Verified Checking of Certified Vote Counting Computations

Abstract

We approach computing an election's result as a sequence of logical rule applications applied to stages of the count, together with a certificate for the computation of the end result. A certificate for the election, then, is a visualisation of the trace of rule applications applied to an initial stage of the count to reach a final stage, where winners are announced. Verifying the certificate of the count amounts to checking that transitions between stages by means of rule applications have occurred correctly.

We verify the certificate produced for computing the outcome of elections which are based on a Single Transferable Vote scheme with fractional values. For this, we formalise the election's protocol as logical rules inside the theorem prover HOL4. Then, we obtain a verified checker for validating a certificate of such an election. By means of a parser defined in HOL4, we translate data of the certificate into the data structure of the formalised election. Consequently, we would finally extract a CakeML program which checks the certificate for correctness.

Keywords keyword1, keyword2, keyword3

ACM Reference Format:
. . Verified Checking of Certified Vote Counting Computations. In *Proceedings of* . ACM, New York, NY, USA, 7 pages. https://doi.org/

1 Introduction

Elections stand at the centre of a democracy, where people cast their preferences to elect those whom they perceive as fittest. Therefore, establishing public's trust in authenticity of the announced result and providing them with the means for independent verification of this result is crucial. Universal verifiability is a measure for evaluation of the degree in which an election count can be examined by any member of the general public. Unfortunately, the popular methods for counting election ballots do not even come close to satisfaction of this factor. Usually, election committees invite members of the parties and public to observe the process in order to witness trustworthiness of the final result. Moreover, scrutiny sheets are published to provide detailed evidence as to how the counting has been processed.

However in practice, scrutiny sheets are deficient. For example costly mistakes have occurred in an election for Senate in Western Australia. Because of such flaws in hand counting an election, the heavy cost which they have along with time consumption of this method, computers have recently been employed to perform the task. But these programs merely output the final result of the election without specifying the process of the computation. Moreover, the source code of these program is kept secret under the "commercial in confidence"excuse. Unfortunately, since there are corner cases in complex protocols such as Single Transferable Vote scheme which the legislation leaves open to interpretation, it is difficult for an independent evaluator to design a program that always returns the result of these commercial programs. Therefore, there is no efficient means for not only an ordinary member of the public, but also experts of the field to convince themselves about correctness of the outcome. Hence, quality indexes such as universal verifiability remain inadequately addressed.

Our approach offers a remedial for this situation. We understand the counting process as a certified computation which formalises the count as a sequence of logical rule applications to stages of the count, together with a certificate that is a visualisation of the trace of rule applications to the ballots cast in the election. Stages of the count, are formalised as typing judgements which are of two kind; non-final and final. The former is constituted of six components that put together provide all the necessary information to know what the current status of the count is. It informs the scrutineers of the ballots which (possible) need to be counted, tally of each candidate, pile of the votes allocated to each candidate according to the preferences, list of candidates whose votes may be distributed later, and list of elected and continuing candidates at this stage. The latter judgement is a terminal stage of the process whereby winners of the election are announced.

The election protocol implicitly specifies steps for advancing the counting process. These steps which tell us when and how to move from one stage of the count (judgement) to another are formalised as logical rules. These rules have side conditions that must be met before such transitions can happen. Side conditions are, essentially, the formal counterpart of the protocol laid down by the legislation. By satisfying these side conditions, one makes sure that the process has proceeded in accordance to the protocol of the election. Moreover, it becomes possible to output, along announcement of the winners, a trace of rule applications specifying as to how this result is obtained. A visualisation of this trace is called the certificate for the count. Such a certificate has the advantage of providing thorough detail of the counting process. The certificate is independent of the means used to

, ,
.

produce it, so that one does not need to invest trust in the election committee, the program which output it, and possible intervening malfunctions of hardware. The certificate is independently checkable for correctness by various tools, which the scrutineers have trust in. The side conditions of rules enforce unique applicability of the rules. Therefore, to check the certificate for correctness, one merely needs to verify that given a judgement visualised in the certificate, the transition occurred to reach to the next judgement in the certificate accords with one of the rules.

