Vote Counting as Mathematical Proof

Dirk Pattinson[1] and Carsten Schürmann[2](✉)

[1] The Australian National University, Canberra, Australia
[2] IT University of Copenhagen, Copenhagen, Denmark
carsten@itu.dk

Abstract. Trust in the correctness of an election outcome requires proof of the correctness of vote counting. By formalising particular voting protocols as rules, correctness of vote counting amounts to verifying 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 an independently checkable certificate of the validity of the result. This reduces the need to trust, or otherwise verify, the correctness of the vote counting software once the certificate has been validated. Using a rule-based formalisation of voting protocols inside a theorem prover, we synthesise vote counting 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 a set of given rules.

1 Introduction

In many countries scrutineers observe traditional, paper-based elections. Their role is to verify that the voting scheme is applied correctly, thus establishing trust in the election outcome.

For electronic vote counting, the situation is much more difficult. The components of the vote counting software are usually written in a high-level programming language such as C++, PL/SQL, or Java, and hinges not only on the correct translation of a voting protocol into a 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 herculean task (e.g. [2]) hinges on

1. the formalisation of the voting protocol in formal logic in a theorem prover
2. the formalisation of the semantics of the chosen programming language
3. a formal correctness proof.

Thus, the trust in the correct counting of votes lies with an extremely small number of highly-trained experts that have the necessary technical skills to affirm that the verification task has been carried out with due diligence. This is in sharp contrast to paper-based elections where trust is established by the presence of election observers.

B. Pfahringer and J. Renz (Eds.): AI 2015, LNAI 9457, pp. 464–475, 2015.
DOI: 10.1007/978-3-319-26350-2_41

When using technology in elections, it becomes much more difficult to observe an election. Computer programs and systems are so complex that additional trust assumptions are required that are difficult to discard: is the software version correct? Was the software properly audited and tested? Was the computer that runs the program compromised? The difficulty with such software systems is the fact that they just delivers results, but no means to independently ascertain their correctness, so that trust in election results requires a careful analysis of the whole computation stack and the tools involved in their determination, in sharp contrast to verifiable voting [4] where no such trust is needed.

This difficulty has been recognised in [7], where *linear logic* [8] is used to specify and analyse for voting schemes and also as a certificate language that allows to validate election results retroactively. Unlike classical logic, linear logic is sensitive to the notion of resource in the form of assumptions that may only be used once. E.g. linear implication $a \Rightarrow b$ expresses that (the resource, e.g. a ballot) is *consumed* to yield b (e.g. an updated tally). This allows us to formulate vote counting algorithms abstractly and concisely, because votes are guaranteed to be counted only once. A proof search algorithm uses the program as input, and while constructing a proof for correct counting, it determines the election result in passing. Checking the correctness of a count corresponds to (machine-) checking the (linear logic) proof generated by the proof search algorithm. This task can be performed by a small proof checker for linear logic (anyone can implement such a checker) that is independent of the particular counting algorithm. 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.

This paper goes one step further and analyses the process of vote counting from the perspective of mathematical proof. Rather than translating voting protocols into a linear logical formalism, we interpret the voting protocol as a set of (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:

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

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 $A \vee B$ (A or 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 the list of currently uncounted votes, composed of two lists u and u' and a singleton list $[c]$, 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 proofs 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 with axioms at the leaves. By analogy, a tree or sequence of correctly applied vote counting rules that determine the outcome of an election is 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 produce it, a sequence of vote counting rules can be checked independently from the software by which it was generated.

Compared to the encoding of voting protocols in 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. This has two advantages. First, such verifiers can be implemented in low level languages with only a few lines of code reducing the need to trust compilers and libraries. Second, 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 increases the pool of (electronic) scrutineers. While the complexity of both correctness checking and counting itself is both polynomial, certificates can be used to precisely pinpoint *where* a count went wrong, and the rule-based formulation of protocols themselves serves as mathematical specification.

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 [3] and define a type of proofs, where a proof is a sequence of correctly applied vote counting rules. We use FPTP and STV as examples to enable comparison with [7]. In contrast to *op.cit.*, the higher expressive power of a general-purpose theorem prover such as Coq (in particular the availability of general arithmetic) allows us to extend our approach to more expressive schemes such as range voting.

