one member, this "arbitrariness" is not an issue).

36 ⟨*Clause 7: More than 1 surplus* 34⟩+≡ (27) ◁35

```
val clause7_def = Define‘
  clause7 (sh: fullstate, c : candidate) =
    let possibles = clause7_set(sh)
    in
      if card(possibles) = 1 then CHOICE possibles
      else c
‘;
```

Defines:
`clause7`, never used.
Uses `candidate` 8b, `card` 57a, `clause7_set` 35, and `fullstate` 21a.

5.6 Clause 8: Exclusion of candidates

Clause 8 is concerned with the selection of a candidate to be excluded, if there are no successful candidates with surplus votes. As in Clause 7 (section 5.5), this process may produce a situation where the commissioner has to choose a candidate to exclude by lot. The definition of this clause is divided into two parts. The first of these is `clause8_set`, which is given an algorithm state and returns a set of candidates to exclude. If this set is not a singleton, then the commissioner will have to make their decision by lot.

Clause 8(1) says "... the candidate with the least total votes shall be excluded." This is modelled below by first calculating what the least number of votes is (the local definition of `least_votes`), and then the set of candidates with this many votes (`with_least`). If this set has one member, this set is returned. Otherwise, the test is to see whether or not there was ever a time "at which all those candidates had unequal votes". This test is modelled by the test to see whether or not the function returning a candidate's count at time `t` was ever injective over the set `with_least`. If it was then there was a time when they all had unequal votes. Then `t` is taken to be the "least" time when this was true (remembering that the lower `t` is, the more recent in the past `t` was), and the appropriate (one-element) set is returned.

37 ⟨*Clause 8: Exclusion of candidates* 37⟩≡ (27) 38▷

```
  val clause8_set_def = Define‘
    clause8_set (s: state, h : state list) =
      let least_votes = LEAST v. ?c. c IN continuing_candidates s /\
                                     (candidate_votes s c = v) in
      let with_least = { c | candidate_votes s c = least_votes }
      in
        if card(with_least) = 1 then with_least (* 1 *)
        else
          if ?t. INJ (candidate_votes (nth t h)) with_least UNIV then
            (* 2(a) *)
            let t = LEAST t. INJ (candidate_votes (nth t h))
                                 with_least UNIV in
            let s0 = nth t h
            in
                { c | c IN with_least /\
                      (candidate_votes s0 c =
                       MIN_SET (IMAGE (candidate_votes s0) with_least))}
          else with_least (* 2(b) *)
  ‘;
```

Defines:
`clause8_set`, used in chunk 38.
Uses `candidate_votes` 22b, `card` 57a, `continuing_candidates` 24, `nth` 57a, and `state` 20.

Finally, as with Clause 7, the definition of `clause8` can be made.

38 ⟨*Clause 8: Exclusion of candidates* 37⟩+≡ (27) ◁37

```
val clause8_def = Define‘
  clause8 (sh: fullstate, c : candidate) =
    let possibles = clause8_set(sh)
    in
        if card(possibles) = 1 then CHOICE possibles
        else c
‘;
```

Defines:
`clause8`, never used.
Uses `candidate` 8b, `card` 57a, `clause8_set` 37, and `fullstate` 21a.

5.7 Clause 9: Votes of excluded candidates

The formalisation of `clause9` is the most complicated of all the clauses because it embeds multiple counts, each corresponding to the transfer of packets of votes of different transfer values. It is otherwise similar to `clause6` (see section 5.4).

Each count is performed by the local function `do_one_count`, which takes a `fullstate` and a bundle of votes known to have the same transfer value. These bundles are in processed in order of decreasing transfer value. The bundles are created by first partitioning those ballots of the excluded candidate that have continuing candidates remaining on them, and then performing a sort.

39 ⟨*Clause 9: Votes of excluded candidates* 39⟩≡ (27)

```
val clause9_def = Define‘
  clause9 ((s: state, h: state list), c: candidate) =
    let votes = s.vote_pile in
    let ct = votes ’ c in
    let ballots = BAG_UNION ct.old_votes ct.latest_votes in
    let ignores = c INSERT ignored_candidates s in
    let has_next_candidate =
          filter (\b. ?c. ballot_is_for ignores c b) ballots in
    let to_set_aside = BAG_DIFF ballots has_next_candidate in
    let tveq (b1, tv1) (b2, tv2) = (tv1 = tv2) in
    let tv_groups = partitionb tveq has_next_candidate in
    let group_tv gp = SND (BAG_CHOICE gp) in
    let order gp1 gp2 = group_tv gp2 <= group_tv gp1 in
    let tv_groups_ordered = sort order (SET_TO_LIST tv_groups) in
    let bfor (b, tv) = HD (filter (\c. ~(c IN ignores)) b) in
    let do_one_count (s,h) gp =
          (s with vote_pile updated_by (transfer_bundle gp bfor),
           s::h)
    in
    let base_s =
          s with <| set_aside updated_by ((BAG_UNION) to_set_aside);
                    vote_pile := votes \\ c ;
                    excluded updated_by ((INSERT) c) |>
    in
        FOLDL do_one_count (base_s, h) tv_groups_ordered
‘;
```

Defines:
  `clause9`, never used.
Uses `ballot_is_for` 12a, `candidate` 8b, `filter` 57a, `ignored_candidates` 23b, `partitionb` 61a, `sort` 62b, `state` 20, and `transfer_bundle` 18.

Chapter 6

Theorems

This chapter includes the statement of the verification results proved about the electoral algorithm. Proofs will be commented on here, but (mainly unreadable) HOL tactics will be relegated to an appendix.

The *key* arithmetic property, which states that the quota is not so large as to allow more than the desired number of candidates to win election:

40 ⟨*Theorems* 40⟩≡ (5b) 41 ▷

```
val key_property = store_thm(
  "key_property",
  ``!x n. x DIV (x DIV (n + 1) + 1) <= n``,
  ⟨Proof of the key property 45a⟩)
```

6.1 Clause 3 creates well-formed states

41 ⟨*Theorems* 40⟩+≡ (5b) ◁40 42a▷
```
val sum_image_empty_bag = store_thm(
  "sum_image_empty_bag",
  ``!s. finite s ==> (SUM_IMAGE {||} s = 0)``,
  SIMP_TAC (srw_ss()) [EMPTY_BAG] THEN
  HO_MATCH_MP_TAC FINITE_INDUCT THEN
  SRW_TAC [][SUM_IMAGE_THM] THEN
  Q_TAC SUFF_TAC `s DELETE e = s` THEN1 SRW_TAC [][] THEN
  SRW_TAC [][EXTENSION] THEN PROVE_TAC []);
val sum_image_bag_insert = store_thm(
  "sum_image_bag_insert",
  ``!s. finite s ==>
        !e b. SUM_IMAGE (BAG_INSERT e b) s =
              if e IN s then SUM_IMAGE b s + 1
              else SUM_IMAGE b s``,
  SIMP_TAC (srw_ss()) [BAG_INSERT] THEN
  HO_MATCH_MP_TAC FINITE_INDUCT THEN
  SRW_TAC [][SUM_IMAGE_THM] THEN
  `s DELETE e = s` by (SRW_TAC [][EXTENSION] THEN PROVE_TAC []) THEN
  SRW_TAC [][] THEN FULL_SIMP_TAC (srw_ss()) [] THEN
  SIMP_TAC arith_ss []);

val finite_bag_sum_image = store_thm(
  "finite_bag_sum_image",
  ``!b. finite b ==>
        !s. finite s ==> SUM_IMAGE b s <= card b``,
  HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
  ASM_SIMP_TAC (srw_ss()) [BAG_CARD_THM, sum_image_bag_insert,
                           sum_image_empty_bag] THEN
  SRW_TAC [][] THEN RES_TAC THEN
  SRW_TAC [numSimps.ARITH_ss][]);

val card_bag_image = store_thm(
  "card_bag_image",
  ``!b: 'a bag. finite b ==> (card (image f b) = card b)``,
  HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
  SRW_TAC [][BAG_IMAGE_EMPTY, BAG_IMAGE_FINITE_INSERT,
             BAG_CARD_THM, BAG_IMAGE_FINITE]);
```

Uses `card` 57a, `finite` 57a, and `image` 57a.