This paper specialises the above generic approach to vote counting, for a specific Single Transferable Vote scheme called ANU-Union. We present a verified checker for certificates produced for elections which are based on the ANU-Union scheme. In our past work, we formalised the ANU-Union protocol as a system of logical rules, and proved some properties of the specification. The formalised rules put together determine how to count ballots, elect a candidate as one of the winners, transfer surplus votes, eliminate weak candidates from the election, and eventually terminate the whole election by announcing all of the winners. A Haskell program capable of computing real elections was extracted. This program, for any given input ballots, number of vacancies and a list of competing candidates, produces a certificate, such a the one in **??**, which is a visualisation of a sequence of applications of those rules for obtaining the output. However, to check these certificates, we do not rely on any such formalisation or implementation of the ANU-Union scheme. Our mere assumption is that the certificate given, is claimed to be an output of a computational process which realises the ANU-Union counting protocol as the set of logical rules specified in the protocol. The task, then, is to verify authenticity of a certificate against the protocol directly rather than by referring to any previous formalisation of the ANU-Union scheme.

For checking task, we formalise the ANU-Union STV scheme in HOL4. The scheme is captured as six independent predicates that are the formal specification of the protocol as rules of counting. The predicates themselves, are conjuncts of some stand-alone sub-predicates. These sub-predicates which match against clauses of the protocol, are the logical specification of auxiliary computational functions that are guaranteed to meet expectations of their specification. For each counting rule defined as a predicate, a computational twin is given. This computational counterpart is basically the conjunction of the auxiliary computational functions mentioned above. Therefore, each rule as a predicate is proven equivalent to the rule captured as a computational entity. Finally, the predicative checker is defined to be a disjunction over predicative counting rules and the computational checker is, also, formalised as disjunction of the computational counting rules. Since components of the predicative and computational checker are shown to be equivalent, we achieve perfect match between the logical and functional checker as well.

2 The Protocol and its Logical Specification

First, we lay down the counting algorithm of the ANU-Union. Then for the ease of communication and pedagogical intentions, we give a mathematical specification of the protocol. However, we make no essential use of this to capture the protocol later as formalised rules in HOL4.

The ANU-Union STV is a scheme which is very close to the STV used for the Australian Senate election. It has three distinctive characteristics.

Step-by-step surplus transfer. Surplus votes of already elected candidates, who are awaiting for their surplus to be transferred, are dealt with, one at a time, in order of first preferences.

Electing after each transfer. After each transfer of values, candidates that reach the quota are elected immediately.

Fractional transfer. The value of vote transfer is a fractional number determined by a specific formula.

The protocol proceeds as follows.

1. decide which ballots are *formal*.
2. determine what the quota exactly is.
3. count the first preference for each *formal* ballot paper and place the vote in the pile of the votes of the preferred candidate.
4. if there are vacancies, any candidate that reaches the quota is declared elected.
5. if all the vacancies have been filled, counting terminates and the result is announced.
6. if the number of vacancies exceeds the number of continuing candidates, all of them are declared elected and the result is announced.
7. if there are still vacancies and all ballots are counted, and there is an elected candidate with surplus, go to step 8 otherwise go to step 9.
8. In case of surplus votes, transfer them to the next continuing preference appearing on each of those votes at a fractional value according to the following formula:

$$\text{new value} = \frac{\text{number of votes of elected candidate} - \text{quota}}{\text{number of votes of elected candidate}} \tag{1}$$

   Subsequent transfer values are computed as the product of the current value with previous transfer value.

9. if there are still vacancies and all ballots are counted, and all surplus votes are transferred, choose the candidate with the least amount of votes and exclude them from the list of continuing candidates. Also transfer all of their votes according to the next preference appearing on each of the votes in his pile. The transfer value of the ballots shall remain unchanged.

