Modular Synthesis of Provably Correct Vote Counting Programs

Florrie Verity and Dirk Pattinson

The Australian National University

Abstract. Vote counting schemes, in particular those that employ various different variants of single transferable vote, often vary in small details. How are transfer values computed? What are the precise rules when two or more candidates tie for exclusion? In what order are candidates elected? These details are crucial for the correctness of vote counting software. While the verification of counting programs using interactive theorem provers gives very high correctness guarantees, correctness proofs need to be re-done when these details change, as this is reflected in both specification and implementation. This paper presents a framework where counting schemes are specified by rules, and provably correct, universally verifiable counting programs can be synthesised automatically, given (formal) proofs of two simple and intuitive properties of the specification of the protocol in rule form.

1 Introduction

Our trust in the correctness of paper-based vote counting rests on two basic pillars: first, vote counting is transparent, i.e. it does not happen behind closed doors. Second, we have a good understanding of the way in which votes are to be counted, that is, there is no confusion as to how the counting process should proceed.

These two properties can be replicated for electronic vote counting by (a) publishing an exact formal specification on how the vote counting program operates, and by (b) publishing a transcript of the count that can be independently verified by third parties.

These two properties go hand-in-hand as the form of the transcript invariably depends on the formal specification of the vote counting scheme. A close correspondence between hand-counting and machine-counting can be achieved by mirroring the actions of hand-counting in the formal specification. The electronic count then proceeds by applying precisely these actions, and the sequence of actions applied provides a transcript of the count that can be (electronically) scrutinised by third parties. In particular, this guarantees *universal verifiability* of the count, as the transcript plays the role of a certificate that can be independently verified by third parties.

For example, the action 'take an uncounted ballot, update the running tally according to the first listed preference, and place the ballot paper onto the pile

of votes counted for for the first preference candidate' becomes a rule that progresses the state of the count. To model this by means of a formal specification, we need to keep track of the data being manipulated. In this example, we need to record uncounted ballots, a running tally, and a pile of ballot papers for each candidate (on which votes counted in favour of this candidate are placed). A formalisation direct formalisation would therefore rely on a notion of *state* of vote counting that represents precisely this data (uncounted ballots, running tally and, for each candidate, a list of votes counted in their favour) that is manipulated by *rules*, i.e. a formal description of how states may be correctly manipulated according to the given vote counting scheme. This approach has been taken in [9] where two simple vote counting schemes are formalised in this way in the Coq theorem prover [3]. In particular, this style of formalisation is amenable to synthesising vote counting programs that are (a) provably correct, that is, each individual step corresponds to correctly applying a rule given in the specification, and (b) also deliver the sequence of rules applied, i.e. a universally verifiable transcript of the count. The synthesis of vote counting programs is not fully automatic, but requires a formal proof of the fact that for every given set of (initial) ballots, there is a sequence of correct rule applications that leads to the determination of election winners.

'vote counting as mathematical proof' [9] In the context of vote counting, the two fundamental principles that engender our trust in paper based elections are the transparency of the process (it doesn't happen behind closed doors) and a good understanding of the overall vote counting scheme (there's no confusion as to how the counting should proceed). The approach of 'vote counting as mathematical proof' [9] tries to bring these benefits to the area of electronic ballot counting. In a nutshell, actions of vote counting officials that progress the count are understood as rules, and rules are applied until the outcome of the count has been determined. Transparency is achieved by the counting algorithms not only providing a determination of the winner, but also the complete sequence of rule application that leads to its determination.

The work reported is based on two case studies using a simplified version of single transferable vote (STV) and first-past-the-post (FPTP), or plurality, scheme. In both cases, the voting scheme was formalised inside a theorem prover (Coq, [3]), and from formal proofs, in both vote counting schemes, of the fact that "every election has a winner", the authors synthesised a provably correct program that counts votes according to this scheme.

In *op.cit.*, both specifications and proofs are monolithic. This has two main disadvantages: (i) proofs become difficult to manage because of sheer size, and (ii) even small changes in the specification of the vote counting scheme necessitate to completely re-do all proofs. Moreover, it is not quite clear whether or not the approach would actually be adaptable to real-world vote counting schemes that are usually more involved than the two case studies that have been analysed.

In this paper, we show that the two deficiencies above can be remedied by introducing a more general framework that allows us to treat, and analyse, vote counting rules on an individual basis, and apply the approach to the voting

system that is used by the ANU union [11] to elect their board of representatives. In a nutshell, the general framework we present here just requires us to give formal proofs of the fact that (a) at every stage of the count, at least one rule is applicable, and (b) every rule application makes 'something smaller', i.e. leads to decrease in a well-founded ordering. Our experience with formalising a real-world voting protocol indicates that the rule-based approach lends itself to more complex voting schemes that are in fact used in real elections.

Related Work. Our work falls within the area of applying formal methods to the specification, analysis and verification of voting protocols. The mainstay of our work is *formal proof* [**?**], i.e. machine-checked assurance of our mathematical arguments.