42a ⟨*Theorems* 40⟩+≡ (5b) ◁41 42b▷
```
  val clause3_wf = store_thm(
    "clause3_wf",
    ``finite votes /\ set votes SUBSET wf_ballot UNIV /\
      finite deads /\
      (!b. b IN set votes ==> ?c. MEM c b /\ ~(c IN deads)) ==>
      wf_state (FST (clause3(votes, num_to_elect, deads)))``,
    ⟨Proof that clause3 preserves well-formed states 47⟩);
```
Defines:
  `clause3_wf`, never used.
Uses `clause3` 28, `finite` 57a, `set` 57a, `wf_ballot` 10a, and `wf_state` 25.

6.2 Clause 6 preserves well-formed states

First a proof that the `transfer_bundle` function preserves the set of candidates mentioned in a the system's finite map (that is, `s.vote_pile` for some state `s`):

42b ⟨*Theorems* 40⟩+≡ (5b) ◁42a 42c▷
```
  val transfer_bundle_preserves_votes = store_thm(
    "transfer_bundle_preserves_votes",
    ``(p ' c = (n, ignored1, BAG_UNION ignored2 bnd)) ==>
      (pile_candidates (transfer_bundle bnd bfor p \\ c) UNION
       ignored1 UNION ignored2 =
       pile_candidates p)``,
    ⟨Proof that transfer_bundle transfers votes 54⟩);
```
Defines:
  `transfer_bundle_preserves_votes`, never used.
Uses `pile_candidates` 16b and `transfer_bundle` 18.

42c ⟨*Theorems* 40⟩+≡ (5b) ◁42b
```
  val clause6_wf = store_thm(
    "clause6_wf",
    ``s0.quota < candidate_votes s0 c /\ c IN all_candidates s0 ==>
      wf_state s0 ==> wf_state(FST (clause6((s0, h), c)))``,
    ⟨Proof that clause6 preserves well-formed states 55⟩)
```
Defines:
  `clause6_wf`, never used.
Uses `all_candidates` 22a, `candidate_votes` 22b, `clause6` 33, and `wf_state` 25.

Chapter 7

Change Log

Version 1 This version of the document was prepared by Michael Norrish with reference only to the Electoral Act, and without any discussion of the Act's possible intricacies with anyone previously involved in the eVACS project.

Version 2 This version of the document was prepared by Michael Norrish after consulting with Jeremy Dawson. There were two significant problems with version 1:

- when doing a transfer of surplus votes, the system should only transfer those parcel of votes that tipped a candidate over from being unsuccessful to successful.
- redistribution of excluded candidates' votes really does need to be in the separate phases described in the act because each such phase is a "count", and the choice of successful candidates whose surplus votes are to be redistributed is done with earliest candidate first. Version 1 would make it seem as if all successful candidates became so at the same time.

I also chose to rework the states so that they did not include their own histories, but rather to have the current history accompany the current state as an argument to all clauses.

Version 2.01: Noticed that the implementation of Clause 9 (Section 5.7) didn't move an excluded candidate into the state's `excluded` field, though it did remove the candidate from the domain of the `vote_pile` map.

Version 2.02: Noticed an error in the definition of `all_candidates`, which would have caused this set to incorrectly exclude candidates with no (first) votes. The same problem afflicted the definition of the function continuing_candidates. The domain of s.vote_pile is that set of candidates who are continuing and have votes, or those candidates who are successful but without a surplus. Given this, it's difficult to imagine the context in which a reference to dom(s.vote_pile) could make any sense.

Version 2.03: An error in clause8_set, which calculated the least number of votes for a candidate. It looked at all possible candidates rather than just continuing candidates. Also fixed all_candidates, which ignored continuing candidates mentioned only on ballots in the set_aside area. These candidates can not play any further role except to be excluded, but defining all_candidates this way would wreck the invariant that the system never "loses" candidates.

Version 2.04: This new version should be equivalent to the previous, but is quite different in its expression of the basic algorithm. A new type, that of the candidate information record, is introduced, and the notion of ignored candidate is expanded to include all successful candidates, not just those that have had surplus votes transferred away from them. The presentation should be logically cleaner, and easier to understand.

It is still undecided as to whether or not successful candidates with exactly the right number of votes should be withdrawn from the algorithm state's "vote pile". This would move these candidates' votes into the set_aside part of the state, and the candidates into the done_successes set.

The advantage of doing this is that it would tighten the correspondence between the algorithm states and the states in the Act, where all successful candidates have their votes set aside. The disadvantage is that it introduces a new phase into the algorithm's basic loop, *"clean up exact successes"*, which doesn't really look like anything in the Act.

Appendix A

Proofs

A.1 Clause 3 creates only well-formed states

This appendix contains proofs that have been elided from the main body of the text. These are for consumption by HOL, and this means that they are not human-readable, unfortunately.

45a ⟨*Proof of the key property* 45a⟩≡ (40)

```
  `!x y z. 0 < y ==> (x DIV y <= z = x < (z + 1) * y)`
     by PROVE_TAC [arithmeticTheory.DIV_LE_X] THEN
  ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss) [] THEN
  Q_TAC SUFF_TAC `!n x. 0 < n ==> x < n * (x DIV n + 1)` THEN1
    ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss) [] THEN
  POP_ASSUM (K ALL_TAC) THEN REPEAT STRIP_TAC THEN
  Q.SPEC_THEN `n` MP_TAC DIVISION THEN
  ASM_REWRITE_TAC [] THEN
  DISCH_THEN (Q.SPEC_THEN `x` STRIP_ASSUME_TAC) THEN
  Q.ABBREV_TAC `q = x DIV n` THEN
  Q.ABBREV_TAC `r = x MOD n` THEN
  ASM_SIMP_TAC (srw_ss()) [AC MULT_ASSOC MULT_COMM,
                           LEFT_ADD_DISTRIB]
```

45b ⟨*Properties of ballot papers* 45b⟩≡ (12b) 46 ▷

```
  val wf_tbp_subset = prove(
    ``wf_tbp s b /\ s SUBSET t ==> wf_tbp t b``,
    Cases_on `b` THEN SRW_TAC [][wf_tbp_def, wf_ballot_def] THEN
    PROVE_TAC [pred_setTheory.SUBSET_TRANS]);
  val wf_tbp_subset' = prove(
    ``!s t. s SUBSET t ==> (wf_tbp s b ==> wf_tbp t b)``,
    PROVE_TAC [wf_tbp_subset]);
```

Uses `wf_tbp` 10b.

The next two properties are particularly trivial. The first should probably be part of the system's “standard library”.

46 ⟨*Properties of ballot papers* 45b⟩+≡ (12b) ◁45b

```
val filter_eq_nil = prove(
  ``!l. (filter P l = []) = (!x. MEM x l ==> ~P x)``,
  Induct THEN SRW_TAC [boolSimps.DNF_ss][]);

val DIFF_INTER_disjoint = prove(
  ``disjoint (X DIFF (Y UNION Z)) Z``,
  SRW_TAC [][EXTENSION, DISJOINT_DEF] THEN PROVE_TAC []);
```

Uses `disjoint` 57a and `filter` 57a.