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

Related Work. We have already mentioned the linear logic formalisation of [7]. 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 linear logic to ascertain that linear logic specifications indeed reflect the protocol. Trust in the correctness of election outcomes in [7] rests in the trust in the correctness of proof checkers. In our approach, proof checkers can be implemented in main stream programming languages, and thus many more than just a few experts can convince themselves about the correctness of the election result.

The idea of not only computing a result, but also provide an independently checkable trace of all intermediate steps was also advocated in [13] and is orthogonal to Necula's proof carrying code [12] where a program executable is accompanied with guarantees. We also advocate programs that not only provide a result, but also an independently verifiable certificate of its correctness.

The relationship between the representation of voting protocols in linear logic [7] and our approach is as follows. In linear logic, linear contexts are multisets of facts; rules when read in forward-chaining way correspond to multiset rewriting. The Coq formalisation in this paper makes this reasoning explicit: rules are state transformers, and they are implemented as such directly in Coq.

The task of verifying vote counting software has been tackled, with various levels of success, for instance in [2,5]. Both approaches focus on verifying the process 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 whose integrity can only be verified by very few experts with in-depth technical knowledge as already outlined above. Here, 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 [9], 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 [6].

Coq Proofs and Haskell Code. The Coq source code from which the vote counting code is generated, proof checkers, and examples are available from http://users.cecs.anu.edu.au/~dpattinson/Software/.

Notation. We write $\mathsf{List}(X)$ for the set of lists where elements are drawn from the set 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, i.e. sometimes identify a list with the set of its elements.

2 First Past the Post Vote Counting

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 the 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) which needs to be made precise in the mathematical formalisation.

Mathematical Formalisation. For the mathematical formalisation of this protocol in terms of (proof) 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. The rules themselves, given in Fig. 1, 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. Here nty is the null tally $\mathsf{nty}(c) = 0$. The rule ($\mathsf{C1}$) formalises the counting of a single ballot, where

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

expresses that t' is the tally resulting from t by incrementing c's votes by one, and (Dw) declares a candidate with a maximal number of votes as the winner. We ignore ties, i.e. any candidate with a maximal number of votes can be declared winner, but we could break ties by adding one more rule.

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

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

Fig. 1. Proof rules for FPTP counting

Logical Formalisation. To generate a provably correct counting program, we have formalised the rules and judgements in the Coq theorem prover. The formalisation is a simple textual translation from the mathematical representation. For example, the rule (C1) becomes

```
| 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 *)
```

and the auxiliary predicate used in the side condition is formalised below.

```
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.
```

This defines a (dependent inductive) type that represents derivations using the rules described above. The formalisation in Coq guarantees that the rules can only be applied if the side conditions are met.

Certifiably Correct Counting. Based on this definition, we prove a simple sanity theorem (in Coq): all elections have a winner. That is, for all sets b of ballots cast (assuming all ballots are formal), there is a candidate `w` that wins the election *provably*, and a proof of this fact, a derivation using the rules where all side conditions have been obeyed. This theorem is proved for an arbitrary type `cand` under the assumption that `cand` is finite, inhabited, and has decidable equality. These assumptions hold 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. We then use Coq's extraction mechanism [10] 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. This program not only determines the winner, but also a derivation of this fact, using the rules above. This derivation can then be independently checked for correctness. We have used the programming language Haskell [11], other possible choices are OCaml and Scheme. This generates a data type for derivations

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

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

Fig. 2. An example proof of the outcome of an FPTP election

and visualising the above data type, we obtain the run displayed in Fig. 2 where ... indicates the omission of (`C1`)-steps, and we have elided the $b \vdash$ prefix.

While the use of the extraction mechanism guarantees provable correctness of the generated program (we can guarantee both that 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 that requires modest skill. 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. One such checker is contained in the source code that accompanies this paper.

3 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 continuing (have not been eliminated) 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 continuing candidates plus the number of elected candidates is less than or equal to the number of available seats.

In this (simple) formulation of STV, we ignore transfer values and ties.

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 votes $a(c)$ with first preference c that have been counted in c's favour (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 continuing (hopeful) candidates that are still in the running, and e is the list of elected candidates. The judgement on the right above asserts that w is the list of election winners. We describe the protocol above by the rules given in Fig. 3. Our formalisation deliberately does not enforce any particular order of rule applications (this could be encoded using side conditions) so that our results pertain to *any* count that is correct according to the given rules.

