Vote Counting as Mathematical Proof

Dirk Pattinson

May 14, 2015

Abstract

Trust in the correctness of an election outcome requires proof of the correctness of vote counting. By formulating particular voting protocols as rules, correctness amounts to demonstrating that all rules have been applied correctly. A proof of the outcome of any particular election then consists of a sequence (or tree) of rule applications and provides and independently checkable certificate of the validity of the result. This eliminates the need to trust, or otherwise verify, the correctness of the vote counting software once certificate has been validated. Using a rule-based formalisation of voting protocols inside a theorem prover, we synthesise vote coating programs that are not only provably correct, but also produce independently verifiable certificates. These programs are generated from a (formal) proof that every initial set of ballots allows to decide the election winner according to the given rules.

1 Introduction

In traditional, paper-based elections, vote-counting is observed by scrutineers. The role of these observers is to verify that the counting protocol is applied correctly, thus establishing trust in the election outcome.

For electronic vote counting, the situation is much worse. The vote counting software is usually written in a high-level programming language, and hinges not only on the correct translation of a voting protocol to the chosen programming language, but also on the correct implementation of the protocol and usually on the correctness of library functions used in the implementation of the software. This truly Herculanean task (e.g. [1]) hinges on

1. the formalisation of the voting protocol in formal logic inside a theorem prover
2. the formalisation of the semantics of the chosen programming language
3. a formal proof that the vote counting software indeed implements the voting protocol.

Our trust in correct counting of votes therefore lies with an extremely small number of highly-trained experts that have the necessary technical skills to affirm that the verification task has indeed been carried out diligently. This is in sharp contrast to paper-based elections where members of the general public can satisfy themselves that the counting protocol is being adhered to, by simply acting as election observers.

From a technical point of view, the difficulty with traditional vote counting software is the fact that it just delivers results, but no means to independently ascertain their correctness so that trust in outcomes requires us to analyse the whole chain of tools involved in their determination.

This difficulty has been recognise in [5] where the use of *linear logic* [6] is proposed for the specification and analysis of voting schemes. Unlike classical logic, linear logic is sensitive to the notion of resource in the form of assumptions that may only be used once, which in turn allows us to formulate e.g. that ballots are only counted once very concisely. An automated translation from the linear logic specification then directly translates this specification into executable code, and asserting the correctness of a count corresponds to (machine-) checking the (linear logic) proof generated by the linear logic program, a task that can be performed independently of the specification by external, light-weight proof checkers. In other words, the linear logic proof plays the role of an independently verifiable certificate that attests to the correctness of an individual election count.

This de-couples the task of certificate generation and verification, and trust in the outcome of a particular election can be established using the relatively simple task of checking a linear logic proof.

This paper goes one step further and advocates the view of vote counting as mathematical proof. Rather than translating voting protocols into a logical formalism such as linear logic, we interpret the specification of a voting protocol as a set of (vote counting) rules that have the same status as proof rules in mathematical logic.

The similarity between vote counting rules and rules used in (mathematical) proof theory is indeed striking. Consider for example the following rule of disjunction (or) introduction and counting a single vote in FPTP elections:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{b \vdash (u \mathbin{;} [c] \mathbin{;} u', t)}{b \vdash (u \mathbin{;} u', t[c \mapsto t(c) + 1)]}$$

In the logical rule on the left, Γ is a set of *assumptions* and the premiss reads as 'A is provable from the assumptions Γ'. The conclusion states that in this case, also the formula $A \vee B$ is provable (from the same assumptions). For vote counting (on the right), we read b as the collection of ballots cast (each ballot being a vote for a single candidate), $u \mathbin{;} [c] \mathbin{;} u'$ as a collection of uncounted votes comprised of u, u' and a single vote for candidate c, and $t : C \to \mathbb{N}$ as the tally, a function from the set C of candidates to the natural numbers, with $t[c \mapsto t(c) + 1]$ denoting the function that maps $c \mapsto t(c) + 1$ and $d \mapsto t(d)$ for $d \neq c$. In words: if we have reached a situation where the uncounted votes contain a (single) vote for candidate c and we have a running tally t, then removing c from this set of uncounted votes, together with the updated tally $t[c \mapsto t(c) + 1]$ is also a correct state of FPTP vote counting. Formal mathematical are based on axioms, i.e. statements for which no further justification is necessary. These axioms correspond to the initial states of vote counting. Again, for simple FPTP elections, the similarity is striking. Consider e.g. the rules

$$\frac{}{\Gamma \cup \{A\} \vdash A} \qquad \frac{}{b \vdash (b, \mathrm{nty})}$$

where $\mathrm{nty} : C \to \mathbb{N}$ is the null tally, i.e. the function $c \mapsto 0$ for all $c \in C$. The logical axiom on the left says that every formula A is a consequence of a set of assumptions containing A. The vote counting axiom on the right stipulates that taking all of B as uncounted, together with a tally that records 0 cast votes for all candidates, is a correct state of vote counting.

In mathematical logic, we accept a statement as provable if we can produce a proof, a tree or sequence of correct rule applications whose leafs are axioms. By analogy, a tree or sequence of correctly applied vote counting rules that determine the outcome of an election is in itself a proof of the correctness of the count. In the same way in which a formal mathematical proof is both machine checkable and independent from the means to generate it, a sequence of of vote counting rules can be checked independently from the software that has been used to generate it.