And the following is the main proof, in one indigestible chunk:

47 ⟨*Proof that* `clause3` *preserves well-formed states* 47⟩≡ (42a)

```
SIMP_TAC bool_ss [clause3_def] THEN
RAISE_LETS_TAC THEN
ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss)
             [pairTheory.UNCURRY, wf_state_def,
              IN_SET_OF_BAG, o_f_FAPPLY,
              BAG_IN_BAG_INSERT, BAG_OF_EMPTY,
              DISJ_IMP_THM, FORALL_AND_THM] THEN
STRIP_TAC THEN
`!b1 b2 fm. add_to_pile b1 (add_to_pile b2 fm) =
            add_to_pile b2 (add_to_pile b1 fm)`
    by (REPEAT GEN_TAC THEN
        Q.PAT_ASSUM `x = add_to_pile`
                    (fn th => REWRITE_TAC [SYM th]) THEN
        RAISE_LETS_TAC THEN SIMP_TAC bool_ss [] THEN
        Cases_on `c = c'` THENL [
          FULL_SIMP_TAC (srw_ss())
                        [FUPD11_SAME_KEY_AND_BASE] THEN
          SRW_TAC [][bagTheory.BAG_INSERT_commutes],
          FULL_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM] THEN
          ASM_SIMP_TAC (srw_ss()) [FUPDATE_COMMUTES]
        ]) THEN
Q.ABBREV_TAC `top_cand = \b. HD (filter (\c. ~(c IN deads)) b)` THEN
`!b fm. add_to_pile b fm =
          let c = top_cand b in
          let oldv = (if c IN dom fm then fm ' c else {||}) in
            fm |+ (c, BAG_INSERT (b,1) oldv)` by SRW_TAC [][] THEN
`wf_pile <| vote_pile := (\b. (card b, {||}, b)) o_f initial_fm;
            set_aside := {||}; dead := deads; excluded := {};
            done_successes := {};
            quota := card votes DIV (num_to_elect + 1) + 1;
            num_to_elect := num_to_elect |>`
  by (REPEAT STRIP_TAC THEN
      ASM_SIMP_TAC (srw_ss()) [wf_pile_def, pairTheory.UNCURRY,
                               o_f_FAPPLY, BAG_IN_BAG_INSERT] THEN
      GEN_TAC THEN STRIP_TAC THEN
      ASM_SIMP_TAC (srw_ss()) [sum_vote_bag_def] THEN
      `!votes. finite votes ==>
               !c. c IN dom (ITBAG add_to_pile votes FEMPTY) ==>
                   finite (ITBAG add_to_pile votes FEMPTY ' c)`
         by (HO_MATCH_MP_TAC bagTheory.STRONG_FINITE_BAG_INDUCT THEN
             ASM_SIMP_TAC (srw_ss()) [bagTheory.COMMUTING_ITBAG_RECURSES] THEN
             GEN_TAC THEN STRIP_TAC THEN REPEAT GEN_TAC THEN
             Cases_on `c' = top_cand e` THENL [
               ASM_SIMP_TAC (srw_ss()) [] THEN COND_CASES_TAC THEN
               ASM_SIMP_TAC (srw_ss()) [],
               ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM]
             ]) THEN
```

```
SIMP_TAC (srw_ss()) [FORALL_AND_THM, IMP_CONJ_THM] THEN
ASM_SIMP_TAC (srw_ss()) [ignored_candidates_def, all_candidates_def] THEN
Q.UNDISCH_THEN `c IN dom initial_fm` MP_TAC THEN
Q.UNDISCH_THEN `set votes SUBSET wf_ballot UNIV` MP_TAC THEN
Q.UNDISCH_THEN `!b. BAG_IN b votes ==> ?c. MEM c b /\ ~(c IN deads)`
               MP_TAC THEN
Q.UNDISCH_THEN `finite votes` MP_TAC THEN
Q.PAT_ASSUM `x = initial_fm` (SUBST_ALL_TAC o SYM) THEN
Q.ID_SPEC_TAC `votes` THEN
HO_MATCH_MP_TAC bagTheory.STRONG_FINITE_BAG_INDUCT THEN
ASM_SIMP_TAC (srw_ss()) [BAG_IN_BAG_INSERT] THEN
GEN_TAC THEN STRIP_TAC THEN GEN_TAC THEN
SIMP_TAC (srw_ss()) [DISJ_IMP_THM, FORALL_AND_THM,
                     SET_OF_BAG_INSERT] THEN
REPEAT (DISCH_THEN STRIP_ASSUME_TAC) THEN
Q.PAT_ASSUM `x ==> y` MP_TAC THEN ASM_SIMP_TAC (srw_ss()) [] THEN
STRIP_TAC THEN
Q.PAT_ASSUM `c IN dom x` MP_TAC THEN
Cases_on `c IN dom (ITBAG add_to_pile votes FEMPTY)` THENL [
  Cases_on `c = top_cand e`,
  ALL_TAC
] THEN
TRY (Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC) THEN
ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES,
                         FAPPLY_FUPDATE_THM, BAG_IN_BAG_INSERT,
                         SET_OF_BAG_INSERT] THEN
`!r1 r2 r3:real. r1 + (r2 + r3) = r2 + (r1 + r3)`
   by RealArith.REAL_ARITH_TAC THEN
REPEAT STRIP_TAC THENL [
  (* ballot exists in pile *)
  SIMP_TAC bool_ss [EXISTS_OR_THM],

  (* pile count <= size of pile *)
  ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES,
                           BAG_IMAGE_FINITE, BAG_CARD_THM] THEN
  REWRITE_TAC [
    GSYM realTheory.REAL_ADD,
    RealArith.REAL_ARITH
      ``!x y c:real. x + c <= c + y = x <= y``] THEN
  ASM_SIMP_TAC (srw_ss()) [],

  (* every ballot well-formed - b = (e, 1) *)
  Q.UNDISCH_THEN `e IN wf_ballot UNIV`
                 (MP_TAC o REWRITE_RULE [SPECIFICATION]) THEN
  ASM_SIMP_TAC (srw_ss()) [wf_ballot_def, wf_tbp_def, SUBSET_DEF,
                           SET_OF_BAG_INSERT, pile_candidates_def] THEN
  STRIP_TAC THEN MAP_EVERY Q.EXISTS_TAC [`1`,`1`] THEN
  SRW_TAC [][],

  (* every ballot well-formed - b IN pile for candidate (top_cand e) *)
```

```
Q.PAT_ASSUM `x ==> y` MP_TAC THEN ASM_SIMP_TAC (srw_ss()) [] THEN
STRIP_TAC THEN
REPEAT (FIRST_X_ASSUM (Q.SPEC_THEN `b` MP_TAC)) THEN
ASM_SIMP_TAC (srw_ss()) [] THEN REPEAT STRIP_TAC THEN
Q.PAT_ASSUM `wf_tbp x y` MP_TAC THEN
MATCH_MP_TAC wf_tbp_subset' THEN
ASM_SIMP_TAC (srw_ss() ++ boolSimps.DNF_ss ++ boolSimps.CONJ_ss)
             [SUBSET_DEF, IN_RAN, pairTheory.FORALL_PROD,
              pairTheory.EXISTS_PROD, o_f_FAPPLY, pile_candidates_def,
              DOMSUB_FAPPLY_THM, SET_OF_BAG_INSERT] THEN
REPEAT GEN_TAC THEN Cases_on `y = top_cand e` THEN
ASM_SIMP_TAC (srw_ss()) [] THEN PROVE_TAC [],

(* ballot is for (top_cand e) - b = (e, 1) *)
ASM_SIMP_TAC (srw_ss()) [ballot_is_for_def] THEN
Q.PAT_ASSUM `x = top_cand` (SUBST_ALL_TAC o SYM) THEN
ASM_SIMP_TAC (srw_ss()) [filter_eq_nil] THEN PROVE_TAC [],

(* ballot is for (top_cand e) - b IN pile for candidate (top_cand e) *)
ASM_SIMP_TAC (srw_ss()) [],

(* ballot is well-formed *)
Q.PAT_ASSUM `x ==> y` MP_TAC THEN
ASM_SIMP_TAC (srw_ss()) [] THEN
REPEAT STRIP_TAC THEN
Q.PAT_ASSUM `!b. BAG_IN b x ==> wf_tbp y b`
            (Q.SPEC_THEN `b` MP_TAC) THEN
ASM_SIMP_TAC (srw_ss()) [] THEN
MATCH_MP_TAC wf_tbp_subset' THEN
ASM_SIMP_TAC (srw_ss() ++ boolSimps.DNF_ss ++ boolSimps.CONJ_ss)
             [SUBSET_DEF, IN_RAN, pairTheory.FORALL_PROD,