10. if there is more than one elected candidate, first transfer the surplus of the candidate who has the largest surplus. If after a transfer of surplus, a continuing candidate exceeds the quota, declare them elected and transfer their surplus, only after all of the earlier elected candidates' surpluses have been dealt with.
11. at transfer stage, candidates who are already elected or eliminated receive no vote.

2.1 The Logical specification

The count is constituted of some key components, all of which appear in a comprehensive certificate for the counting process:

1. candidates competing in the election
2. number of vacancies
3. quota of the election
4. ballots consisting of a list of candidates to be ranked and a fractional value of the ballot
5. stages of the counting (or computation)

At every non-final stage of the counting, we need to know that if there are ballots to be counted, how the votes have been distributed up to now, what is the tally of each candidate, and if any candidate was elected or eliminated from the election. Therefore, for stages of the count to thoroughly inform the scrutineers of how the situation of the count is at the moment, it is necessary and sufficient to contain the following components:

1. the (possibly) uncounted ballots
2. a group of candidates called *elected* candidates
3. a group of candidates called *continuing* candidates
4. a group of candidates already elected who have exceeded the quota
5. the tally of votes of each candidate
6. the set of ballots that have been counted for each individual candidate.

In order to specify the protocol in a precise formal language, we introduce symbols that characterise these integral parts of the computation. Below is the list of mathematical symbols together with the intended concepts that they represent.

C	a set of candidates
$\mathbb{Q}$	the set of rational numbers
$\mathsf{List}(C)$	the set of all possible list of candidates
$\mathsf{List}(C) \times \mathbb{Q}$	the set of all (possible) ballots
B	shorthand for $\mathsf{List}(C) \times \mathbb{Q}$
A	initial list of all of the candidates
st	initial number of vacancies
bs	initial list of ballots cast to be counted
bl	backlog of elected candidates
b, d	to represent a ballot
ba, ba'	list of ballots
ba_ϵ	empty list of ballots
c, c'	to represent a candidate
t, nt, t'	tally function, from C into $\mathbb{Q}$
p, np	pile function, from C into B
e, ne	list of elected candidates so far
$[]$	representing empty list of candidates
$l_1{+}{+}l_2$	list l_2 is appended to the end of list l_1
h, nh	list of continuing candidates in the election
qu	quota of the election as a rational number

Here a ballot $b \in B$ has two parts: one part is a list of candidates and the other is the value that the ballot has. So a ballot b is a pair (l, q), for some $l \in \mathsf{List}\ (C)$ and a number $q \in \mathbb{Q}$. The character $ba \in B$, is reserved to show the set of ballots which require to be counted in each single state of the count (ba for "ballots requiring attention"). The tallying operation done in a hand-count election is formalised by function t. Item 9 above is expressed by the function pile p. At each stage of the count for any candidate c, $p(c)$ determines which votes are given to the candidate c. The list $bl \in \mathsf{List}(C)$ which is the backlog, is the list of already elected candidates whose votes are yet to be transferred. The notation e (for "elected") and h (for "hopeful", as c already represents candidates) respectively represent the list of elected and continuing candidates at each stage.

We should note that the use of lists, instead of sets, for ballots, and continuing or elected candidates is simply for the convenience of formalisation in a theorem prover. But the counting rules ,defined shortly afterwards, make no essential use of this representation.

Moreover, we must encapsulate the concept of stages of the count mathematically. For this end, we use two kind of judgements, called non-final and final judgements.

Non-final Judgement. $bs, st, A \vdash \mathsf{state}(ba, t, p, bl, e, h)$:
In an election, assuming we have an initial list of ballots bs, initial number of vacancies st, and a list A of all candidates competing in the election, $\mathsf{state}(ba, t, p, bl, e, h)$ is an intermediate stage of the computation, where ba is the list of uncounted ballots at this point, for a candidate c, $t(c)$ is the tally recording the number of votes c has received up to this point, $p(c)$ computes the pile of votes for the candidate c, bl is the list of elected whose surplus have not yet been transferred, e is the list of elected candidates by this point, and h is the list of continuing candidates up to this stage.