Compared to the encoding of voting protocols into existing logical formalisms, the use of protocol-specific (proof) rules minimises the gap between the natural language specification of a voting protocol and its formalisation while retaining machine-checkable certificates: a certificate is nothing more but a sequence (or tree) of rule applications that can be verified for correctness by simply checking that all rules have been applied correctly. The technical skills required to independently produce code that verifies the correctness of certificates do not extend beyond the skills acquired in a first programming course which dramatically increase the pool of (electronic) scrutineers.

This paper exemplifies this approach using two voting protocols, first-past-the-post (FPTP) elections, and single transferable vote (STV). To obtain a provably correct vote counting program, we formalise the rules that describe both formalisms in the theorem prover Coq [2] and define a type of proofs, where a proof is a sequence of correctly applied vote counting rules.

We then establish, again in Coq, that there exists both a candidate, or set of candidates, and a proof, in the above sense, of this fact. We then use Coq's program extraction facility to automatically construct a (provably correct) program that determines election winners, and constructs a formal proof of this fact using the vote counting rules. This proof is represented by a standard, inductive data type in a programming language (we choose Haskell) and so open to (electronic) scrutiny by means of proof checking programs.

Related Work. We have already mentioned [5] who take a similar approach and formalize election protocols using linear logic. While this enables us to use off-the-shelf proof checkers to verify the correctness of an election count, it requires a substantial degree of familiarity with the formalism to ascertain that linear logic specifications indeed reflect the protocol. We claim that the gap between the mathematical formalisation and the textual description is much smaller in our case. Trust in the correctness of election outcome in *op.cit.* rests in trust of in the correctness of proof checkers whereas in the approach presented here, proof checkers can be implemented in main-stream programming languages, and we claim that the skills required do not go beyond those acquired in in a first-year programming course.

The task of verifying vote counting software has been attacked, with various levels of success, for instance in [3] and [1]. Both approaches focus on certifying the process that is being used to compute election results, but no certificates are produced. Our trust in the correctness of election outcomes therefore relies on a whole tool-chain and only very few experts with in-depth technical knowledge can testify to soundness of methods and proper execution of procedures. In our approach no such trust is needed: we would accept the proof of the correctness of an election outcome as long as it passes (independent) verification, irrespective of the method used to obtain it. Software that is currently used to compute election results, such as the EVACS system used in the Australian Capital Territory [7], represent the voting protocol in a main-stream programming language. This results in a non-trivial gap between the textual and the formal specification of the voting protocol in use, and does not produce independently verifiable certificates. The correctness of the outcome not only relies on the correctness of the program *per se* but also on the vast amount of third-party libraries used in the implementation.

Our work addresses the auditable correctness of vote counting rather than the overall security of electronic voting protocols and is therefore independent of, but complementary to the verification of security properties of voting schemes, as e.g. carried out in [4].

Coq Proofs and Haskell Code. The coq proofs from which the vote counting code is generated, as well as the auxiliary Haskell functions for proof checking are downloadable from `http://users.cecs.anu.edu.au/~dpattinson/Software/`.

2 Preliminaries and Notation

This section introduces (mostly standard) notation for lists that we use throughout the paper. We write $\mathsf{List}(X)$ for the set of lists with elements in X, $[a_0, a_1, \ldots, a_n]$ for the list with elements $a_0, a_1, \ldots, a_n$, and denote a list with first element f and remainder fs by $f{:}fs$. The length of a list l is written as $\mathsf{len}(l)$ and the concatenation of two lists l and m by $l \mathbin{;} m$. Otherwise, we use set notation and say, e.g. $c \in l$ to express that c occurs in the list l, and use $\bigcup l$ for the set of elements of l.

3 First Past the Post Elections

We use the FPTP elections as a simple, introductory example. A textual specification of FPTP is, for example, the following.

1. Mark all ballots as being uncounted votes.
2. To count a single vote, pick an uncounted vote, mark it as counted, and increase the candidate's tally by one.
3. If no uncounted votes remain, the candidate with highest tally will be declared the winner.

This informal specification relies on contextual knowledge (e.g. that the initial tally of each candidate is 0 that needs to be made precise in the mathematical formalisation.

Mathematical Formalisation. For the mathematical formalisation of this protocol in terms of (logic look-alike) rules, we fix a set C of candidates and the two judgements

$$b \vdash \mathsf{winner}(c) \quad \text{and} \quad b \vdash \mathsf{state}(\mathsf{u}, \mathsf{t})$$

where $b \in \mathsf{List}(C)$ is the list of votes cast (we identify a vote for a candidate with the actual candidate). The judgement on the left asserts that $c \in C$ wins the election where the list b of ballots have been cast, and the judgement on the right represents a state of counting the ballots b where $u \in \mathsf{List}(C)$ are uncounted votes and $t : C \to \mathbb{N}$ is the running tally.