              pairTheory.EXISTS_PROD, o_f_FAPPLY, pile_candidates_def,
              DOMSUB_FAPPLY_THM, SET_OF_BAG_INSERT] THEN
REPEAT GEN_TAC THEN Cases_on `y = top_cand e` THEN
ASM_SIMP_TAC (srw_ss()) [] THEN PROVE_TAC [],

(* pile non-empty *)
FULL_SIMP_TAC (srw_ss()) [],

(* pile count <= size of piles  - c = top_cand e *)
Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC THEN
ASM_SIMP_TAC (srw_ss()) [BAG_CARD_THM, COMMUTING_ITBAG_RECURSES],

(* b well-formed - b = (e, 1), c = top_cand e *)
Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC THEN
Q.UNDISCH_THEN `b = (e, 1)` SUBST_ALL_TAC THEN
Q.UNDISCH_THEN `e IN wf_ballot UNIV`
               (MP_TAC o REWRITE_RULE [SPECIFICATION]) THEN
ASM_SIMP_TAC (srw_ss() ++ boolSimps.DNF_ss ++ boolSimps.CONJ_ss)
             [SUBSET_DEF, IN_RAN, pairTheory.FORALL_PROD,
                      pairTheory.EXISTS_PROD, o_f_FAPPLY, wf_ballot_def,
                      wf_tbp_def, DOMSUB_FAPPLY_THM, SET_OF_BAG_INSERT,
                      pile_candidates_def] THEN
        STRIP_TAC THEN MAP_EVERY Q.EXISTS_TAC [`1`, `1`] THEN SRW_TAC [][],

        Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC THEN
        Q.PAT_ASSUM `BAG_IN b (COND x y z)` MP_TAC THEN
        ASM_SIMP_TAC (srw_ss()) [],

        Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC THEN
        Q.UNDISCH_THEN `b = (e, 1)` SUBST_ALL_TAC THEN
        Q.PAT_ASSUM `x = top_cand` (SUBST_ALL_TAC o SYM) THEN
        ASM_SIMP_TAC (srw_ss()) [ballot_is_for_def, filter_eq_nil] THEN
        PROVE_TAC [],

        Q.UNDISCH_THEN `c = top_cand e` SUBST_ALL_TAC THEN
        Q.PAT_ASSUM `BAG_IN b (COND x y z)` MP_TAC THEN
        ASM_SIMP_TAC (srw_ss()) []
      ]) THEN
REPEAT GEN_TAC THEN STRIP_TAC THEN REPEAT CONJ_TAC THENL [
  (* wf_pile - already proved *)
  ASM_REWRITE_TAC [],

  (* successful candidates are finite *)
  ASM_SIMP_TAC (srw_ss()) [successful_candidates_def] THEN
  MATCH_MP_TAC pred_setTheory.INTER_FINITE THEN
  ASM_SIMP_TAC (srw_ss() ++ boolSimps.DNF_ss)
               [IN_RAN, all_candidates_def, pairTheory.FORALL_PROD,
                o_f_FAPPLY, GSYM LEFT_FORALL_IMP_THM, pile_candidates_def,
                DOMSUB_FAPPLY_THM, SET_OF_BAG_INSERT] THEN
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC IMAGE_FINITE THEN
  Q.PAT_ASSUM `X = initial_fm` (SUBST_ALL_TAC o SYM) THEN
  Q.PAT_ASSUM `y IN dom X` MP_TAC THEN
  Q.ID_SPEC_TAC `y` THEN
  Q.UNDISCH_THEN `finite votes` MP_TAC THEN
  Q.ID_SPEC_TAC `votes` THEN
  HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
  ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES] THEN
  GEN_TAC THEN STRIP_TAC THEN REPEAT GEN_TAC THEN
  Cases_on `y = top_cand e` THENL [
    ASM_SIMP_TAC (srw_ss()) [SET_OF_BAG_INSERT] THEN
    COND_CASES_TAC THENL [PROVE_TAC [], SRW_TAC [][BAG_OF_EMPTY]],
    ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM]
  ],

  (* cardinality of successful candidates is <= num_to_elect *)
  ASM_SIMP_TAC (srw_ss()) [successful_candidates_def] THEN
  `!n. 0 < n ==>
       !s. wf_pile s ==>
```

```
         { c | n <= candidate_votes s c } SUBSET all_candidates s`
    by (REPEAT (POP_ASSUM (K ALL_TAC)) THEN
        GEN_TAC THEN STRIP_TAC THEN
        `!P x. n <= (if P then x else 0) = P /\ n <= x`
             by SRW_TAC [numSimps.ARITH_ss][] THEN
        REWRITE_TAC [wf_pile_def] THEN REPEAT STRIP_TAC THEN
        Q_TAC SUFF_TAC `!c. c IN {c | n <= candidate_votes s c} ==>
                            c IN all_candidates s` THEN1
              SRW_TAC [][SUBSET_DEF] THEN
        SRW_TAC [boolSimps.DNF_ss]
                [all_candidates_def, candidate_votes_def,
                 pile_candidates_def, pairTheory.UNCURRY,
                 pairTheory.EXISTS_PROD, IN_RAN] THEN
        Q.PAT_ASSUM `!c. c IN dom s.vote_pile ==> X`
                    (Q.SPEC_THEN `c` MP_TAC) THEN
        ASM_SIMP_TAC (srw_ss()) [pairTheory.UNCURRY] THEN
        STRIP_TAC THEN
        `wf_tbp (all_candidates s) b /\
         ballot_is_for (ignored_candidates s) c b` by PROVE_TAC [] THEN
        `?b0 tv. b = (b0, tv)`
             by PROVE_TAC [TypeBase.nchotomy_of "prod"] THEN
        POP_ASSUM SUBST_ALL_TAC THEN
        `?pile1 pile2 cnt. (s.vote_pile ' c) = (cnt, pile1, pile2)`
             by PROVE_TAC [TypeBase.nchotomy_of "prod"] THEN
        FULL_SIMP_TAC (srw_ss()) [ballot_is_for_def, filter_eq_nil] THEN
        Q_TAC SUFF_TAC `MEM c b0` THEN1 PROVE_TAC [] THEN
        Q.UNDISCH_THEN `MEM c' b0` MP_TAC THEN
        Q.UNDISCH_THEN `~(c' IN ignored_candidates s)` MP_TAC THEN
        Q.UNDISCH_THEN
          `HD (filter (\c. ~(c IN ignored_candidates s)) b0) = c`
          MP_TAC THEN
        Q.ID_SPEC_TAC `b0` THEN
        REPEAT (POP_ASSUM (K ALL_TAC)) THEN Induct THEN
        SRW_TAC [][] THEN PROVE_TAC []) THEN
`!s t:candidate set. s SUBSET t ==> (t INTER s = s)`
    by PROVE_TAC [SUBSET_INTER_ABSORPTION, INTER_COMM] THEN
ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss) [] THEN
Q.PAT_ABBREV_TAC `s = r with vote_pile := v` THEN
`(candidate_votes s = image top_cand votes) /\
 !c. c IN dom initial_fm ==> finite (initial_fm ' c)`
   by (ASM_SIMP_TAC (srw_ss())[candidate_votes_def, FUN_EQ_THM,
                               BAG_IMAGE_DEF] THEN
       Q.PAT_ASSUM `X = s` (SUBST_ALL_TAC o SYM) THEN
       ASM_SIMP_TAC (srw_ss()) [o_f_FAPPLY] THEN
       Q.PAT_ASSUM `X = initial_fm` (SUBST_ALL_TAC o SYM) THEN
       Q.UNDISCH_THEN `finite votes` MP_TAC THEN
       Q.ID_SPEC_TAC `votes` THEN
       HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
       ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES] THEN
       REPEAT STRIP_TAC THEN
       Cases_on ‘x = top_cand e‘ THEN ASM_SIMP_TAC (srw_ss()) [] THENL [
         ASM_SIMP_TAC (srw_ss()) [FINITE_BAG_FILTER, BAG_CARD_THM] THEN
         Cases_on
           ‘top_cand e IN dom (ITBAG add_to_pile votes’ FEMPTY)‘
         THENL [
           ASM_SIMP_TAC (srw_ss()) [BAG_CARD_THM] THEN