Final Judgement. $bs, st, A \vdash \mathsf{winners}(w)$:
In an election, assuming we have an initial list of ballots bs, initial number of vacancies st, and a list A of all candidates competing in the election, $\mathsf{winners}(w)$ is a final stage of the computation, where w is the final list consisting of all of the declared elected candidates.

The formalised protocol has six rules which specify how count ballots, when to elect a candidate as one of the winners, who to eliminate from the election, and where to terminate the count by declaring all of the winners. We begin by the rule *count* which determines when and how to count ballots.

Definition 2.1 (count)**.** Suppose for $ba \in$List(B), $t : C \rightarrow \mathbb{Q}$, $p : C \rightarrow$List(B), $bl \in$List(C), $e, h \in$List(C), and the non-final judgement state(ba, t, p, bl, e, h) is the current stage of the computation. Then we can move to state$(ba_\epsilon, nt, np, bl, e, h)$ as the next stage of the computation if the conditions below are met.

$$\frac{\text{state}(ba, t, p, bl, e, h)}{\text{state}(ba_\epsilon, nt, np, bl, e, h)}\ count$$

1. ba is not empty, i.e. $ba \neq []$
2. if a candidate c is not continuing, then c's pile and tally remain the same,i.e. $\forall c \notin h$, $np(c) = p(c)$ and $nt(c) = t(c)$
3. if a candidate c is continuing, then find all of the ballots which have c as their first continuing preference and put them in the pile of c, i.e. $\forall c \in h$, $np(c) = p(c)$++ l_c, and $nt(c)$ equals to the sum of values of the ballots in the updated pile

As the definition above shows, counting of ballots happens according to *first continuing preferences*. This means that in the first component of each ballots, which is a list, we look for the candidate who is neither elected nor eliminated and their name precedes all the other whose name appears in the list part of the ballot. Then we count such a ballot for this candidate.

When counting the first preferences is dealt with, we elect all of those candidates who have reached the quota and announce them as elected. Subsequently, a new fractional value is computed according to formula (1) for the each of these candidates' surpluses. The candidates are removed from the list of continuing candidates and appended to the backlog in order of the amount of their tallies.

Definition 2.2 (elect)**.** Assume state $(ba_\epsilon, t, p, bl, e, h)$ is a judgement and ba_ϵ is the empty list of ballots. Then we have the following rule whenever there exists $l \in$List (C), a list of candidates to be elected, such that each of the conditions below hold:

$$\frac{\text{state }(ba_\epsilon, t, p, bl, e, h)}{\text{state }(ba_\epsilon, t, np, nbl, ne, nh)}\ elect$$

1. length of the list l is less than or equal to st − length(e) (there are enough vacant seats)
2. every candidate in the list l has reached (or exceeded) the quota qu
3. the list l is ordered with respect to the number of votes each elected candidate (whose name appears in l) has received.
4. the updated list of elected candidates ne, contains every already elected candidates (in e) plus the ones appearing in the list l
5. the updated list nh has every continuing candidate whose name is in h, except those whose name also exists in l
6. nbl equals to bl appended by the list l,i.e. $nbl = bl$ ++ l
7. if a candidate c is not in the list l, then pile of c is kept the same,i.e. $\forall c \notin l, np(c) = p(c)$
8. if a candidate c is in l, then update their pile by keeping the votes already attributed to them, but changing the value of those votes to a new fractional value according to formula (1).

If the number of the elected candidates equals to vacancies, then the computation terminates and all of the winners are declared by the rule *ewin*.

Definition 2.3 (ewin)**.** Let state (ba, t, p, bl, e, h) be a stage of the computation. The inference rule *ewin* asserts that winners (e) is the next judgement, provided that length (e) = st.

$$\frac{\text{state }(ba, t, p, bl, e, h)}{\text{winners }(e)}\ ewin$$

In case where the number of elected candidates and those who are still continuing put together does not exceeds the quota, then we announce all of them as winners and finish the count.