Remark 3.1 (Lists vs Multisets). We could have formalised ballots as multisets, rather than lists of votes to reflect the fact that the position of a vote in the lists of votes has no influence on the final outcome. We chose lists over multisets as a list-based formulation can be more directly formulated inside a theorem prover. The fact that the position of a vote does not matter is reflected in the formulation of rule ($\mathsf{C1}$): a single vote can be counted whenever we can split the list u of uncounted votes into a left-hand part u_0, the vote c to be counted and a right-hand part, that is, $u = u_0 \mathbin{;} [c] \mathbin{;} u_1$.

The rules themselves are direct transcriptions of the protocol above. The first (premiss-free) rule plays the role of an axiom in formal logical reasoning and bootstraps the process of vote counting. It takes the form

$$(\mathsf{Ax})\frac{}{b \vdash \mathsf{state}(b, \mathsf{nty})}$$

where $\mathsf{nty} : C \to \mathbb{N}$ is the null tally, i.e. $\mathsf{nty}(c) = 0$ for all $c \in C$, and asserts that the state where all ballots cast are uncounted, together with the null tally constitutes a correct (initial) state of counting. To count one vote, we need to increment the running tally of a candidate by one. We express that candidate c's tally has been incremented by one in the tally t' by the formula

$$\mathsf{inc}(c, t, t') \equiv t'(c) = t(c) + 1 \land \forall d \in C.(d \neq c \to t'(d) = t(d))$$

which additionally asserts that the tally of all other candidate remains unchanged. This gives the rule

$$(\mathsf{C1})\frac{b \vdash \mathsf{state}(u_0 \,;\, [c] \,;\, u_1, t)}{b \vdash \mathsf{state}(u_0 \,;\, u_1, t')} \quad (\mathsf{inc}(c, t, t'))$$

that formalises the counting of a single vote. This vote (for candidate c in the formulation above) appears at an arbitrary position in the list of uncounted votes in the premiss and is removed from the uncounted votes in the conclusion. The side condition (that must be met for the rule to be applicable) expresses that c's tally has been incremented and the tally of all other candidates remains unchanged. In words: if we have an uncounted vote for candidate c, we mark it as counted by removing it from the list of uncounted votes, and increase c's tally by one. Finally the rule

$$(\mathsf{Dw})\frac{b \vdash \mathsf{state}([], t)}{b \vdash \mathsf{winner}(c)} \quad (\forall d \in C.\, t(d) \leq t(c))$$

declares c to be the election winner if all other candidates have fewer votes, and no uncounted votes remain. This last rule precisifies our reading of the informal description: a candidate *can* be declared winner all others have as many votes or less. This is to say that in case of a draw, we have more than one election outcome.

Logical Formalisation. We now formalise the rules and the judgements in the Coq theorem prover. The right-hand part of judgements (election winners and state of counting) are a type that accommodates both possibilities

```
(* intermediate and final states of counting *)
Inductive Node  :=
  winner : cand -> Node                          (* electon winner *)
| state  : (list cand) * (cand -> nat) -> Node  (* election state *).
```

and proof trees that are built by successive applications of these rules as the inductive type given in Figure 1.

```
Inductive Pf (b: list cand) : Node -> Type :=
  ax : forall u t,                   (** start counting **)
  u = b ->                           (* if the uncounted votes comprise all ballots *)
  t = nty ->                         (* and the tally is nill *)
  Pf b  (state (u, t))               (* we may start counting with nil tally and all ballots *)
| c1 : forall u0 c u1 nu t nt,       (** count one vote **)
```

```
   Pf b  (state  (u0 ++[c]++u1, t)) -> (* if we have an uncounted vote for c *)
   inc c t nt ->                       (* and the new tally increments c's votes by one *)
   nu = u0++u1 ->                      (* and the vote has been removed from the uncounted votes *)
   Pf b (state (nu, nt))               (* we continue  with new tally and consume the vote for c *)
 | dw : forall c  t,                   (** declare winner **)
   Pf b  (state ([], t)) ->            (* if all votes have been counted *)
   (forall d : cand, (t d <= t c)) ->  (* and all cands have fewer votes than c *)
   Pf b  (winner c)                    (* then c may be declared the winner *).
```

Figure 1: The type of proofs for FPTP voting

In the above, we use the shorthands

```
(* null tally *)
Definition nty : cand -> nat := fun x => 0.

(* the new tally is the old tally with c's votes incremented by one *)
Definition inc (c:cand) (t: cand -> nat) (nt: cand -> nat) : Prop :=
  (nt c = t c + 1)%nat /\  forall d, d <> c -> nt d = t d.
```

to express the null tally and the side condition of rule (`C1`). This defines `Pf` as a (dependent) inductive type, parametrised by the list `b` of type `list cand` that represents all ballots cast. Inhabitants of this type can then be constructed using the constructors `ax`, `c1` and `dw` that directly represent the above rules. The side conditions of the rules appear as predicates that need to be satisfied for the constructor to be applicable which guarantees that they are indeed satisfied when constructing a proof. It is immediate to ascertain that the inductive type `Pf` defined in this way is an exact representation of the proof rules introduced earlier.