           METIS_TAC [],
           ASM_SIMP_TAC (srw_ss()) [BAG_CARD_THM] THEN
           ‘card (filter (\e0. top_cand e0 = top_cand e) votes’) = 0‘
               by METIS_TAC [] THEN
           ASM_SIMP_TAC (srw_ss()) []
         ],
         ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM],
         COND_CASES_TAC THEN ASM_SIMP_TAC (srw_ss()) [],
         COND_CASES_TAC THEN ASM_SIMP_TAC (srw_ss()) [],
         COND_CASES_TAC THEN
         ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM] THEN
         COND_CASES_TAC THEN ASM_SIMP_TAC (srw_ss()) [],
         COND_CASES_TAC THEN
         ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM] THEN
         COND_CASES_TAC THEN ASM_SIMP_TAC (srw_ss()) []
       ]) THEN
‘card (candidate_votes s) = card votes‘
    by SRW_TAC [][card_bag_image] THEN
Q.PAT_ABBREV_TAC ‘putatives = {c | x <= candidate_votes s c}‘ THEN
‘putatives SUBSET set (candidate_votes s)‘
   by (Q.PAT_ASSUM ‘X = putatatives‘ (SUBST_ALL_TAC o SYM) THEN
       SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss)
                [SUBSET_DEF, BAG_IN, BAG_INN]) THEN
‘finite (candidate_votes s)‘ by PROVE_TAC [BAG_IMAGE_FINITE] THEN
‘finite putatives‘ by PROVE_TAC [FINITE_SET_OF_BAG, SUBSET_FINITE] THEN
‘SUM_IMAGE (candidate_votes s) putatives <= card (candidate_votes s)‘
   by PROVE_TAC [finite_bag_sum_image] THEN
Q.ABBREV_TAC ‘Q = card votes DIV (num_to_elect + 1) + 1‘ THEN
‘!c. c IN putatives ==> Q <= candidate_votes s c‘ by SRW_TAC [][] THEN
‘card putatives * Q <= SUM_IMAGE (candidate_votes s) putatives‘
   by PROVE_TAC [SUM_IMAGE_lower_bound] THEN
‘card putatives * Q <= card (candidate_votes s)‘ by DECIDE_TAC THEN
‘0 < Q‘ by (SRW_TAC [][] THEN DECIDE_TAC) THEN
Q.UNDISCH_THEN ‘card (candidate_votes s) = card votes‘ SUBST_ALL_TAC THEN
‘card putatives <= card votes DIV Q‘
   by ASM_SIMP_TAC (srw_ss())[X_LE_DIV] THEN
‘card votes DIV Q <= num_to_elect‘ by PROVE_TAC [key_property] THEN
DECIDE_TAC,

(* continuing candidates and deads are disjoint *)
ASM_SIMP_TAC (srw_ss()) [continuing_candidates_def,
                         all_candidates_def,
                         successful_candidates_def,
                         ignored_candidates_def, DIFF_INTER_disjoint],
(* continuing candidates and successful candidates are disjoint -
    follows immediately from definition of continuing candidate *)
ASM_SIMP_TAC (srw_ss()) [continuing_candidates_def, DISJOINT_DEF,
                         EXTENSION] THEN PROVE_TAC [],

(* deads and successful candidates are disjoint *)
ASM_SIMP_TAC (srw_ss()) [successful_candidates_def, candidate_votes_def,
                         DISJOINT_DEF, EXTENSION] THEN
Q.X_GEN_TAC `c` THEN
Cases_on `c IN deads` THEN ASM_SIMP_TAC (srw_ss()) [] THEN
Q_TAC SUFF_TAC `~(c IN dom initial_fm)` THEN1 ASM_SIMP_TAC (srw_ss()) [] THEN
Q.UNDISCH_THEN `c IN deads` MP_TAC THEN
Q.ID_SPEC_TAC `c` THEN
Q.UNDISCH_THEN `!b. BAG_IN b votes ==> ?c. MEM c b /\ ~(c IN deads)`
               MP_TAC THEN
Q.PAT_ASSUM `X = initial_fm` (SUBST_ALL_TAC o SYM) THEN
Q.UNDISCH_THEN `finite votes` MP_TAC THEN
Q.ID_SPEC_TAC `votes` THEN HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES, BAG_IN_BAG_INSERT,
                         DISJ_IMP_THM, FORALL_AND_THM] THEN
Q.PAT_ASSUM `X = top_cand` (SUBST_ALL_TAC o SYM) THEN
ASM_SIMP_TAC (srw_ss()) [] THEN NTAC 2 STRIP_TAC THEN
Induct THEN SRW_TAC [][] THEN PROVE_TAC [],

(* total live votes doesn't get too big *)
`!x y z. x + (y + z:real) = y + (x + z)` by RealArith.REAL_ARITH_TAC THEN
Q.PAT_ABBREV_TAC `s = R with vote_pile := X` THEN
`!c. c IN dom initial_fm ==> finite (initial_fm ' c)`
   by (Q.UNDISCH_THEN `finite votes` MP_TAC THEN
       Q.PAT_ASSUM `X = initial_fm` (SUBST_ALL_TAC o SYM) THEN
       Q.ID_SPEC_TAC `votes` THEN
       HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
       ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES] THEN
       GEN_TAC THEN STRIP_TAC THEN REPEAT GEN_TAC THEN
       Cases_on `c = top_cand e` THEN
       ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM] THEN
       COND_CASES_TAC THEN ASM_SIMP_TAC (srw_ss()) []) THEN
`(total_live_votes s = &(card votes)) /\
 !c. c IN dom initial_fm ==> finite (initial_fm ' c)`
  by (Q.UNDISCH_THEN `finite votes` MP_TAC THEN
       Q.PAT_ASSUM `X = s` (SUBST_ALL_TAC o SYM) THEN
       Q.PAT_ASSUM `X = initial_fm` (SUBST_ALL_TAC o SYM) THEN
       Q.ID_SPEC_TAC `votes` THEN
       HO_MATCH_MP_TAC STRONG_FINITE_BAG_INDUCT THEN
       ASM_SIMP_TAC (srw_ss()) [total_live_votes_def,
                                o_ABS_R, COMMUTING_ITBAG_RECURSES,
                                FRANGEB_THM, sum_vote_bag_def,
                                BAG_CARD_THM] THEN
       GEN_TAC THEN STRIP_TAC THEN GEN_TAC THEN STRIP_TAC THENL [
                COND_CASES_TAC THENL [
                  ASM_SIMP_TAC (srw_ss()) [COMMUTING_ITBAG_RECURSES] THEN
                  Q.ABBREV_TAC `fm = ITBAG add_to_pile votes' FEMPTY` THEN
                  `?fm0 v. (fm = fm0 |+ (top_cand e, v)) /\
                           ~(top_cand e IN dom fm0)`
                     by PROVE_TAC [FM_PULL_APART] THEN
                  Q.PAT_ASSUM `X = &(card votes')` MP_TAC THEN
                  ASM_SIMP_TAC (srw_ss()) [DOMSUB_NOT_IN_DOM, FRANGEB_THM,
                                           COMMUTING_ITBAG_RECURSES] THEN
                  REWRITE_TAC [GSYM REAL_ADD] THEN
                  RealArith.REAL_ARITH_TAC,
                  ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss)
                               [DOMSUB_NOT_IN_DOM, COMMUTING_ITBAG_RECURSES]
                ],
                GEN_TAC THEN Cases_on `c = top_cand e` THEN
                ASM_SIMP_TAC (srw_ss()) [FAPPLY_FUPDATE_THM] THEN
                METIS_TAC [FINITE_EMPTY_BAG]
              ]) THEN
     ASM_SIMP_TAC (srw_ss()) [] THEN
     ASM_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss)[GSYM DIV_LT_X]
   ]
```