In the literature, many different ways of formally specifying vote counting schemes, and verifying associated vote counting programs, have been put forward. What they have in common is that they involve some form of expressive logical formalism that is rich enough to express both the voting scheme and its properties. De Young and Schürmann advocate *linear logic* [6], Gore *et.al* [7] use Higher-Order logic, Cochran and Kinry use the Alloy tool [5], and Cochran has used various light-weight methods for the same task but with limited success [4]. Beckert *et.al* make a distinction between light weight (fully automated) and heavy weight (interactive) methods and concludes that the scope of fully automated methods is severely limited. Our work is carried out within the formal logic of an interactive theorem prover and clearly falls into the heavy weight category, together with the work reported in [7]. It has been argued in [4] that only heavy-weight methods will give us full assurance on functional correctness of the software used to count votes. On the downside, heavy weight methods require a significant investment in learning both the use of theorem provers and acquiring familiarity with their underlying logic. One side effect of the work reported here is that this burden is somewhat lessened as one is not required to construct fully fledged proofs about complex vote counting schemes, but instead only comparatively small proofs that pertain to individual rules.

`mention constructive logic / program extraction`

2 Rule-Based Specification of Voting Schemes

`Check definitions: they are lacking?`

It was argued in [9] that vote counting schemes are both naturally and conveniently expressed using a rule based system. The formalisation centres around a notion of *state* that is manipulated by *rules*. It is instructive to think of a state in analogy with hand counting of paper ballots. When hand counting, we would maintain e.g. the current tally of all candidates, one (or more) piles containing yet uncounted ballot papers as well as for each candidate, a pile onto which the ballot papers counted in their favour are being placed. Continuing the metaphor, a rule describes one (of generally many) action of an individual that progresses the count according to the vote counting scheme in force. An informal description of such a rule could be "take a ballot from the pile of uncounted votes,

update the tally of the candidate listed as first preference on the ballot, and place the ballot onto the pile corresponding to this candidate". In a rule-based formulation, we would construe ballots as (rank-ordered) lists of candidates and express this as follows `explain u, t, p`

$$\frac{\mathsf{state}(u_1 +\!\!+ [c{:}cs] +\!\!+ u_1, t, p)}{\mathsf{state}(u_1 +\!\!+ u_2, t', p')}$$

and stipulate that this rule is only applicable if the following side conditions are met

- $t'(c) = t(c) + 1$, $t'(d) = t(d)$ for all $d \neq c$
- $p'(c) = (c{:}cs){:}p(c) +\!\!+ [c]$, $p'(d) = p(d)$ for all $d \neq c$

where we assume a set C of candidates, $c \in C$ and write $[x_1, \ldots, x_n]$ for the list containing $x_0, \ldots, x_n$ and $l_1 +\!\!+ l_2$ for the concatenation of lists and $x{:}xs$ for the list with first (head) element x and tail xs. The side condition here is crucial, as it must be met for the rule to be applied according to the counting scheme.

States of the above form are called *intermediate* and are augmented by so-called *final* states, e.g. of the form $\mathsf{winners}(w)$, that indicate that the list w of candidates have won the election.

Given this style of specification, a *count* is a sequence of rule applications that ends in a final state, and commences in a specified initial state, usually a state where all ballots are uncounted and the current tally records zero votes for all candidates.

One crucial aspect of a *meaningful* specification of a voting protocol in this manner is the ability to proceed to a final state from any given state by means of correctly applied rules. In other words, if we can prove that *every election has a winner*. Formally, this amounts to establishing that for every initial state, there exists a sequence of correct rule applications that ends in a final state. The authors of [9] then argue that this sequence plays the role of a universally verifiable certificate that attests to the correctness of the count, and moreover, that provably correct vote counting programs can be synthesised from *constructive* formal proofs of the above termination criterion, i.e. every election has a winner. This was demonstrated using two small examples, plurality voting and simple instance of single transferable vote.

In both cases, the formal proofs of termination were given in a monolithic way, and small changes of the vote counting scheme necessitate to re-do the entire proof. In this paper, we take one step further and develop a modular framework that automatically generates termination proofs (and as a consequence, also provably correct vote counting programs), given two properties of the rule set: (a) there is at least one rule that can be applied at any non-final state of the count, and (b) non-final states are equipped with a measure (taking values in a well-founded ordering) that decreases with every rule application.

`Mention that rules themselves don't guarantee that counting is deterministic or determined.`

3 Modular Termination Proofs

As electoral authorities across the world use versions of STV that differ in small detail, it is clearly advantageous for an approach to provably correct, independently verifiable electronic vote-counting to be readily adaptable. Our experience indicates that the extent of the revisions required to adapt an existing formalisation to a very similar protocol is great, while at the same time, the features in common between the formalisations of two very different protocols – FPTP and Simple STV – are many. We therefore abstract the features in common to all protocols under the rule-based specification approach into one generic framework. We then prove meta-results about the framework which can then by instantiated with specific protocols. Under this approach, we can change the protocol without having to completely re-do a proof. Rather, it is only necessary to re-establish properties for one or two relevant rules.

We begin by defining this framework, which involves separating out the rules, rather than giving them as a single type, and then defining a termination condition local to the rules.