Certifiably Correct Counting. Based on this definition, we prove a simple sanity theorem (inside Coq): all elections have a winner. More specifically: for all sets b of ballots cast, there exists a candidate `w` that wins the election *provably*, that is, there is (also) a proof of this fact (using the counting rules), i.e. an inhabitant (element) of the type `Pf b (winner w)`.

```
Theorem exists_winner_pf : forall b, existsT w, Pf b (winner w).
```

In this theorem, `existsT` is a type-level existential quantifier (a Σ-type) that asserts the existence of a candidate, together with a proof of the fact that the candidate is the election winner. This theorem is proved for an arbitrary type `cand` under the assumption that `cand` is finite, inhabited, and has decidable equality. These assumptions are established automatically for any type that represents a finite set.

Crucially, we give a *constructive* proof of the theorem above. As consequence, the proof contains enough information to in fact *find* an election winner together with a proof of this fact. The use of the type-level existential quantifier allows us to either execute the proof itself in Coq to determine winners, or to use Coq's extraction mechanism [8] to generate a vote counting program that (a) delivers the election result together with a proof of this fact, and (b) does this in a provably correct way, i.e. the generated program *provably* generates a correct outcome/proof pair. From the proof of `exits_winner_pf` above, we automatically construct a program that both determines the winner, and produces a proof of this fact, using the vote counting rules. We choose to extract this program in the Haskell programming language [9], other possible choices are OCaml and Scheme. The extraction mechanism defines the type

`Pf` of proofs, and a function `exists_winner_pf` that – given a list of ballots – computes an election winner and a proof of this fact. The generated data type itself is polymorphic in the type `cand` of candidates:

```
data Pf cand =
   Ax (List cand) (cand -> Nat)
 | C1 (List cand) (cand -> Nat) (Pf cand)
 | Dw cand (Pf cand)
```

The Haskell data type above is (automatically) generated from the definition in Coq by removing the side conditions (that are not representable in a programming language). The function `exists_winner_pf` delivers both a candidate and a proof, wrapped into the (inductive) type `SigT`, provided our initial assumptions can be met: the set of candidates is finite and non-empty. These assumptions are routinely established for finite inductive types such as the following

```
Inductive cand := Alice | Bob | Claire | Darren.
```

so that we can specialise (polymorphic) counting to this type, and at the same time discharge the assumptions of finiteness and non-emptiness. Extracting this specialised function produces a Haskell-function of the type below.

```
cand_exists_winner_pf :: (List Cand) -> SigT Cand (Pf Cand)
```

Running this code, and visualising the representation of `Pf`, we obtain the following for a simple, four-person election, we obtain the proof displayed in Figure 2.

```
(ax)----------------------------------------------------------------------------
    st([Alice, Bob, Bob, Claire, Darren], Alice[0] Bob[0] Claire [0] Darren [0])
(c1)---------------------------------------------------------------------
    st([Bob, Bob, Claire, Darren], Alice[1] Bob[0] Claire [0] Darren [0])
(c1)----------------------------------------------------------------
    st([Bob, Claire, Darren], Alice[1] Bob[1] Claire [0] Darren [0])
(c1)-----------------------------------------------------------
    st([Claire, Darren], Alice[1] Bob[2] Claire [0] Darren [0])
(c1)---------------------------------------------------
    st([Darren], Alice[1] Bob[2] Claire [1] Darren [0])
(c1)---------------------------------------------
    st([], Alice[1] Bob[2] Claire [1] Darren [1])
(dw)-----------
    winner(Bob)
```

Figure 2: An example proof of the outcome of an FPTP election

We see the election winner (Bob, the first component of the Σ-type) in the first line of the output, the remainder of the output is a proof of this fact (the second component of the Σ-type). While the use of the extraction mechanism guarantees probable correctness of the generated program (we can guarantee that both election winners are computed correctly and correctness of the generated proof), the proof serves as a machine-checkable certificate that can be verified independently.

Proof Checking. It is routine to implement a proof-checker that confirms whether or not a proof of an election outcome, i.e. an element of the data type `Pf Cand`, represents a correctly formed proof tree whose last judgement declares the claimed winner. Under the proof-as-certificate interpretation, this is a certificate verifier. The certificate verifier is written in a general-purpose programming language (we choose Haskell as proofs are already Haskell data types) and is of the same length as the specification (about ten lines of Haskell code), a programming task solvable by any first-year undergraduate. We argue that the simplicity of certificate checkers entails a substantial gain in the trust in the election outcome once the certificate is checked by a (n ideally large) number of checkers, constructed by different individuals.

4 Single Transferable Vote

Several variations of STV voting are in use in various countries around the world (e.g. Malta, Ireland, India and Australia). Every member of the electorate does not vote for a single candidate, but instead ranks the candidates in order of her personal preference. We use a vanilla version of STV to demonstrate our approach. STV is parametrised by the number of available seats and a quota that determines the number of votes a candidate must achieve to be elected. The most commonly used quota is the Droop quota, given by

$$q = \frac{\sharp\mathsf{ballots}}{\sharp\mathsf{seats} + 1} + 1$$

but our development is independent of the particular quota used. STV counting proceeds as follows.

1. if candidate has enough first preference to meet the quota, (s)he is declared elected. Any surplus votes for this candidate are transferred.
2. if all first preference votes are counted, and the number of seats is (strictly) smaller than the number of candidates that are either (still) hopeful or elected, a candidate with the least number of first preference votes is eliminated, and her votes are transferred.
3. if a vote is transferred, it is assigned to the next candidate (in preference order) on the ballot.
4. vote counting finishes if either the number of elected candidates is equal to the number of available seats, or if the number of remaining hopeful candidates plus the number of elected candidates is less than or equal to the number of available seats.