Uses `all_candidates` 22a, `ballot` 9a, `ballot_is_for` 12a, `candidate` 8b, `candidate_votes` 22b, `card` 57a, `disjoint` 57a, `dom` 57a, `filter` 57a, `finite` 57a, `ignored_candidates` 23b, `image` 57a, `set` 57a, `total_live_votes` 23c, `wf_ballot` 10a, `wf_pile` 17, and `wf_tbp` 10b.

A.2 Clause 6 preserves well-formed states

54 ⟨*Proof that* `transfer_bundle` *transfers votes* 54⟩≡ (42b)
```
  ALL_TAC
```

55 ⟨*Proof that* clause6 *preserves well-formed states* 55⟩≡ (42c)

```
SIMP_TAC bool_ss [clause6_def] THEN
RAISE_LETS_TAC THEN
`?c's_count c's_ballots c's_lastballots.
   x = (c's_count, c's_ballots, c's_lastballots)`
   by PROVE_TAC [TypeBase.nchotomy_of "prod"] THEN
POP_ASSUM SUBST_ALL_TAC THEN
SIMP_TAC bool_ss [pairTheory.UNCURRY, pairTheory.FST, pairTheory.SND] THEN
RAISE_LETS_TAC THEN
Q.PAT_ABBREV_TAC `s1 = X with done_successes := Y` THEN
`s1.dead = s0.dead` by SRW_TAC [][] THEN
`s1.excluded = s0.excluded` by SRW_TAC [][] THEN
`ignored_candidates s1 = c INSERT ignored_candidates s0`
   by (SIMP_TAC (srw_ss()) [ignored_candidates_def] THEN
       ASM_SIMP_TAC (srw_ss()) [] THEN
       SRW_TAC [][EXTENSION] THEN mesonLib.MESON_TAC []) THEN
`s1.quota = s0.quota` by SRW_TAC [][] THEN
`s1.num_to_elect = s0.num_to_elect` by SRW_TAC [][] THEN
`s1.done_successes = c INSERT s0.done_successes` by SRW_TAC [][] THEN
STRIP_TAC THEN
`c IN successes` by SRW_TAC [][successful_candidates_def, LESS_OR_EQ] THEN
`c IN FDOM s0.vote_pile`
   by (Q.UNDISCH_THEN `s0.quota < candidate_votes s0 c` MP_TAC THEN
       SIMP_TAC (srw_ss()) [candidate_votes_def] THEN SRW_TAC [][]) THEN
`all_candidates s1 = all_candidates s0`
   by (ASM_SIMP_TAC (srw_ss()) [all_candidates_def]) THEN
ASM_SIMP_TAC (srw_ss()) [wf_state_def] THEN REPEAT STRIP_TAC THENL [
  (* wf_pile *)
  ASM_SIMP_TAC (srw_ss()) [wf_pile_def, DOMSUB_FAPPLY_THM] THEN
  Q.X_GEN_TAC `d` THEN
  `?d's_count d's_ballots d's_lastballots.
      new_pile ' d = (d's_count, d's_ballots, d's_lastballots)`
      by PROVE_TAC [TypeBase.nchotomy_of "prod"] THEN
  ASM_SIMP_TAC (srw_ss()) []
]
```

Uses all_candidates 22a, candidate_votes 22b, ignored_candidates 23b, and wf_pile 17.

Appendix B

Additional HOL Material

This is the material necessary to make the source code “work” when passed to the HOL system, but is otherwise of limited interest.

56a ⟨*Preliminaries* 56a⟩≡ (5b)
  ⟨*Open standard HOL context* 56b⟩
  ⟨*Include parent theories* 56c⟩
  ⟨*Declare new theory* 56d⟩
  ⟨*Make parsing/pretty-printing alterations* 57a⟩
  ⟨*Auxiliary Definitions* 57c⟩

First open a standard HOL context:

56b ⟨*Open standard HOL context* 56b⟩≡ (56a)
```
open HolKernel Parse boolLib bossLib BasicProvers metisLib
```

The theory of the Electoral Act needs to refer to the built-in HOL theories of bags (multi-sets), real numbers and sets.

56c ⟨*Include parent theories* 56c⟩≡ (56a)
```
open realTheory bagTheory pred_setTheory finite_mapTheory
open arithmeticTheory
local open containerTheory in end
```

The theory will be called `ElectoralAct`.

56d ⟨*Declare new theory* 56d⟩≡ (56a)
```
val _ = new_theory "ElectoralAct"
```

Issue some overloading commands, to let functions from the HOL theory of bags be used with prettier names:

57a ⟨*Make parsing/pretty-printing alterations* 57a⟩≡ (56a) 57b ▷

```
val _ = overload_on ("card", ``BAG_CARD``);
val _ = overload_on ("card", ``pred_set$CARD``);
val _ = overload_on ("dom", ``finite_map$FDOM``);
val _ = overload_on ("disjoint", ``pred_set$DISJOINT``);
val _ = overload_on ("filter", ``BAG_FILTER``);
val _ = overload_on ("filter", ``list$FILTER``);
val _ = overload_on ("finite", ``FINITE_BAG``);
val _ = overload_on ("finite", ``pred_set$FINITE``);
val _ = overload_on ("image", ``BAG_IMAGE``);
val _ = overload_on ("image", ``pred_set$IMAGE``);
val _ = overload_on ("nth", ``list$EL``);
val _ = overload_on ("ran", ``finite_map$FRANGE``);
val _ = overload_on ("set",  ``SET_OF_BAG``);
val _ = overload_on ("set", ``list$LIST_TO_SET``);
```

Defines:
- `card`, used in chunks 18, 25, 28–30, 33, 35–38, 41, 47, and 59.
- `disjoint`, used in chunks 25, 46, and 47.
- `dom`, used in chunks 17, 18, 22b, 28, 35, 47, 59, and 60.
- `filter`, used in chunks 12a, 33, 39, 46, 47, and 61a.
- `finite`, used in chunks 15a, 25, 41, 42a, 47, and 59.
- `image`, used in chunks 11, 33, 41, and 47.
- `nth`, used in chunks 35, 37, and 62b.
- `set`, used in chunks 10a, 11b, 17, 20, 25, 35, 42a, 47, 60, and 61.