After establishing the generic termination framework, we demonstrate its adaptability by first applying it to the two protocols studied in [9]: FPTP and Simple STV. Finally, we extend this proof of concept to a real world vote-counting protocol, the version of STV used by The Australian National University Union Incorporated [11]. This protocol uses a common feature of STV that is not used in Simple STV, namely the assignment of *fractional transfer values* to ballots.

Our method hinges on identifying a termination condition local to the rules. This starts with the observation from working with the proof-rules formalisations of FPTP and Simple STV, as the count proceeds (i.e. the sequence of rule applications gets longer) there is always something that is decreasing. In the case of FPTP, from the initial judgement onwards, the number of uncounted votes decreases after each rule application until it reaches zero and a winner may be declared. For Simple STV it is more complicated, but there are similar observations - at every rule application, the number of uncounted votes and the number of continuing candidates, for example, are non-increasing.

`the following is garbled?` We make the understanding before of some judgements being intermediate and judgements of a certain form being final, and then define a function that assigns to a non-final judgement to value of the data in the judgement that is always decreasing (the measure of a non-final judgement). Intuitively, if the measure decreases at every rule application, and there is always a rule that can be applied to a non-final judgement, then we can prove termination – that a final judgement is always reached. We can then use the principle of well-founded induction to establish termination.

For FPTP, the less than relation on the natural numbers is a suitable well-founded order. Since there is not one piece of data always decreasing in the case of Simple STV, we use the lexicographic order on triples of natural numbers as codomain of the measure function. We describe the abstract framework and then give concrete instances for three different vote counting scheme.

Throughout the rest of the section, we fix a type of j of
`explain judgement`
judgements that will later be instantiated with the judgements of the concrete voting protocol, and set R of *rules*. Formally, every rule is a property of pairs of judgements (in Coq, a function `j -> j -> Prop`, i.e. a proposition depending on two judgements, the premiss and the conclusion of the rule), and rule r is applicable to a judgement p (that we think of as the premiss) if there is a conclusion c such that the property $r(p, c)$ holds. We also separate out final and non-final judgements by means of an abstract propositional function $\mathsf{final} : \mathsf{j} \to \mathsf{Prop}$, i.e. final judgements are precisely those $c \in \mathsf{j}$ for which $\mathsf{final}(c)$ holds.
`explain Prop / Bool`
Throughout, we fix a set R of rules, and a (measure) function $m : \mathsf{j} \to O$ where O is a set equipped with a well-founded ordering. We formalise two properties which together give a termination condition local to the rules.

Definition 1. Let dec be the property of a list of rules R such that $\mathsf{dec}(R)$ if whenever a rule holds true of two judgements, the value of the measure of the premise is greater than the value of the measure of the conclusion. Formally, we have that $r(p, c)$ implies that $m(c) > m(p)$ for all rules $r \in R$. In other words, whenever a rule is applied the measure decreases.

The second property states that (at least) one rule is applicable at any stage of the count.

Definition 2. Let app be the property of a list of rules R such that $\mathsf{app}(R)$ if for every non-final judgement, there is always a rule that may be applied. Again formally, we have that for every non-final judgement p (the premiss) there exists a judgement c (the conclusion) and a rule $r \in R$ such that $r(p, c)$ holds.

The main termination theorem is as follows. For two judgements a and b and a list of rules R, we say that a *proof* from a to b via R is a sequence of correct applications of rules in R, starting with a and ending in b.

Theorem 3. *For any set R of rules such that $\mathsf{dec}(R)$ and $\mathsf{app}(R)$ hold, and every judgement i there exists a final judgement f and a sequence of rule applications beginning in i and ending in f.*

The theorem is proved by well-founded induction, and a formal proof can be found in the Coq sources that accompany this paper. Although the proof is not mathematically deep, the computational information it contains is precisely what allows us to synthesise provably correct counting programs later.

4 Formalisation in *Coq*

We prove the main theorem of the previous section formally in Coq. The constructive nature of the Coq theorem prover then enables us to do *program extraction*, i.e. automatically construct a provably correct program from a formal termination proof.

Implementation 4. We define the type `Judgement` that captures both final and non-final states of the count. This type is left abstract (not instantiated) in the general framework, whereas in concrete instances, judgements will be states of the count. In the examples described later, judgements will be of the form $\mathsf{state}(\mathsf{u}, \mathsf{t}, \mathsf{p})$ and $\mathsf{winners}(w)$.