Mathematical Formalisation. As for FPTP, we express the election protocol in terms of (proof) rules that we then formalise in the Coq theorem prover. As before, we have a set C of electable candidates, and represent ballots by (rank-ordered) lists of candidates so that a single ballot is of type $B = \mathsf{List}(C)$. The formalisation uses two judgements:

$$(b, q, s) \vdash \mathsf{state}(u, a, t, h, d) \quad \text{and} \quad (b, q, s) \vdash \mathsf{winners}(w).$$

Both judgements are parameterised by a triple (b, q, s) where $b \in \mathsf{List}(B)$ is the list of ballots cast, $q \in \mathbb{N}$ is the quota used and $s \in \mathbb{N}$ is the total number of seats available. The judgement on the left represents an intermediate state of vote counting, where $u \in \mathsf{List}(B)$ is the list of uncounted ballots, $a : C \to \mathsf{List}(B)$ is an assignment that records, for each candidate $c \in C$, the first preferences $a(c)$ that have been counted in favour of c (so that the first preference of each ballot $b \in a(c)$ is c). The remaining components are the tally $t : C \to \mathbb{N}$ that records, for each candidate, the number of first preferences already counted in favour of c, the list h of hopeful candidate (that are still in the running), and e is the list of elected candidates.

Remark 4.1. There are several obvious variations of the formalisation above. As for FPTP we could, for example, represent the uncounted votes by multisets of ballots (rather than lists), or the elected candidates by sets of candidates (rather than lists). We use a list-based representation as it can be represented in a theorem prover with minimal overhead. In the formalisation below, we reflect the use of lists by the fact that e.g. a vote being counted can be drawn from an arbitrary position in the list of uncounted votes. The representation of e.g. hopeful candidates as lists (rather than sets) of candidates is reflected in the fact that the initial list of hopefuls is required to consist of pairwise disjoint elements, and we can prove that the property of being pairwise distinct is preserved throughout the protocol.

We describe the protocol above by means of the following rules, where we state side conditions for the applicability of the rule as a bullet-point list to the right of the rule. The first rule describes the initial state of vote counting where nty is the null tally and nas is the null assignment, that is, $\mathsf{nty}(c) = 0$ and $\mathsf{nas}(c) = []$ for all $c \in C$.

$$(\mathsf{Ax})\frac{}{(b, q, s) \vdash \mathsf{state}(u, a, t, h, e)}$$

- $u = b$, $a = \mathsf{nas}$, $t = \mathsf{nty}$, $e = []$
- h pairwise distinct, $C = \bigcup h$

"When we begin counting, all ballots are uncounted, no votes are recorded as counted, the running tally is 0 for all candidates, all candidates are hopefuls and the list of hopefuls is pairwise distinct."

The rule $(\mathsf{C1})$, read *count one vote*, formalises the process of counting one fist preference. This requires to update both tally and vote assignment. We express that c's tally has been incremented by one in the tally t' by the formula $\mathsf{inc}(c, t, t')$ as in the previous section, and the fact that the vote assignment a' arises from the assignment a by adding vote v for candidate c by expressing that the list of votes cast for c can be split in two such that inserting v in the middle gives the updates (list) of votes. As a formula:

$$\mathsf{add}(c, v, a, a') \equiv \exists l_1, l_2.(a(c) = l_1 \mathbin{;} l_2 \wedge a'(c) = l_1 \mathbin{;} [v] \mathbin{;} l_2) \wedge \\ \forall d \in C.(d \neq c \to a'(d) = a(d))$$

where we additionally express that the vote assignment doesn't change for other candidates. To ensure that the vote that has just been counted is removed from the list of uncounted votes, we use a generic judgement on lists,

$$\mathsf{eqe}(c, l, l') \equiv \exists l1, l2. (l = l1 \mathbin{;} l2 \wedge l' = l1 \mathbin{;} [c] \mathbin{;} l2)$$

that expresses that l and l' are equal, except that l' additionally contains c (at an arbitrary position). The rule then takes the following form:

$$(\mathsf{C1})\frac{(b,q,s) \vdash \mathsf{state}(u,a,t,h,e)}{(b,q,s) \vdash \mathsf{state}(u',a',t',h,e)}$$

- $\mathsf{eqe}((f{:}fs),u',u))$, $f \in h, t(f) < q$,
- $\mathsf{add}(f, f{:}fs, a, a')$ $\mathsf{inc}(f,t,t')$

"If we have an uncounted vote with first preference f, f is a hopeful and the tally of f is still below the quota, we record this vote as counted for f and increase the tally for f by one."

The rule (El), for "elect", applies to a candidate whose number of first preference votes meet the quota. We express the fact that the candidate is moved from the hopefuls to the elected candidates by assuming appropriate splittings of both lists.

$$(\mathsf{El})\frac{(b,q,s) \vdash (u,a,t,h,e)}{(b,q,s) \vdash (u,a,t,h',e')}$$

- $c \in h, t(c) = q, \mathsf{len}(e) < s$
- $\mathsf{eqe}(c,h',h)$, $\mathsf{eqe}(c,e,e')$

"If a hopeful candidate has reached the quota and there are still seats available, we declare this candidate as elected by moving her from the list of hopefuls to the list of elected candidates."