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 $c \in C$. The rule ($\mathsf{C1}$) describes the counting of one first preference, where inc is defined as in Sect. 2, and we use

$$\mathsf{eqe}(c, l, l') \equiv \exists l_1, l_2. (l = l_1 \mathbin{;} l_2 \wedge l' = l_1 \mathbin{;} [c] \mathbin{;} l_2)$$

to expresses that l and l' are $\underline{\text{eq}}$ual, $\underline{\text{e}}$xcept that l' additionally contains c (at an arbitrary position), equivalently l' contains c and l arises by removing one

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

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

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

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

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

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

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

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

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

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

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

- $\mathsf{len}(e)+\mathsf{len}(h)>s$, $c\in h$, $u=a(c)$
- $\forall d\in h.(t(c)\leq t(d))$, $\mathsf{eqe}(c,h,h')$

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

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

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

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

Fig. 3. Proof rules for STV counting

occurrence of c from l. Adding a vote v as counted in favour of c to assignment a is represented by

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

where a' is the assignment resulting from this operation. The rule (El) applies once a candidate has reached the quota, (Tv) formalises the transfer of votes (where the first preference has either been eliminated or is already elected). Here, we use

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

to say that l' is the list l with one occurrence of c replaced by d. The rule (Ey) discards empty votes (with no remaining preferences), (Tl) eliminates a candidate with a minimal number of first preferences in a situation where all first preferences are counted and there are still seats to fill, by removing a candidate with least number of first preferences from the list of continuing candidates, and marking all ballots cast in her favour as uncounted. When counting these votes, preferences will be transferred using (Tv). The termination conditions are (Hw) that asserts that once the number of hopefuls and elected candidates falls below the number of available seats, both hopefuls and elected candidates shall be declared winners, and (Ew) that asserts that the elected candidates win, once their number is equal to the number of available seats. 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 which (again) is a matter of changing the syntax, as for FPTP counting. E.g. the rule (Tl) becomes:

```
| 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) ->                          (* we transfer c's votes by marking them as uncounted *)
  Pf b q s (state (u, a, t,nh, e))     (* and continue in this new state *)
```

Certifiably Correct Counting. As for FPTP, we do not implement a program, but instead extract a provably correct program from a constructive proof that winners always exist. From this, we generate a program that produces both election winners and a sequence of rule applications that justify this. The output of running this program on a sample election is depicted in Fig. 4 where some intermediate steps, as well as the initial $(b, q, s) \vdash$, have been elided.

For presentation purposes, we only show the first uncounted vote, the remaining uncounted votes are indicated with ellipses, and we elide the assignment of

```
(ax)---------------------------------------------------------------------------------
    state([[Ana,Bob], ...], Ana[0] Bob[0] Chris[0] Deb[0], [Ana,Bob,Chris,Deb], [])
(c1)-----------------------------------------------------------------------------------
    state([[Ana,Chris], ...], Ana[1] Bob[0] Chris[0] Deb[0], [Ana,Bob,Chris,Deb], [])
                                . . .
    state (*, Ana[2] Bob[1] Chris[1] Deb[1], [Bob,Chris,Deb], [Ana])
(tl)------------------------------------------------------------------
    state([[Deb,Chris], ...], Ana[2] Bob[1] Chris[1] Deb[1], [Bob,Chris], [Ana])
(tv)-------------------------------------------------------------------------------
    state([[Chris], ...], Ana[2] Bob[1] Chris[1] Deb[1], [Bob,Chris], [Ana])
(c1)---------------------------------------------------------------------------
    state(*, Ana[2] Bob[1] Chris[2] Deb[1], [Bob,Chris], [Ana])
(el)-------------------------------------------------------------
    state(*, Ana[2] Bob[1] Chris[2] Deb[1], [Bob], [Chris,Ana])
(ew)-------------------------------------------------------------
    winners ([Chris,Ana])
```

Fig. 4. An example STV proof

candidates to votes (counted in their favour). More examples are contained in the source code that accompanies this paper.

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 with basic programming skills.