Where before there was an implicit notion of a 'final' judgement giving the end result of the count, we make this explicit by defining it as a property of a judgement. We also specify that this property is decidable – for every judgement there is either a proof that it is final or a proof that it is non-final. The latter is important as it engenders a function that determines whether a judgement is final or not. In general, the law of excluded middle is *not* an axiom of our constructive meta-theory as a function that decides whether a statement A or its converse holds cannot always be procured.

```
Variable Judgement : Type.
Variable final: Judgement -> Prop.
final_dec: forall j : Judgement, (final j) + (not (final j)).
```

The keywords `Variable` and `Hypothesis` designate these as abstract, and instantiating the abstract framework amounts to (among other things) giving concrete definitions for the above. Similarly, we define generic relation `wfo` on a type `WFO`, and hypothesise that this relation is well-founded, and a measure defined on the set (type) of non-final judgements. The (constructive) notion of well-foundedness is taken from the *Coq* standard library.

```
Variable WFO : Type.
Variable wfo: WFO -> WFO -> Prop.
Hypothesis wfo_wf: well_founded wfo.
Variable m: { j: Judgement | not (final j) } -> WFO.
```

A rule is defined as a relation on two judgements, where the first judgement is thought of as a premise and the second as a conclusion.

```
Definition Rule := Judgement -> Judgement -> Prop.
```

Finally we define a type of proofs, that is, sequences of correct rule applications that we think of as evidence for the fact that the final judgement in the sequence has been obtained in accordance with the given rules. As in the original paper, this will allow us to produce an independently verifiable certificate of the correctness of the count. The type of proofs is given as a dependent inductive type with two constructors, or ways of giving evidence that a judgement has the property of provability. It is parametrised by an initial judgement and a list of rules.

```
Inductive Pf (a : Judgement) (Rules : list Rule) : Judgement -> Type :=
  ax : forall j : Judgement, j = a -> Pf a Rules j
| mkp: forall c : Judgement, forall r : Rule, In r Rules ->
    forall b : Judgement, r b c -> Pf a Rules b -> Pf a Rules c.
```

The `ax` constructor, read *axiom*, says that every judgement has a proof if it is equal to the initial judgement. The second constructor `mkp`, read *make proof*, says that if there is a proof from a judgement `a` to a judgement `b`, and a rule from the list holds true of `b` and a third judgement `c`, then there is a proof from `a` to `c`.

This establishes the elements of the general framework: given a vote counting scheme, defined by a set R of rules, and an initial judgement j, the type $\mathsf{Pf}\, j\, r$ can be thought of as an indexed family, or function, that – for every other (usually final) judgement j' – represents all correctly formed sequences of rule applications that start with j and end in j'. Thus, an element of this type is evidence for the correctness of a count where the result j' has been obtained from initial state j. We now encode two properties, parametrised by a list of rules, to capture our reasoning from before.

Implementation 5. The properties are encoded as parametrised dependent function types. The first property `dec`, read *decrease*, is defined as:

```
Definition dec (Rules : list Rule) : Type :=
  forall r, In r Rules -> forall p c : Judgement, r p c ->
  forall ep : not (final p), forall ec : not (final c),
    wfo (m (mk_nfj c ec )) (m (mk_nfj p ep)).
```

We read this as saying for the rules in the list and the pairs of judgements satisfying the rules, with evidence that these judgements are non-final, the well-founded order holds for the measures of the judgements. That is, the measure of the conclusion is accessible from the measure of the premise. The function `mk_nfj` constructs non-final judgements. For the second property `app`, read *application*,

```
Definition app (Rules : list Rule) : Type :=
  forall p : Judgement, not (final  p) ->
  existsT r, existsT c, (In r Rules * r p c).
```

This says that for every judgement with evidence that it is non-final, there exists a rule and a judgement such that the rule is contained in the list and the rule holds of the two judgements.

Note that although we refer to `dec` and `app` as properties, the codomain is `Type` rather than `Prop`. This is for the same reason as using the type level existential quantifier and the type level disjunction - if we defined it as `Prop` we would lose the evidence and just have knowledge of truth or falsity whereas the type level existential quantifer allows us to reconstruct the rule. It is precisely this type-level information that allows us to extract a program from the proof of the fact that all elections have a winner.

The main result we want to show is that if these two properties hold for a list of rules, then we have *termination*. In the formalisation, termination corresponds to the existence of a term of the type `Pf a Rules c` where `c` is a final judgement. In the syntax of Coq:

```
Corollary termination:  forall Rules : list Rule,
  dec Rules -> app Rules ->
  forall a : Judgement, (existsT c : Judgement, final c * Pf a Rules c).
```

As indicated by the keyword, this is a corollary of a more general statement that stipulates that every sequence of rule applications that links a judgement a to a non-final judgement b can be extended to a final judgement (c in this case). The key stepping stone in the proof is the ability to extend every sequence of rule applications by just one rule, thereby decreasing the measure.

5 Instances of the General Framework

We demonstrate that the two examples that were treated in [9] can be seen as instances of the more general framework presented here. We then take a (simple) voting protocol, single transferable vote with fractional transfer values as used in ANU union elections, extract a rule-based specification and show that this voting scheme is also an instance of our generic approach.

5.1 First past the post

For a first simple instantiation of the vote-counting protocol, we consider simple plurality voting, and replicate the voting scheme discussed in [9] as instance of our general framework.