For a candidate no longer in the running, votes are transferred. This applies when we have an uncounted vote, the first preference of which names a candidate that is not a hopeful. In this case, we delete the first preference by replacing the vote with an identical copy where this first preference is deleted. Replacing elements in lists is expressed by the formula

$$\mathsf{repl}(c,d,l,l') \equiv \exists l_1, l_2. \, (l = l_1 \,⨟\, [c] \,⨟\, l_2 \wedge l' = l_1 \,⨟\, [d] \,⨟\, l_2)$$

that we read as "l' is the list l with one occurrence of c replaced by d".

$$(\mathsf{Tv})\frac{(b,q,s) \vdash (u,a,t,h,e)}{(b,q,s) \vdash (u',a,t,h,e)}$$

- $f \notin h$
- $\mathsf{repl}((f{:}fs), fs, u, u')$

"If we have an uncounted vote with first preference f, and f is not a hopeful candidate, we delete this first preference from the ballot."

The next rule, (Ey) discards empty votes. Empty votes could either be included in the initial set of ballots, or occur by successively transferring votes.

$$(\mathsf{Ey})\frac{(b,q,s) \vdash (u,a,t,h,e)}{(b,q,s) \vdash (u',a,t,h,e)}$$

- $\mathsf{eqe}([], u', u)$

"Uncounted votes with no preferences are discarded."

The next rule models the elimination of a candidate with minimal number of first preferences. We remove this candidate from the list of hopefuls and re-count her votes. As the candidate is no longer hopeful, votes will automatically be transferred.

$$(\mathsf{Tl})\frac{(b,q,s) \vdash ([],a,t,h,e)}{(b,q,s) \vdash (u,t,h',e)}$$

- $\mathsf{len}(e) + \mathsf{len}(h) > s$, $c \in h$
- $\forall d \in h.(tc \leq td)$, $\mathsf{eqe}(c,h,h')$, $u = a(c)$

"If the number of available seats exceeds the sum of hopeful and elected candidates, no uncounted votes remain, and candidate c has a minimal number of votes, then c is removed from the list of hopefuls and all votes cast for c are transferred."

It remains to formalise the two conditions that conclude the election: either the number of elected candidates equals the number of seats available, or the number of seats is greater than or equal to the union of elected and hopeful candidates. We begin with the first second, (Hw) for "hopeful are winners".

$$(\mathsf{Hw})\frac{(b,q,s) \vdash \mathsf{state}(u,a,t,h,e)}{(b,q,s) \vdash \mathsf{winners}(w)}$$

- $\mathsf{len}(e) + \mathsf{len}(h) \leq s$
- $w = e \mathbin{;} h$

"If the number of candidates that are either hopeful or elected is less than or equal to the number of seats available, then scrutiny ceases and all candidates that are either elected or hopeful are declared winners of the election".

The rule (Ew), for "elected are winners", is entirely analogous but applies if the number of seats equals the number of candidates marked as elected.

$$(\mathsf{Ew})\frac{(b,q,s) \vdash \mathsf{state}(u,a,t,h,e)}{(b,q,s) \vdash \mathsf{winners}(w)}$$

- $\mathsf{len}(e) = s$
- $w = e$

"If the number of elected candidates equals the number of seats available, scrutiny ceases and the elected candidates are declared the winners of the election".

We now express the rules above in the internal logic of Coq.

Logical Formalisation. As before, we formalise the rules and the associated notion of proofs, in a (parameterised, dependent) inductive type. We assume a finite type `cand` with decidable equality. In contrast to FPTP, we do not need to assume that this type is non-empty (for if it is, the empty set of candidates are valid election winners) but need to assume that the type of candidates is given by an enumeration with pairwise distinct elements. This is required to be able to always maintain a pairwise distinct list of hopefuls.

The auxiliary statements that are used to express the side conditions of the rules are direct translations from informal logic into Coq.

```
(* the new tally is the old tally with c's votes incremented by one *)
Definition inc (c:cand) (t: cand -> nat) (nt: cand -> nat) : Prop :=
  (nt c = t c + 1)%nat /\  forall d, d <> c -> nt d = t d.

(* the new assignment is the old assignment with ballot b added for candidate c *)
Definition add (c:cand) (b:ballot) (a: cand -> list ballot) (na: cand -> list ballot) : Prop :=
  na c = b::(a c) /\  forall d, d <> c -> na d = a d.

(* l and nl are equal except that nl additionally contains x *)
Definition eqe {A: Type} (x:A) (l: list A) (nl: list A) : Prop :=
  exists l1 l2: list A, l = l1 ++ l2 /\ nl = l1 ++ [x] ++ l2.

(* l contains x and replacing x with y in l yields nl *)
Definition rep {A:Type} (x y: A) (l nl: list A) :=
  exists l1 l2: list A, l = l1 ++ [x] ++ l2 /\ nl = l1 ++ [y] ++ l2.
```