4 Discussion

We have presented an approach for certifiably correct vote counting by formalising election protocols as rules that resemble logical deduction rules in spirit. Our implementations of vote counting are provably correct (as they are obtained from program extraction in a theorem prover) and deliver a machine-checkable certificate of the correctness of the count that can be validated with basic programming skills. For any system to be used in real elections, it needs to be demonstrated that the computed election outcome is in fact consistent with the vote counting method formalised in legislation. This amounts to showing that the execution of the computer code to generate the election result in fact conforms to the textual specification, enshrined in law. Various methods provide different types of evidence.

Open-Source Software such as e.g. open-source EVACS System [9] used e.g. in the Australian Capital Territory, can be scrutinised by any member of the general public. This means that any member of the public can scrutinize the computer *code* that has generated the election outcome, but no evidence of its correct *execution* is available. Moreover, even to ascertain the correctness of computer code is an extremely laborious and skill-intensive task.

Verified Vote Counting such as described e.g. in [2] provides a mathematical proof of program correctness with respect to its mathematical specification. To attest soundness of the overall system, we need to be sure that (a) the logical specification is correct with respect to the legislation, and (b) that the verification is carried out with due diligence, both of which require considerable skill. Even after both are carried out, no evidence of the correct *execution* is available.

Certifiably Correct Vote Counting. The approach presented in [7] goes one step further and eliminates trust in the verification process, as every count comes with a certificate (the linear logic proof) that can be checked independently. Moreover, this can be done with a proof checker that is verifiably correct down to the level of machine instructions [1]. On the other hand, the rather special formalism used (linear logic) requires substantial training and limits the pool of possible scrutineers.

Domain-Specific Certifiable Vote Counting. In the approach presented in this paper, only basic familiarity with mathematical concepts (and no knowledge of logic) is required to ascertain the correctness of the specification of the protocol. The certificates generated by our method can be verified by relatively simple computer programs that can be written and verified by anyone with basic programming skills. Running certificate checkers written in different programming languages running on different hardware back up the computed election result.

Conclusion. Clearly more research into the nature of trust in electronic elections is needed. The approach outlined here generates (independently verifiable) evidence for the correctness of election outcomes. Our formalisation only requires little mathematical knowledge and is very close to a textual specification of the voting protocol. As a consequence, correctness of the specification can be asserted by a significantly larger group of people compared to other approaches (that heavily rely on logical notions). The same applies to the certificates generated: their grounding in mathematically simple structures makes it possible to independently implement certificate checkers with moderate programming skills.

References

1. Appel, A.W., Michael, N.G., Stump, A., Virga, R.: A trustworthy proof checker. J. Autom. Reasoning **31**(3–4), 231–260 (2003)
2. Beckert, B., Goré, R., Schürmann, C., Bormer, T., Wang, J.: Verifying voting schemes. J. Inf. Sec. Appl. **19**(2), 115–129 (2014)
3. Bertot, Y., Castran, P., Huet, G., Paulin-Mohring, C.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
4. Chaum, D.: Secret-ballot receipts: true voter-verifiable elections. IEEE Secur. Priv. **2**(1), 38–47 (2004)
5. Cochran, D., Kiniry, J.: 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)
6. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols: a taster. In: Chaum, D., Jakobsson, M., Rivest, R.L., Ryan, P.Y.A., Benaloh, J., Kutylowski, M., Adida, B. (eds.) Towards Trustworthy Elections. LNCS, vol. 6000, pp. 289–309. Springer, Heidelberg (2010)
7. DeYoung, H., Schürmann, C.: Linear logical voting protocols. In: Kiayias, A., Lipmaa, H. (eds.) VoteID 2011. LNCS, vol. 7187, pp. 53–70. Springer, Heidelberg (2012)
8. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
9. Software Improvements. Electronic and voting and counting sytems. http://www.softimp.com.au/evacs/index.html (2015). Accessed 12 May 2015
10. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
11. Marlow, S., Peyton Jones, S.: The glasgow haskell compiler. In: The Architecture of Open Source Applications, vol. 2. Lulu (2012)
12. Necula, G.C.: Proof-carrying code. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Proceedings of the POPL 1997, pp. 106–119. ACM Press (1997)
13. Schürmann, C.: Electronic elections: trust through engineering. In: Proceedings of the RE-VOTE 2009, pp. 38–46. IEEE Computer Society (2009)