Also, hide the `open` constant from the topology theory, which we don't need here:

57b ⟨*Make parsing/pretty-printing alterations* 57a⟩+≡ (56a) ◁ 57a

```
val _ = hide "open";
```

B.1 Auxiliary HOL Definitions

Define functions that should probably be in the standard HOL library.

57c ⟨*Auxiliary Definitions* 57c⟩≡ (56a) 59 ▷

  ⟨*Testing a bag of sets for disjointness* 58a⟩

A predicate on a *bag* of sets that is true when all of the sets are mutually disjoint:

58a ⟨*Testing a bag of sets for disjointness* 58a⟩≡ (57c) 58b▷

```
  val disjoint_sets_def = Define`
    disjoint_sets b =
      !s1 s2. BAG_IN s1 b /\ BAG_IN s2 (BAG_DIFF b {|s1|}) ==>
              DISJOINT s1 s2
  `;
```

Defines:
  disjoint_sets, used in chunks 25 and 58b.

Two obvious properties of disjoint_sets (for all that the second one is surprisingly painful to prove):

58b ⟨*Testing a bag of sets for disjointness* 58a⟩+≡ (57c) ◁58a

```
  val disjoint_sets_empty = store_thm(
    "disjoint_sets_empty",
    ``disjoint_sets {||} = T``,
    SRW_TAC [][disjoint_sets_def]);

  val disjoint_sets_insert = store_thm(
    "disjoint_sets_insert",
    ``disjoint_sets (BAG_INSERT s b) =
         (!s2. BAG_IN s2 b ==> DISJOINT s s2) /\
         disjoint_sets b``,
    SRW_TAC [][disjoint_sets_def, bagTheory.BAG_IN_BAG_INSERT,
               RIGHT_AND_OVER_OR, LEFT_AND_OVER_OR,
               DISJ_IMP_THM, FORALL_AND_THM] THEN
    EQ_TAC THEN REPEAT STRIP_TAC THENL [
      ASM_SIMP_TAC bool_ss [],
      FIRST_X_ASSUM MATCH_MP_TAC THEN ASM_REWRITE_TAC [] THEN
      `?b0. b = BAG_INSERT s1 b0`
         by PROVE_TAC [BAG_IN_BAG_DELETE, BAG_DELETE] THEN
      `BAG_INSERT s b = BAG_INSERT s1 (BAG_INSERT s b0)`
         by SRW_TAC [][BAG_INSERT_commutes] THEN
      FULL_SIMP_TAC (srw_ss()) [BAG_IN_BAG_INSERT],
      ASM_SIMP_TAC bool_ss [],
      `?b0. b = BAG_INSERT s1 b0`
         by PROVE_TAC [BAG_IN_BAG_DELETE, BAG_DELETE] THEN
      `BAG_INSERT s b = BAG_INSERT s1 (BAG_INSERT s b0)`
         by SRW_TAC [][BAG_INSERT_commutes] THEN
      FULL_SIMP_TAC (srw_ss()) [] THEN
      FIRST_X_ASSUM (Q.SPECL_THEN [`s1`,`s2`] MP_TAC) THEN
      FULL_SIMP_TAC (srw_ss()) [BAG_IN_BAG_INSERT] THEN
      PROVE_TAC [pred_setTheory.DISJOINT_SYM]
    ]);

  val _ = export_rewrites ["disjoint_sets_empty",
                           "disjoint_sets_insert"]
```

Uses disjoint_sets 58a.

Some properties of finite maps, and finite maps with bags as the range type. FRANGEB returns the range of a finite map as a bag. An element is in the bag n times if n keys in the map point at it.

59 ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁57c 60a▷

```
val FRANGEB_def = Define`
  FRANGEB fm x = card { k | k IN dom fm /\ (fm ' k = x) }
`;
val FRANGEB_THM = store_thm(
  "FRANGEB_THM",
  ``(FRANGEB FEMPTY = {||}) /\
    (FRANGEB (fm |+ (k, v)) = BAG_INSERT v (FRANGEB (fm \\ k)))``,
  SRW_TAC [][BAG_EXTENSION, FRANGEB_def, BAG_INN, BAG_INSERT] THENL [
    SRW_TAC [numSimps.ARITH_ss][],
    SIMP_TAC (srw_ss() ++ boolSimps.CONJ_ss) [DOMSUB_FAPPLY_THM] THEN
    `{k' | ((k' = k) \/ k' IN dom fm) /\ ((fm |+ (k, v)) ' k' = e)} =
     (if v = e then {k} else {}) UNION
     {k' | (k' IN dom fm /\ ~(k' = k)) /\ (fm ' k' = e)}`
       by (SRW_TAC [boolSimps.CONJ_ss, boolSimps.COND_elim_ss,
                    boolSimps.DNF_ss][EXTENSION, FAPPLY_FUPDATE_THM] THEN
           EQ_TAC THEN REPEAT STRIP_TAC THEN SRW_TAC [][]) THEN
    POP_ASSUM SUBST_ALL_TAC THEN
    COND_CASES_TAC THENL [
      SRW_TAC [][GSYM INSERT_SING_UNION] THEN
      AP_THM_TAC THEN AP_TERM_TAC THEN
      Q_TAC SUFF_TAC
            `finite {k' | (k' IN dom fm /\ ~(k' = k)) /\ (fm ' k' = e)}`
            THEN1 SRW_TAC [numSimps.ARITH_ss][] THEN
      SRW_TAC [][GSPEC_AND],
      SRW_TAC [][]
    ]
  ]);
val FINITE_FRANGEB = store_thm(
  "FINITE_FRANGEB",
  ``!fm. FINITE_BAG (FRANGEB fm)``,
  HO_MATCH_MP_TAC fmap_INDUCT THEN
  SRW_TAC [][FRANGEB_THM] THEN
  `fm \\ x = fm` by
     (SRW_TAC [][GSYM fmap_EQ_THM, DOMSUB_FAPPLY_THM] THEN
      PROVE_TAC [DELETE_NON_ELEMENT]) THEN
  SRW_TAC [][]);
val _ = export_rewrites ["FINITE_FRANGEB"]
```

Uses `card` 57a, `dom` 57a, and `finite` 57a.

Then, ranb is a function that merges all of the bags in a finite map's range to create one big bag.

60a ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁59 60b▷

```
  val ranb_def = Define`
    ranb fm = ITBAG BAG_UNION (FRANGEB fm) {||}
  `;
  val IN_RAN_FCOMP = prove(
  ``!g. x IN ran (f o_f g) = ?y. (x = f y) /\ y IN ran g``,
   HO_MATCH_MP_TAC fmap_INDUCT THEN SRW_TAC [][] THEN
   EQ_TAC THENL [
     Cases_on `x = f y` THENL [
       METIS_TAC [],
       ASM_SIMP_TAC (srw_ss()) [DOMSUB_NOT_IN_DOM] THEN METIS_TAC []
     ],
     ASM_SIMP_TAC (srw_ss()) [DOMSUB_NOT_IN_DOM] THEN METIS_TAC []
  ])

  val IN_RAN = prove(
    ``x IN ran f = ?y. y IN dom f /\ (x = f ' y)``,
    SRW_TAC [][FRANGE_DEF] THEN PROVE_TAC []);
  val BOOL_UNIV = store_thm(
    "BOOL_UNIV",
    ``UNIV = {T; F}``,
    SRW_TAC [][EXTENSION]);
  val _ = export_rewrites ["BOOL_UNIV"]
```

Defines:
 ranb, used in chunk 16a.
Uses dom 57a.