Intermediate and final states of counting are similar to FPTP, with the exception that more book-keeping is needed.

```
(* intermediate and final states in vote counting *)
Inductive Node :=
   state:                    (** intermediate states **)
       list ballot           (* uncounted votes *)
     * (cand -> list ballot)  (* assignment of counted votes to * candidates *)
     * (cand -> nat)          (* tally *)
     * (list cand)            (* hopeful cands still in the running *)
     * (list cand)            (* elected cands no longer in the running *)
     -> Node
  | winners:                  (** final state **)
    list cand -> Node.        (* election winners *)
```

Finally, proof trees are captured by the following type (family) that is again a direct translation of the mathematical formalisation and is given in Figure 3.

```
Inductive Pf (b: list ballot) (q: nat) (s: nat) : Node -> Type :=
  ax :  forall  u a t h e,             (* start counting *)
  (forall c: cand, In c h) ->          (* if all candidates are hopeful and *)
  PD h ->                              (* hopefuls are pairwise distinct *)
  u = b ->                             (* if the list of ballots contaeqe all ballots *)
  a = nas ->                           (* and the initial assignment is the null assignment *)
  t = nty ->                           (* and we begin with the null tally *)
  e = nbdy ->                          (* and nobody is elected initially *)
  Pf b q s (state (u, a, t, h, e))     (* we start counting with this data *)
| c1 : forall u nu a na t nt h e f fs, (** count one vote **)
  Pf b q s (state (u, a, t, h, e )) -> (* if we are counting votes, *)
  eqe (f::fs) nu u ->                  (* have a uncounted vote with 1st pref f removed *)
  In f h ->                            (* and f is a hopeful *)
  t f < q ->                           (* and this isn't surplus *)
  add f (f::fs) a na ->                (* and the new assignment records the vote for f *)
  inc f t nt ->                        (* and the new tally increments the votes for f by one *)
  Pf b q s (state (nu, na, nt, h, e))  (* we continue with updated tally and assignment *)
| el : forall u a t h nh e ne c,       (** elect a candidate **)
  Pf b q s (state (u, a, t, h, e)) ->  (* if we have an uncounted vote with 1st preference f *)
  In c h ->                            (* and c is a hopeful *)
  t(c) = q ->                          (* and c has enough votes *)
  length e < s ->                      (* and there are still unfilled seats *)
  eqe c nh h ->                        (* and f has been removed from the new list of hopefuls *)
  eqe c e ne ->                        (* and added to the new list of electeds *)
  Pf b q s (state (u, a, t, nh, ne))   (* then proceed with updated  hopefuls and elected cands *)
| tv : forall u nu a t h e f fs,       (** transfer vote **)
  Pf b q s (state (u, a, t, h, e)) ->  (* if we are counting votes *)
  ~(In f h) ->                         (* and f no longer in the running *)
  rep (f::fs) fs u nu ->               (* and f is being removed from an uncounted ballot *)
  Pf b q s (state (nu, a, t, h, e))    (* we continue with updated set of uncounted votes *)
| ey : forall u nu  a t h e,           (** empty vote **)
  Pf b q s (state (u, a, t, h, e)) ->  (* if we are counting votes *)
  eqe [] nu u ->                       (* and an empty vote is removed from uncounted votes *)
  Pf b q s (state (nu, a, t, h, e))    (* continue with updated set of uncounted votes *)
| tl : forall u a t h nh e c,          (** transfer least **)
  Pf b q s (state ([], a, t, h, e)) -> (* if we have no uncounted votes *)
  length e + length h > s ->           (* and there are still too many candidates *)
  In c h  ->                           (* and candidate c is still hopeful *)
  (forall d, In d h-> t c <= t d) ->   (* but all others have more votes *)
  eqe c nh h ->                        (* and c has been removed from the new list of hopefuls *)
  u = a(c) ->                          (* and marked to be transfered *)
  Pf b q s (state (u, a, t,nh, e))     (* transfer c's votes and proceed with new hopefuls *)
| hw : forall w u a t h e,             (** hopefuls win **)
  Pf b q s (state (u, a, t, h, e)) ->  (* if at any time *)
  length e + length h <= s ->         (* we have more hopefuls and electeds  than seats *)
  w = e ++ h ->                       (* and the winning candidates are their union *)
  Pf b q s (winners (w))              (* then they are declared winners *)
| ew : forall w u a t h e,            (** elected win **)
  Pf b q s (state (u, a, t, h, e)) -> (* if at any time *)
  length e = s ->                     (* we have as many elected candidates as setas *)
  w = e ->                            (* and the winners are precisely the electeds *)
  Pf b q s (winners w).               (* they are declared the winners *)
```

Figure 3: The inductive type for STV proofs

We invite the reader to ascertain the 1-1 correspondence between the mathematical and logical formalisation.