Implementation 6. Judgements in FPTP counting are either states or declare the election winner

```
Inductive FPTP_Judgement : Type  :=
  state : (list cand) * (cand -> nat) -> FPTP_Judgement
| winner : cand -> FPTP_Judgement.
```

where a state records the uncounted votes (we identify votes with candidates as every vote is a vote for one candidate only) and the current tally. Final judgement is of the form `winner w`, and it is immediate that every statement is either final or it is not.

```
Definition FPTP_final (a : FPTP_Judgement) : Prop := exists c, a = winner c.

Lemma final_dec: forall j : FPTP_Judgement, (FPTP_final j) + (not (FPTP_final j)).
```

For the rules, we specialise the definition of a rule to the type of judgement we defined.

```
Definition FPTP_Rule := FPTP_Judgement -> FPTP_Judgement -> Prop.
```

In contrast to [9] where the rules where absorbed into one huge, monolithic type representing runs of the vote counting scheme, here we treat, and define each rule individually. In particular, the property that rule application decreases in measure does not need to be re-established if we use the same rule in a different voting scheme. We have two rules (only), one that represents counting of a single vote, and the second determines the winner. Formally:

```
Definition count (p: FPTP_Judgement) (c: FPTP_Judgement) : Prop :=
  exists u1 t1 u2 t2, p = state (u1, t1)
    /\ (exists l1 c l2, u1 = l1 ++ [c] ++ l2 /\ u2 = l1 ++ l2 /\ inc c t1 t2)
    /\ c = state (u2, t2).
```

where `inc c t1 t2` expresses that `t2` is the tally obtained from `t1` by incrementing `c`s tally by one (and leaving all other tallies as they are). The second rule takes the form

```
Definition declare (p: FPTP_Judgement) (c: FPTP_Judgement) : Prop :=
  exists u t d, p = state (u, t) /\ u = []
    /\ (forall e : cand, t e <= t d) /\ c = winner d.
```

i.e. winners can be declared provided no other candidate has strictly more votes.

We then define the list of rules used for FPTP counting as `FPTPR = [count; declare]`, i.e containing both `count` and `declare`.

Note that there are only two rules - we don't have axioms as in [9]. This is because a starting state is specified in the type of proofs, where as it wasn't before. This means we can start with any judgement, but for a count we will obviously be entering the ballots as uncounted.

We observe of these rules that under every rule application, either the number of uncounted votes decreases or a final judgement is deduced. Therefore, the relevant well-founded order is the predecessor relation on the natural numbers. We instantiate the framework accordingly, defining first the type on which the order exists, the order and then proving the order is well-founded. The measure just needs to be defined on non-final states, and we define the measure of a non-final state as the number of uncounted votes. With this formalised, it is a matter of proving the two properties. Instantiating the general framework, we now obtain a proof of termination of FPTP counting, from which we can extract a program that not only counts in a provably correct way, but also delivers the count, i.e. the sequence of rule applications that lead to the result, as an independently verifiable certificate.

5.2 Simple STV

As a second example, we demonstrate that a simple version of STV, in fact the same that was also used as an example in [9], can also be derived as part of our more generic framework. We recall simple STV from *op.cit.*:

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.

In order to subsume simple STV as instance of the generic framework presented here, we had to adapt the type of judgement. While we still have the same *type*, we extend it to contain more *information*. In particular, we don't only maintain

a list of candidates already elected at every state of the count, we *additionally* require that this list doesn't contain more candidates than there are seats to fill. In the same vein, we also require that the tally is never greater than the quota (as candidates get elected once they have reached the quota, and future first-preference votes for the same candidate get transferred to the next preference). Technically, this amounts to replacing both the list of elected candidates and the tally with a Σ-type. Informally, we can think of it in terms of set-comprehension, a notation that is also supported by Coq:

Implementation 7. Judgements for simple STV are represented as follows in Coq

```
Inductive STV_Judgement :=
  state:                            (** intermediate states **)
    list ballot                     (* uncounted votes *)
     * (cand -> list ballot         (* assignment of counted votes to first pref candidate *)
     * { tally : (cand -> nat) | forall c, tally c <= qu }     (* tally *)
     * (list cand)                  (* continuing cands still in the running *)
     * { elected: list cand | length elected <= s}             (* elected cands *)
     -> STV_Judgement
| winners:                          (** final state **)
    list cand -> STV_Judgement.     (* election winners *)
```

A final judgement is defined to be a judgement of the second form, declaring a set of winners, and it is routine to show that finality of judgements is a decidable property. The notion of STV rule is as before

```
Definition STV_Rule := STV_Judgement -> STV_Judgement -> Prop.
```

The rules may then be defined, with an individual type for each rule. They expressed differently but correspond to the same rules as before, except for minor adjustments due to the dependent types in the judgement type and as in the case of FPTP, we dispense of the rule corresponding to the start of the count. We include the definition of the rule for excluding the weakest candidate (the full definition may be found in the Coq sources that accompany this paper):

```
Definition tl (p: STV_Judgement) (c: STV_Judgement) : Prop :=
  exists u a t h nh e d,                          (** transfer least **)
    p = state ([], a, t, h, e) /\ (* if we have no uncounted votes *)
    length (proj1_sig e) + length h > s /\        (* and there are still too many candidates *)
    In d h /\                                     (* and candidate d is still hopeful *)
    (forall e, In e h-> (proj1_sig t) d <= (proj1_sig t) e) /\   (* but all others have more votes *)
    eqe d nh h /\ (* and d has been removed from the new list of hopefuls *)
    u = a(d) /\                                   (* we transfer d's votes by marking them as uncounted *)
    c = state (u, a, t,nh, e).                    (* and continue in this new state *)
```

The remainder of the rules are written in the same form. They are omitted here but included in the code.