Definition 2.4 (hwin)**.** If state (ba, t, p, bl, e, h) is a judgement and (length (e) + length $(h) \leq$ st), then we can transit to the stage winners $(e$ ++ $h)$.

$$\frac{\text{state }(ba, t, p, bl, e, h)}{\text{winners }(e \text{ ++ } h)}\ hwin$$

We may reach a stage of the computation where there is no ballot left to count, no one has exceeded the quota, the number of elected candidates is strictly less than the vacancies, and there is at least one candidate who has already been elected but their surplus votes awaits to be distributed. Then the rule *transfer* takes the first candidate in the backlog out and places their votes into the list of ballots to be dealt with later.

Definition 2.5 (transfer)**.** Suppose state $(ba_\epsilon, t, p, bl, e, h)$ is the current judgement. Then the rule *transfer* asserts state (nba, t, np, nbl, e, h) is the judgement we reach to:

$$\frac{\text{state }(ba_\epsilon, t, p, bl, e, h)}{\text{state }(nba, t, np, nbl, e, h)}\ transfer$$

and the side conditions for applying the rule are

1. there are still seats to fill, i.e. length $(e) < st$
2. no candidate has reached the quota,i.e. $\forall c', c' \in h \rightarrow (t(c) < qu)$
3. there exist a list $l \in$List(C) and a candidate c' such that

3.1 c' is the first candidate in the backlog and l is the tail of bl, i.e. $bl = c' :: l$

3.2 remove c' from the backlog bl and update it, i.e. $nbl = l$

3.3 move the votes in the pile of c' to the list of uncounted ballots, $nba = p(c')$
3.4 empty the pile of c', i.e. $np(c') = ba_\epsilon$
3.5 do not tamper with pile of candidates other than c', i.e. $\forall c'', c'' \neq c' \to np(c'') = p(c'')$.

Finally, it is possible that we obtain a situation where one candidate has to be eliminated from the counting process. The rule *elim* specifies when and how to proceed in such conditions.

Definition 2.6 (elim)**.** Suppose state $(ba_\epsilon, t, p, [], e, h)$ is the current stage of computation. If st < (length (e) + length (h)), and no candidate has reached the quota then subject to below conditions by the rule *elim*, we move to the judgement state $(nba, t, np, [], e, h)$.

$$\frac{\text{state } (ba_\epsilon, t, p, [], e, h)}{\text{state } (nba, t, np, [], e, h)} \; elim$$

1. All continuing candidates are below the quota
2. there exists a weakest candidate c' such that

2.1 other continuing candidates have strictly more votes than c'
2.2 exclude c' from current continuing list of candidates (namely h) and update it to nh
2.3 remove the ballots in the pile of c' without changing the value of those ballots and put them in the list of uncounted ballots, i.e. $nba = p(c')$ and $np(c') = ba_\epsilon$.
2.4 do not tamper with the pile of other candidates, i.e. $\forall c'', c'' \neq c' \to np(c'') = p(c'')$

Remark 1. Note that in the definition of *elim*, it does not specify how to decide on exclusion of some candidates whose tallies are tied at the least amount. Therefore, the specification of *elim* allows for any tie breaking method between candidates as long as their tallies are equal and less than other continuing candidates.

3 The Formalisation in HOL4

Figure 1 depicts the formalisation of judgements in HOL4. Every component of each constructor of the data type judgement is of type αlist, for some particular α. The advantage of such formalisation is that it makes judgements to be an instance of equality type class. This, in turn, makes it easier to formalise rules and reason about them inside HOL4. Moreover, it leaves no gap between formalisation of rules and checking certificates for correctness. What understandably appears in a certificate, are some association lists of ballots, candidates, tally of candidates, and pile of those candidates. Therefore for the task of certificate verification, one has to only deal with association lists.