The function fmrangeset takes a finite map into bags, and returns the union of all of the elements mapped to.

60b ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁60a 61a▷

```
  val fmrangeset_def = Define`
    fmrangeset (fm:'a |-> 'b bag) =
        { x | ?c. c IN dom fm /\ x IN set (fm ' c) }
  `;
  val _ = overload_on("ran", ``fmrangeset``);
```

Defines:
 fmrangeset, never used.
Uses dom 57a and set 57a.

To partition a bag into a set of bags, according to an equivalence relation R, use the function partitionb. Note that one need only produce a *set* of bags because a particular partition element could hardly occur twice.

61a ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁60b 61b▷
```
val partitionb_def = Define`
  partitionb (R : 'a -> 'a -> bool) (b : 'a bag) : 'a bag set =
    pred_set$IMAGE (\s. filter s b) (partition R (set(b)))
`;
```
Defines:
  partitionb, used in chunks 18 and 39.
Uses filter 57a and set 57a.

All finite sets of real numbers have maximum elements. The function RMAX_SET returns this maximum element.

61b ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁61a 61c▷
```
val RMAX_SET_lemma = prove(
  ``!s : real set. FINITE s ==> ~(s = {}) ==>
                 ?x. x IN s /\ !y. y IN s ==> y <= x``,
  HO_MATCH_MP_TAC FINITE_INDUCT THEN
  SIMP_TAC bool_ss [NOT_INSERT_EMPTY, IN_INSERT] THEN
  REPEAT STRIP_TAC THEN
  Q.ISPEC_THEN `s` STRIP_ASSUME_TAC SET_CASES THENL [
    ASM_SIMP_TAC (srw_ss()) [],
    `?m. m IN s /\ !y. y IN s ==> y <= m`
      by PROVE_TAC [NOT_INSERT_EMPTY] THEN
    Cases_on `e <= m` THENL [
      PROVE_TAC [],
      `m <= e` by PROVE_TAC [REAL_NOT_LE, REAL_LE_LT] THEN
      PROVE_TAC [REAL_LE_REFL, REAL_LE_TRANS]
    ]
  ]);
val RMAX_SET_DEF = new_specification(
  "RMAX_SET_DEF", ["RMAX_SET"],
  SIMP_RULE bool_ss [AND_IMP_INTRO, GSYM RIGHT_EXISTS_IMP_THM,
                     SKOLEM_THM] RMAX_SET_lemma)
val _ = overload_on ("MAX_SET", ``RMAX_SET``)
```
Defines:
  RMAX_SET, never used.
Uses set 57a.

Take the natural number truncation of a real with real_to_num_trunc:

61c ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁61b 62a▷
```
val real_to_num_trunc_def = Define`
  real_to_num_trunc (r:real) = (LEAST n. r < &n) - 1
`;
val _ = overload_on ("trunc", ``real_to_num_trunc``);
```

And some properties of the same:

62a ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁61c 62b▷

```
val real_to_num_trunc_thm = store_thm(
  "real_to_num_trunc_thm",
  ``real_to_num_trunc (&n) = n``,
  SRW_TAC [][real_to_num_trunc_def] THEN
  CONV_TAC (UNBETA_CONV ``LEAST x. n < x``) THEN
  MATCH_MP_TAC LEAST_ELIM THEN SRW_TAC [][] THENL [
    Q.EXISTS_TAC `n + 1` THEN SRW_TAC [numSimps.ARITH_ss][],
    SPOSE_NOT_THEN ASSUME_TAC THEN
    `n < n' - 1` by DECIDE_TAC THEN
    `~(n' - 1 < n')` by PROVE_TAC [] THEN
    FULL_SIMP_TAC (srw_ss() ++ numSimps.ARITH_ss) []
  ]);
```

Lists can be sorted by a relation. In order to define what it is to be sorted, it's necessary to define what it is for two lists to be permutations of each other, and what it is for a list to be sorted with respect to a relation.

62b ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁62a 63▷

```
val permutation_def = Define`
  permutation l1 l2 = (LIST_TO_BAG l1 = LIST_TO_BAG l2)
`;
val sorted_def = Define`
  sorted R l = !m n. m < n /\ n < LENGTH l ==> R (nth m l) (nth n l)
`;
val sort_def = Define`
  sort R l = @l'. permutation l l' /\ sorted R l'
`;
```

Defines:
  `sort`, used in chunk 39.
Uses `nth` 57a.

B.2 Auxiliary Tactics

A tactic for moving lets to the top of a term and then moving into the assumptions as abbreviations.

63 ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁62b 64a▷

```
val LET_FORALL = prove(``LET f v = !x. (v = x) ==> f x``, SRW_TAC [][]);
val LET_RAND' = prove(``P (LET f v) = LET (P o f) v``, SRW_TAC [][])
val LET_RATOR' = prove(
  ``(LET f v) x = LET (combin$C f x) v``,
  SRW_TAC [][combinTheory.C_THM]);

val o_ABS_R = prove(
  ``f o (\x. g x) = (\x. f (g x))``,
  SRW_TAC [][FUN_EQ_THM])
val o_UNCURRY_R = prove(
  ``f o UNCURRY g = UNCURRY ((o) f o g)``,
  SRW_TAC [][pairTheory.UNCURRY, FUN_EQ_THM]);
val C_ABS_L = prove(
  ``combin$C (\x. f x) y = (\x. f x y)``,
  SRW_TAC [][FUN_EQ_THM])
val C_UNCURRY_L = prove(
  ``combin$C (UNCURRY f) x = UNCURRY (combin$C (combin$C o f) x)``,
  SRW_TAC [][FUN_EQ_THM, pairTheory.UNCURRY])
val RAISE_LETS_CONV =
    SIMP_CONV bool_ss [o_ABS_R, o_UNCURRY_R, C_ABS_L, C_UNCURRY_L,
                       LET_RATOR', LET_RAND']
fun ELIM_LET t = if is_let t andalso is_var (rand t) then
                   (REWR_CONV LET_THM THENC BETA_CONV) t
                 else NO_CONV t
val RAISE_LETS_TAC = CONV_TAC RAISE_LETS_CONV THEN
                     REPEAT (CONV_TAC (HO_REWR_CONV LET_FORALL) THEN
                             GEN_TAC THEN
                             DISCH_THEN (fn th => REWRITE_TAC [th] THEN
                                                  ASSUME_TAC th) THEN
                             CONV_TAC (DEPTH_CONV ELIM_LET))
```

A tactic to remove some record fields, and replace them with more convenient names.

64a ⟨*Auxiliary Definitions* 57c⟩+≡ (56a) ◁63
```
open bagTheory
val std_state_abbrev =
  Q.ABBREV_TAC ‘N = s.num_to_elect‘ THEN
  Q.ABBREV_TAC ‘dead = s.dead‘ THEN
  Q.ABBREV_TAC ‘votes = s.votes‘ THEN
  Q.ABBREV_TAC ‘excl = s.excluded‘ THEN
  Q.ABBREV_TAC ‘dones = s.done_successes‘ THEN
  Q.ABBREV_TAC ‘Q = s.quota‘ THEN
  NTAC 6 (POP_ASSUM (K ALL_TAC))
```
Defines:
`std_state_abbrev`, never used.

B.3 Finishing

To close, it is only necessary to call `export_theory`.

64b ⟨*Closing* 64b⟩≡ (5b)
```
val _ = export_theory()
```

Appendix C

Chunk Index

All of the document's chunks are listed here.

Appendix D

Identifier Index

All of the HOL definitions are indexed here. An underlined reference indicates where a definition is made. Other references are to chunks where the term is used.