The well-founded order in which the measure takes values is slightly more complex that for simple plurality. For example, under the 'count one' rule, the number of uncounted votes decreases and the number of hopeful candidates remains the same, while t the 'transfer least' rule reduces the number of continuing

candidates. We therefore use the lexicographic order on triples of natural numbers as well-founded ordering and define the measure of a non-final judgement as follows

$$\mathsf{state}(u, a, t, h, e) \mapsto \big(|h|, |u|, \sum_{v \in u} |v|\big)$$

that is, a triple of the length of the list of hopeful candidates, the length of the list of uncounted votes and the sum of the lengths of the uncounted votes. In two of the five rules concerning non-final judgements, the number of hopeful candidates decreases. Of the remaining rules, the 'count one' and 'empty votes' rules both preserve the number of hopeful candidates but decrease the number of uncounted votes, while the 'transfer votes' rule preserves the number of hopeful candidates and the number of uncounted votes but decreases the sum of the lengths of the uncounted votes. As a consequence, rule application leads to decrease in the well-founded order on triples of natural numbers. In other words, the dec property applies to all rules, and it is easy to show that one rule can be applied at any point of the counting process.

In summary, supplying (formal) proofs of the app and dec property for the generic framework, we were able to obtain a proof of termination for simple STV from which we have extracted a provably correct vote counting function by simply instantiating Coq's extraction mechanism [8].

5.3 The *ANU Union* vote-counting protocol

The Australian National University Union Incorporated (the Union) uses a protocol based on a variant of STV using *fractional transfer* values. A fractional transfer value is a rational number less than 1 assigned to a candidate's surplus at the stage of transfer. In our version of simple STV, we did not take this into account. The voting procedure for the Union is outlined in section 20 of the Union constitution [11], and we report on both our experience of transcribing a real-life voting protocols into a rule-based format, and also on instantiating the generic termination proof with this particular protocol, once formalised.

With fractional transfer, the tally is the sum of the transfer values on the ballots. The formalisation draws on the method of manual counting in which there is a 'pile' of ballots corresponding to each candidate. Throughout the count, ballots are moved between the piles as candidates are eliminated and their votes are transferred. We also keep a backlog of candidates requiring their votes to be transferred. The order of transfer is important, as transfers happen in the order candidates were eliminated.

For the mathematical formalisation, we fix a set C of candidates, and represent a ballot by a pair $B = (v, w)$, where the 'vote' $v \in \mathsf{List}(C)$ is a *permutation* of the set of candidates and $w \in \mathbb{Q}$ is the 'weight' of the ballot, also known as the transfer value. We use the Droop quota $q = \frac{|b|}{s+1} + 1$, rounded upwards to the next integer.

Definition 8. If $b \in \mathsf{List}(B)$ represents the list of ballots cast and $s \in \mathbb{N}$ represents the number of seats available to be filled, then a judgement takes one of two forms:

$$(b, s) \vdash \mathsf{state}(ba, t, p, bl, e, h)$$

where $ba \in \mathsf{List}(B)$ the list of ballots requiring attention; $t : C \to \mathbb{N}$ a tally recording the votes for each candidate; $p : C \to \mathsf{List}(B)$ a 'pile' of ballots being counted towards a particular candidate; $bl \in \mathsf{List}(C)$ the 'backlog' of candidates whose votes are to be transferred; $e \in \mathsf{List}(C)$ the elected candidates; and $h \in \mathsf{List}(C)$ the list of hopeful candidates still in the running; or

$$(b, s) \vdash \mathsf{winners}(w)$$

where $w \in \mathsf{List}(C)$ represents the list of winners of the election. A judgement is *final* if it is of the form $\mathsf{winners}(w)$.

We now describe the formulation of the ANU Union protocol in the form of vote counting rules.

Definition 9. The ANU union protocol [11] is formalised by seven vote-counting rules. For each rule, we given a short description, then the formulation of the rule with side condition as bullet-point list on the right, and then provide an informal reading of the rule.