The type of tallies is `(Cand # rat) list`, where `Cand` is the type of candidates and `rat` is the HOL4 native type for fractional numbers. Also the type of piles is `(Cand #(((Cand list)# rat) list)` so that a pile p is a list of pair values which are candidates and the ballots attributed to them.

```
val _ = Hol_datatype `judgement =
  state of
   ((Cand list) # rat) list
   # (Cand # rat) list
   # (Cand # (((Cand list) # rat) list)) list
   # Cand list
   # Cand list
   # Cand list
  | winners of (Cand list) `;
```

Figure 1. judgement data type

```
val NO_DUP_PRED = Define `
 (NO_DUP_PRED h (c: Cand) =
    (h = []) ∨ (¬ MEM c h)
  ∨ (∃ h1 h2. (h = h1 ++ [c]++ h2)
         ∧ (¬ MEM c h1)
         ∧ (¬ MEM c h2))) `;
```

Figure 2. predicate for duplication-free list of candidates

For formalising rules of the count, we need to define auxiliary predicates. Some of these predicates are explicit formalisations of the side conditions given for rules in the last section. However, some predicates capture implicit parts of the protocol. For applying a rule correctly, one has to make sure that there are, for example, no duplicate members in any of the lists that occur as a component of `state` or `winners`. As another example, all of the candidates whose name appears in the list of elected or continuing candidates, or as the first component of tally lists and piles, should also be in the initial list of competing candidates. The mathematical representation given earlier hides such subtleties, simply because tallies and piles are formulated as functions operating on the list of all of candidates, hence automatically covering such concerns. Besides, from the perspective of certificate verification, it could very well be the case that someone has illegally duplicated name of a candidate in, for instance, list of elected candidates at some stage of the computation. A through checker should certify that such malicious certificates are detected and declare them invalid.

Figure 2 shows the definition of a predicate that asserts when a list of candidates is duplicate-free. We need preconditions on tally and pile lists as well. Figure 3 contains the predicate `Valid_PileTally` which will be used to enforce tally and pile lists to be lists of pairs where if one *maps* over the first value of each pair, the resulting list is exactly the initial list of competing candidates. Note that we assume HOL4 built-in functions/predicates such as `MAP` and `FST` as given. The predicate `Valid_Init_CandList` in Figure3, illustrates the requirement imposed on the initial

```
val Valid_PileTally = Define `
  Valid_PileTally t (l: Cand list) =
   (∀c. (MEM c l) <=> (MEM c (MAP FST t)))`;

val Valid_Init_CandList = Define `
  Valid_Init_CandList (l: Cand list) =
   ((l <> []) ∧ (∀ c. NO_DUP_PRED l c)) `;
```

Figure 3. A predicate for partial validation of tallies/piles

```
val no_dup = Define
  `(no_dup [] = T)
  (no_dup (h::t) = (if (not_elem h t)
                 then (no_dup t) else F)) `;

val Valid_PileTally_DEC1_def = Define `
 (Valid_PileTally_DEC1 [] (l: Cand list)= T)
∧ (Valid_PileTally_DEC1 (h::t) l =
   (MEM (FST h) l)
 ∧ (Valid_PileTally_DEC1 t l))`;

val Valid_PileTally_DEC2_def = Define `
(Valid_PileTally_DEC2 t ([]: Cand list)= T)
∧ (Valid_PileTally_DEC2 t (l0::ls)=
   if (MEM l0 (MAP FST t)) then
   (Valid_PileTally_DEC2 t ls) else F) `;
```

Figure 4. Some auxiliary functions

list of candidates running in the election to be non-empty and duplicate-free.

Each of the above predicates have their corresponding computational definitions, some of which are illustrated in Figure 4. They compute an output for a given input object. Definitions of the predicates presented above, are meant as logical specification of these computational functions. We prove the equivalence of these predicates with the computational counterparts inside HOL4.