Certifiably Correct Counting. As for FPTP, we do not implement a program that produces both election winners and proofs of this fact from any given set of ballots, but instead prove constructively that election winners always exist. In Coq:

```
Theorem ex_winners_pf: forall b q s, q > 0 ->
  existsT w: list cand, Pf b q s (winners w).
```

The additional assumption of a non-negative quota is technical: if the quota would be 0, all candidates that are initially hopeful could be elected straight away. The details of turning this proof into (executable) Haskell code are as for FPTP. For an example election with four candidates competing for two seats, we obtain for instance the proof displayed in Figure 4.

```
(ax)-----------------------------------------------------------------------------------------
    ([[Alice,Bob], ...], Alice[0] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(c1)---------------------------------------------------------------------------------------------
    ([[Alice,Charlie], ...], Alice[1] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(c1)----------------------------------------------------------------------------------------------
    ([[Deliah,Charlie], ...], Alice[2] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(el)---------------------------------------------------------------------------------------------
    ([[Deliah,Charlie], ...], Alice[2] Bob[0] Charlie[0] Deliah[0], [Bob,Charlie,Deliah], [Alice])
(c1)----------------------------------------------------------------------------------------
    ([[Bob,Alice], ...], Alice[2] Bob[0] Charlie[0] Deliah[1], [Bob,Charlie,Deliah], [Alice])
(c1)--------------------------------------------------------------------------------------
    ([[Charlie], ...], Alice[2] Bob[1] Charlie[0] Deliah[1], [Bob,Charlie,Deliah], [Alice])
(c1)-----------------------------------------------------------------------
    (*, Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie,Deliah], [Alice])
(tl)--------------------------------------------------------------------------------------
    ([[Deliah,Charlie], ...], Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie], [Alice])
(tv)-------------------------------------------------------------------------------
    ([[Charlie], ...], Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie], [Alice])
(c1)----------------------------------------------------------------
    (*, Alice[2] Bob[1] Charlie[2] Deliah[1], [Bob,Charlie], [Alice])
(el)----------------------------------------------------------------
    (*, Alice[2] Bob[1] Charlie[2] Deliah[1], [Bob], [Charlie,Alice])
(ew)-------------------------
    winners ([Charlie,Alice])
```

Figure 4: An example STV proof

For presentation purposes, we only show the first uncounted vote, the remaining uncounted votes are indicated with ellipses, and we elide the assignment of candidates to votes (counted in their favour).

Proof Checking. In contrast to FPTP, proof checking involves the verification of eight (rather than three) proof rules. Each rule can be verified using at most six lines of Haskell code, leading to a proof checker that is of about the same size as the specification. As proof rules are independent, this only adds quantity (not complexity) to the task of implementing a proof checker. As for FPTP, we claim that the proof checker can be implemented by a first-year undergraduate student.

5 Conclusions and Outlook

We have presented an approach to certifiably correct vote counting where the voting protocol is formalised as a set of (vote counting) rules. Our conceptual claim is that the mathematical formalisation of voting protocols as rules is very close to the procedural understanding of the counting process so that the gap between the legal text and the mathematical formulation is minimal.

We argue furthermore that vote counting software should not only produce an election result but additionally a proof of this fact, and we have demonstrated that the rule based formulation is ideally suited for this task. Proofs of the election outcome in this sense are certificate that can be independently verified by third parties and so greatly contribute to the overall amount of trust that we are prepared to place in electronic counting. The fact that proof checkers, or certificate verifiers, are comparatively simple to implement allows anyone with basic programming skills to independently verify the election result. In our experience, the size of (the essential parts of) a proof checker is almost identical with the size of the formal specification, and scales linearly with the number of proof rules.

We have demonstrated our approach by formalising election protocols in Coq. While the formalisation in Coq in itself guarantees that the generated vote counting program is in itself provably correct, it additionally produces a certificate (proof) of election counts that is open to independent scrutiny.

The case study presented in this paper can only be the beginning of a larger effort. We have formalised vote counting non-deterministically so that ties are never being broken, but all outcomes lead to valid proofs. This is not (yet) reflected in the (generated) software where ties are broken arbitrarily. This can be alleviated by either generating all possible outcomes of vote counting (where ties are broken in all possible ways). One could also introduce ties as a judgement and ways to break ties into the input.

Avenues that deserve further exploration are meta-properties of voting protocols such as modularity or uniqueness of winners, and the application to real-world voting protocols.

References

[1] Bernhard Beckert, Rajeev Goré, Carsten Schürmann, Thorsten Bormer, and Jian Wang. Verifying voting schemes. *J. Inf. Sec. Appl.*, 19(2):115–129, 2014.

[2] Yves Bertot, Pierre Castran, Grard Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.

[3] Dermot Cochran and Joseph Kiniry. Votail: A formally specified and verified ballot counting system for irish PR-STV elections. In *Pre-proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, 2010.

[4] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols: A taster. In David Chaum, Markus Jakobsson, Ronald L. Rivest, Peter Y. A. Ryan, Josh Benaloh, Miroslaw Kutylowski, and Ben Adida, editors, *Towards Trustworthy Elections, New Directions in Electronic Voting*, volume 6000 of *Lecture Notes in Computer Science*, pages 289–309. Springer, 2010.

[5] Henry DeYoung and Carsten Schürmann. Linear logical voting protocols. In Aggelos Kiayias and Helger Lipmaa, editors, *Proc. VoteID 2011*, volume 7187 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2012.

[6] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[7] Software Improvements. Electronic and voting and counting sytems. http://www.softimp.com.au/evacs/index.html, 2015. accessed at May 12, 2015.

[8] Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Proc. TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2003.

[9] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. Lulu, 2012.