Count applies when there are ballots requiring attention, for example at the start of the count or after votes have been transferred. The ballots requiring attention are distributed amongst the candidates' piles, according to the first continuing candidate on the ballot. The candidates' tallies are updated by adding together the weights of the ballots in their updated pile. To distribute the ballots, let fcc be the 'first continuing candidate' relation,

$$\mathsf{fcc}(ba, h, c, b) \equiv b \in ba \ \wedge \ c \in h \ \wedge \ \exists l1, l2.\big(\pi_1(b) = l1 \fatsemi c \fatsemi l2 \wedge \forall d.(d \in l1 \Rightarrow d \notin h)\big)$$

holding for a list of ballots requiring attention, a list of hopeful candidates, a candidate c and a ballot b when b requires attention, and c is the first hopeful candidate on the ballot. Formally, Count is the rule on the left subject to the side condition on the right:

$$\frac{(b, s) \vdash \mathsf{state}(ba, t, p, bl, e, h)}{(b, s) \vdash \mathsf{state}(ba', t', p', bl, e, h)}(\mathsf{Count})$$

- $ba \neq \emptyset, ba' = \emptyset$.
- $\forall c, \exists l$ such that
  - $p'(c) = p(c) \fatsemi l$
  - $\forall b, b \in l \Leftrightarrow \mathsf{fcc}(ba, h, c, b)$
  - $t'(c) = \sum_{b \in p'(c)} \pi_2(b)$

> "If there are ballots requiring attention, redistribute each ballot from this pile to the pile corresponding to the first continuing candidate on the ballot. Update the tally for each candidate according to the transfer value on the ballot."

Transfer applies when there are no ballots requiring attention and no candidates that may be elected, however there is a backlog of candidates no longer in the running that need their votes transferred. As a formal rule:

$$\frac{(b,s) \vdash \mathsf{state}(va,t,p,bl,e,h)}{(b,s) \vdash \mathsf{state}(va',t,p',bl',e,h)}(\mathsf{Transfer})$$

- $va = \emptyset$
- $\forall c, c \in h \Rightarrow t(c) < qu$
- $\exists l, c$ such that
  - $bl = c{:}l$
  - $va' = p(c)$
  - $bl' = l$
  - $p'(c) = \emptyset$
  - $\forall d.(d \neq c \Rightarrow p'(d) = p(d))$

> "If there are no ballots requiring attention, none of the hopeful candidates have reached the tally and there is a backlog of candidates to have their votes transferred, take the pile of ballots for the candidate at the front of the backlog and add it to the list of ballots requiring attention. The backlog is updated by removing the head, duplication of ballots is prevented by specifying that the pile of the candidate in question is now empty, and every other pile remains unchanged."

Elect applies when there are no candidates requiring attention and there are hopeful candidates who have reached the quota to be elected. To specify that the lists of hopeful candidates and elected candidates are updated, let leqe be the relation that holds for $k, l, l' \in \mathsf{List}(X)$ if and only if l and l' are equal, except that l' additionally contains all elements of the list k.

Let $\mathsf{ordered}$ be a function ordering a list according to according to a rational-valued function f such that if $f(x) \geq f(y)$, x is before y in the list. Let map denote applying a function to all elements of a list (used here to update transfer values) of elected candidates). The elect rule then takes the following form:

$$\frac{(b,s) \vdash \mathsf{state}(va,t,p,bl,e,h)}{(b,s) \vdash \mathsf{state}(va,t,p',bl',e',h')}(\mathsf{Elect})$$

- $va = \emptyset$
- $\exists l$ such that
  - $l \neq \emptyset$
  - $|l| \leq s - |e|$
  - $\forall c.(c \in l \Leftrightarrow (c \in h \wedge t(c) \geq q))$
  - $\mathsf{ordered}(t, l)$
  - $\mathsf{leqe}(l, h', h), \mathsf{leqe}(l, e, e')$
  - $\forall c, c \in l \Rightarrow p'(c) = \mathsf{map}(\lambda(v,w).\big(v, w * \frac{t(c)-q}{t(c)}\big), p(c))$
  - $\forall c, c \notin l \Rightarrow p'(c) = p(c)$
  - $bl' = bl{:}l$

> "If there are no ballots requiring attention, and there are continuing candidates who have reached the quota (but no more than the number

of available seats), order these candidates by surplus and declare them elected by moving them from the list of hopefuls to the list of elected candidates. Update the transfer values in the piles of the newly elected candidates, while leaving the other piles unchanged. Add the list of newly elected candidates to the end of the backlog. ”

Elimination applies when there are no ballots requiring attention, no transfer backlog and too many candidates still in the running. As a formal rule:

$$\frac{(b,s) \vdash \mathsf{state}(va,t,p,bl,e,h)}{(b,s) \vdash \mathsf{state}(va',t,p',bl,e,h')}(\mathsf{Elim})$$

- $va = \emptyset, bl = \emptyset$
- $\mathsf{length}\ h + \mathsf{length}\ e > s$
- $\forall c \in h, t(c) < q$
- $\exists c$ such that
  - $\forall d \in h, t(c) \leq t(d)$
  - $h' = h \setminus [c]$
  - $va' = p(c)$
  - $\forall d, d \neq c \Rightarrow p'(d) = p(d)$
  - $p'(c) = \emptyset$

“If there are no ballots requiring attention, there is no backlog of candidates to have their votes transferred and the sum of hopeful and elected candidates exceeds the number of available seats, then take the candidate with the minimum number of votes and remove them from the hopefuls. Move their pile of ballots to the pile requiring attention, while leaving all of the other piles unchanged.”