Theorem 3.1. *For all lists of candidates h, if for all candidates c, the predicate stating that h has no duplicate with respect to c holds, then computationally the list h has no duplicate,i.e.* `∀h.(∀c. (NO_DUP_PRED h c)) ==> (no_dup h)` *Moreover, if the list h is computationally duplicate-free, then so it is predicatively,i.e.* `∀h ∀c. (no_dup h) ==> (NO_DUP_PRED h c))`

Theorem 3.2. *Suppose l is a list of candidates and t is a list of pairs where the first value of each pair is a candidate. If predicatively each of the candidates appearing as a first component of a pair in t, also belongs to l, then, so it does computationally, i.e.* `∀l t. (∀c. (MEM c (MAP FST t)) ==> (MEM c l)) ==> (Valid_PileTally_DEC1 t l).` *The reverse direction holds as well,i.e.* `∀l t. (Valid_PileTally_DEC1 t l) ==> (∀c. (MEM c (MAP FST t)) ==> (MEM c l)).`

Now we can state predicates which are the formal counterpart of the protocol's counting rules. For the sake of space limitation, we only present the formalisation of the rule *elim* in detail and refer the reader to the source code included in the paper. As it is shown in Figure 5, the predicate for the rule *elim*, is constituted of the implicit predicates laid down above and some other conditions which are specific constraints for applying the rule *elim*. The predicate `elim_cand_def` assumes some parameters which are `st`, `qu`, `l`, `c`, `j1`, and `j2` standing for the number of vacancies, quota of the election, list of all of the competing candidates, the premise and conclusion judgements, respectively. We have numbered each

```
val elim_cand_def = Define `(elim_cand
  st (qu:rat)(l:Cand list)(c: Cand) j1 j2)=
  (∃ p e h nh nba np.
   (j1 = state ([], t, p, [], e, h))
    1. ∧ Valid_Init_CandList l
    2. ∧ (∀c'. MEM c' (h++e) ==> (MEM c' l))
    3. ∧ (∀c'. NO_DUP_PRED (h++e) c')
    4. ∧ (Valid_PileTally p l)
    5. ∧ (Valid_PileTally np l)
    6. ∧ (LENGTH (e ++ h) > st)
    7. ∧ (LENGTH e < st)
    8. ∧ (∀c'. NO_DUP_PRED (MAP FST t) c')
    9. ∧ (Valid_PileTally t l)
   10. ∧ (∀c'.(MEM c' h ==> (∃x. MEM (c',x) t
                          ∧ (x < qu))))
   11. ∧ (MEM c h)
   12. ∧ (∀d. (MEM d h ==> (∃x y.(MEM (c,x) t)
                  ∧ (MEM (d,y) t) ∧ (x <= y))))
   13. ∧ (eqe c nh h)
   14. ∧ (nba = get_cand_pile c p)
   15. ∧ (MEM (c,[]) np)
   16. ∧ (∀d'. ((d' <> c) ==> (!l.
          (MEM (d',l) p ==> MEM (d',l) np)
        ∧ (MEM (d',l) np ==> MEM (d',l) p))))
    ∧ (j2 = state (nba, t, np, [], e, nh)))`;
```

Figure 5. The predicate for the *elim* rule

constraint in the Figure 5 to address them more efficiently and clearly. We explain how the items collectively match with the clauses of the definition 2.6. As the definition 2.6 asserts in the clause 2, in order to apply the rule *elim* legitimately, no continuing candidate should have reached the quota. Item 10 above along with number 1 and 9 capture this clause of the definition 2.6. Item 10 states that for every continuing candidate c', there exists a fractional value x where the pair (c', x) belongs to the tally list. By number 9, and theorem 3.2, c' belongs to the initial list of competing candidates l. Number 1 postulates that l has no duplicate element. Therefore, if there was some other fractional value x' for which (c', x) belonged to l but $x \neq x'$, then we would obtain duplication of c' in the list l. That is a contradiction. Hence, collectively, items 1, 9 and 10 enforce the existential x in 10 to match exactly with the tally amount of the candidate c'. All of the above reasoning has been formalised and proven correct in HOL4. In the same manner, one can see how the predicate `elim_cand_def` and the definition 2.6 for the *elim* match against each other.

Figure 6 is an illustration of the computational twin of the predicate `elim_cand_def`. This computational definition is composed of functions each of which are equivalent to either one exact item of the Figure 5, or a conjunction of some of them. Therefore, as we have proved, the computational `elim_cand_dec` is equivalent to the predicate `elim_cand_def`. The computational function `remove_one_cand` is specified against the predicate `eqe` given in Figure 5. The property `eqe` states when two lists $l1$ and $l2$ are exactly the same except for one element. The function `remove_one_cand` takes a list l and a candidate c and removes one occurrence of c from the list l.

Theorem 3.3. *The following properties prove that the function* `remove_one_cand` *meets its specification, which is* `eqe`*.*

```
∀h1 h2 (c: Cand). (MEM c h2) ∧ (eqe c h1 h2)
            ==> (h1 = remove_one_cand c h2)

∀h (c: Cand). (MEM c h) ∧ (NO_DUP_PRED h c)
           ==> (eqe c (remove_one_cand c h) h)
```

```
val Elim_cand_dec_def = Define `
(Elim_cand_dec st qu l c (j, winners w)= F)
∧ (Elim_cand_dec st qu l c (winners w,j)= F)
∧ (Elim_cand_dec st qu l c
      (state (ba,t,p,bl,e,h),
          state (ba',t',p',bl',e',h')) =
            ((empty_list ba)
          ∧ (empty_list bl)
          ∧ (t = t') ∧ (bl = bl') ∧ (e = e')
          ∧ (LENGTH (e ++ h) > st)
          ∧ (LENGTH e < st)
          ∧ (non_empty l) ∧ (no_dup l)
          ∧ (list_MEM (h++e) l)
          ∧ (no_dup (h++e))
          ∧ (Valid_PileTally_DEC1 p l)
          ∧ (Valid_PileTally_DEC2 p l)
          ∧ (Valid_PileTally_DEC1 p' l)
          ∧ (Valid_PileTally_DEC2 p' l)
          ∧ no_dup (MAP FST t)
          ∧ (Valid_PileTally_DEC1 t l)
          ∧ (Valid_PileTally_DEC2 t l)
          ∧ (MEM c h)
          ∧ (less_than_quota qu h t)
          ∧ (h' = remove_one_cand c h)
          ∧ (bigger_than_cand c t h)
          ∧ (ba' = get_cand_pile c p)
          ∧ (MEM (c,[]) p')
          ∧ (subpile1 c p p')
          ∧ (subpile2 c p' p)))`;
```

Figure 6. The computational *elim* rule

Without providing details, we explain about some of the other functions appearing in Figure 6. For a list h of candidates, the function `less_than_quota` checks if all of the elements of h have a tally which is less than or equal to quota qu. This function is specified by the clause 10 of the Figure 5. Also the function `bigger_than_cand`, which is captured by help of clause 12 in Figure 5, checks if each element of a list h have tally bigger than the tally of a particular candidate c. Finally, clause 16 in the definition of `elim_cand_def`, is matched extensionally against the two functions `subpile1` and `subpile2`. For each of these equivalences, there are proved lemmas in our formalisation. By calling these lemmas, we prove the following theorem.

Theorem 3.4. *The predictive and computational formalised counterpart of the rule* elim *are equivalent, i.e.*

```
∀st qu l c j1 j2.elim_cand st qu l c j1 j2
     <=> (Elim_cand_dec st qu l c (j1,j2))
```

In the same way, we define predicates which are formalised twins of the other rules in the last section. Also we give computational functions that are shown to match exactly with their predicative counterpart. Hence, we verify that when a rule, as a computational function, applies, it always meets the expectations of the predicative part. These predicative and computational counting rules formalised, are put together to constitute the predicative and computational checker, respectively. Consequently, we ascertain that when the checker checks an input as valid, the returned result is not questionable.

4 Further Work

To complete the project and reach to executable machine code, we need to translate mechanically into CakeML executable code. For that, we need the rational arithmetic of HOL4 to be translatable from HOL4 into CakeML executable code. When the translation is available, we would need to give a specification for the input of the program and the output of the computation and prove it for correctness inside HOL4. Through mechanical extraction and by help of the above specification, we will obtain verified machine code.