Hopeful win declares the winners of the election in the case where the number of elected plus continuing (hopeful) candidates is no greater than the number of seats.

$$\frac{(b,s) \vdash \mathsf{state}(ba,t,p,bl,e,h)}{(b,s) \vdash \mathsf{winners}(w)}(\mathsf{Hwin})$$

- $|e| + |h| \leq s$
- $w = e \fatsemi h$

“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”.

Elected win declares the winners of the election in the case where the number of seats is the same as the number of candidates marked as elected. Define the rule:

$$\frac{(b,s) \vdash \mathsf{state}(ba,t,p,bl,e,h)}{(b,s) \vdash \mathsf{winners}(w)}(\mathsf{Ewin})$$

- $|e| = s$
- $w = e$

> "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 consider the lexicographic order on the set of triples of natural numbers and define the following measure of non-final judgements

$$m(\mathsf{state}(ba, t, p, bl, e, h)) = ((|h|, |bl|, |ba|)$$

and we can show that each rule decreases the measure so defined by simply inspecting the individual rules.

The formalisation of this protocol in Coq is very much similar to the formalisation of FPTP and simple STV so that we don't discuss it further here but refer the reader to the Coq code that comes with this paper.

6 Discussion

```
Discuss question of validating rules
   mention that having rules is easier than verifying, mention receipt.
Possibly earlier?
```

Our work is based on the idea of specifying voting protocols as rule-based systems [9]. Our contribution is two-fold, and we have demonstrated (a) that provably correct vote counting programs can also be constructed in a modular way, by separating out two properties of a rule-based specification that can be established individually, and (b) using the rule-based approach to specify and formalise a real-world voting protocol. We comment on both in turn.

Modular Generation of Provably Correct Vote Counting Code. The basic idea, and underlying technical principle, of generating provably correct vote counting code is identical to [9]: we give a constructive proof of the fact that every election has a winner, and then employ program extraction, described in [8] for the Coq-system that we are using, and [2] for a more general context. In contrast to [9], our proofs of termination are not monolithic, but both more modular and more principled. We treat each of the rules in turn, and show that their application decreases a measure that takes values in a well-founded ordering. This not only clarifies the mechanism behind the termination proofs of *op.cit.* but makes the process of synthesising vote counting programs more manageable, and enables quicker prototyping: rather than re-working large formal proofs, only changes local to some of the counting rules are required.

Formalisation of a real-world voting protocol Up to know, the only examples formalised as a rule-based system were plurality voting and a very simple version of single transferable vote. Both voting protocols are very simple in nature, and don't display any of the subtleties often found in real-world voting schemes. The ANU union voting protocol adds complexity in that ballots come with transfer values, and in form of the requirement that votes are to be transferred in a particular order. As a consequence, the formulation of this protocol

is slightly more involved, but the main leitmotif of rule-based specification still applies: every rule should formalise an action of a vote-counting officer that is in accordance with the protocol.

Conclusion. In our experience, modularising the termination proof from which vote counting programs can be constructed not only has the advantage that it becomes more modular, but it also becomes more manageable as it is broken down into smaller chunks. The formalisation of the ANU Union rules in our opinion showed the flexibility and strength of the approach, and in particular the usefulness of the guiding metaphor: every rule should embody one action of a human counting the votes. What is needed now are larger, and more case studies, in combination with a careful analysis into the efficiency and scalability of code extracted from mathematical proofs.

References

1. B. Beckert, T. Börmer, R. Goré, M. Kirsten, and T. Meumann. Reasoning about vote counting schemes using light-weight and heavy-weight methods. In *Proc. VERIFY 2014: Workshop associated with IJCAR 2014*, 2014.
2. U. Berger, H. Schwichtenberg, and M. Seisenberger. The warshall algorithm and dickson's lemma: Two examples of realistic program extraction. *Journal of Automated Reasoning*, 26(2):205–221, 2001.
3. Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.
4. D. Cochran. *Formal Specification and Analysis of Danish and Irish Ballot Counting Algorithms*. PhD thesis, 2012.
5. D. Cochran and J. R. Kiniry. Formal model-based validation for tally systems. In J. Heather, S. A. Schneider, and V. Teague, editors, *Proc. Vote-ID 2013*, volume 7985, pages 41–60. Springer, 2013.
6. H. DeYoung and C. Schürmann. Linear logical voting protocols. In A. Kiayias and H. Lipmaa, editors, *Proc. VoteID 2011*, volume 7187 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2012.
7. R. Goré and T. Meumann. Proving the monotonicity criterion for a plurality vote-counting program as a step towards verified vote-counting. In R. Krimmer and M. Volkamer, editors, *Proc. EVOTE 2014*, pages 1–7. IEEE, 2014.
8. P. Letouzey. Extraction in coq: An overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Proc. CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.
9. D. Pattinson and C. Schürmann. Vote counting as mathematical proof. In B. Pfahringer and J. Renz, editors, *Proc. AI 2015*, volume 9457 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2015.
10. A. Troelstra and D. van Dalen. *Constructivism in mathematics: an introduction.* North Holland, 1988. Two volumes.
11. T. A. Union. Constitution and board minutes, 2016. accessed May 27, 2016.