Proving the Monotonicity Criterion for a Plurality Vote-Counting Program as a Step Towards Verified Vote-Counting

Rajeev Goré
The Australian National University

Tom Meumann
The Australian National University

***Abstract*—We show how modern interactive verification tools can be used to prove complex properties of vote-counting software. Specifically, we give an ML implementation of a vote-counting program for plurality voting; we give an encoding of this program into the higher-order logic of the HOL4 theorem prover; we give an encoding of the monotonicity property in the same higher-order logic; we then show how we proved that the encoding of the program satisfies the encoding of the monotonicity property using the interactive theorem prover HOL4. As an aside, we also show how to prove the correctness of the vote-counting program. We then discuss the robustness of our approach.**

The command "`detex evote14.tex | wc`" indicates that our souce-file contains 4426 words, including the appendix.

I. Introduction

Paper-based elections consist of three main phases: printing and transporting ballot papers to polling places; collecting and transporting ballots after polling; and hand-counting ballots centrally to determine the result. Our confidence in the result is based on blind trust and scrutiny. We trust electoral officials to act honestly, but allow scrutiny by observers from political parties and independent organisations when ballots are transported, opened, and counted. That is, we rely on the difficulty of compromising all of these different non-centralised entities simultaneously. Such elections are slow to announce results, are (becoming) prohibitively expensive and impinge on the privacy of impaired voters who must be assisted by others to cast their vote. Paper ballots and hand-counting are therefore being replaced, gradually, by electronic alternatives [1], and although such vote-casting and vote-counting are very different aspects, they are often conflated into the term electronic voting.

End-to-end voter-verifiable systems attempt to provide full confidence by verifying the processed output of each phase rather than actually verifying any computer code. Such systems allow voters to verify that: their votes are cast correctly into a digital ballot; that these digital ballots are transported from the polling place to the central vote-counting authority without tampering; and that their digital ballot appears in the final tally. The methods used to guarantee these properties invariably involve sophisticated cryptographic methods, including methods for computing the sum of the encrypted votes without having to decrypt the votes themselves. But such cryptographic methods only work when the tallying process is a simple sum. No currently implemented "end to end voter-verifiable" system [2]–[5], can guarantee that votes are counted correctly using a complex preferential vote-counting method such as single transferable voting (STV).

The accepted wisdom for elections that involve complex preferential vote-counting methods, such as STV, is to publish the ballots on a web page so that they can be tallied by multiple different implementations, built by interested (political) parties. That is, in e-voting, it is not the code that we should verify, but the processed output. For example. the Australian Electoral Commission (AEC) uses a computer program to count votes cast in senate elections. The AEC makes the votes public but has refused to make the code public. Antony Green, a journalist and electoral commentator, has built his own implementation of the STV method used to count the votes. The only known "scrutiny" of the results of the previous senate election is the fact that Green's code produced the same results as those produced by the AEC computer code.

But what if the official results from the AEC differ from those of Green, or from those of the political party that loses? In particular, what if the losing party appeals to the court of disputed returns? There is no reason why the results of the AEC should be accepted over those of others.

Do we resort to time-consuming and error-prone hand-counting to resolve the discrepancy? Or do we commission someone to write yet another program? Or do we enter a complex court case to argue why the AEC code should be accepted? None of these options will engender confidence in the result.

Thus, given the complexity of preferential vote-counting methods like STV, even the most secure and most sophisticated end-to-end voter-verifiable system will still fail to gain the trust of voters if it cannot guarantee that votes are not only cast correctly and transported without tampering but that they are also counted correctly.

Here, we focus on verified vote-counting where "verification" is the process of proving that an actual computer program correctly implements a formal specification of some desirable property. We first explain the various forms of software verification that are possible today and briefly explain the pros and cons of these approaches. We then describe our work on verifying that a computer program for counting votes according to a simple plurality voting scheme meets Arrow's monotonocity criterion. We then also prove that the program counts votes correctly. The case study nicely highlights the issues involved in formal verification of software.

How does our work tie into the electoral process and how does it help to improve it?

Most preferential vote-counting methods are simplified to make it possible to count the ballots by hand since humans are notoriously bad at such mechanical tasks. The greatest simplifications are usually made to the way ballots are transferred from one candidate to another even though the simplifications are known to engender some unfairness in the final tally. Simplifications are also made in tracing back through the previous rounds when breaking ties, again even though quite simple examples can be constructed which show that these approximations can lead to unfairness. Sometimes, the result can come down to a simple coin toss at some crucial juncture.

The ability to count votes using computers opens up the possibility to design new, even more complex, voting schemes which guarantee various theoretical desiderata, and to use them in real elections. How can we be sure that the new schemes enjoy the desired properties while remaining practical for counting by computer for large numbers of votes? More importantly, how can we convince voters that the safety-net provided by hand-counting is no longer necessary?

One way is to develop the voting scheme incrementally and iteratively. We start with a simple implementation and specification and gradually add complexity as we iron out errors in the implementation and specification, and gain insights into the practicality of the desired theoretical desiderata. By involving electoral officials in this iterative process, we can ensure that the officials themselves are convinced of the correctness of the implementations beyond any doubt.

Our work has the potential to revolutionise elections using preferential methods of voting since it allows us to produce fairer, but necessarily complex, versions of vote-counting and produce computer programs that are guaranteed to implement these complex vote-counting methods correctly.

For example, if the AEC used a computer program that had been formally verified as correct, there would be a strong case to reject the conflicting results from other computer programs.

II. Various forms of Software Verification

Modern software verification methods can be broadly classified into two main categories which we shall call "light-weight" and "heavy-weight" for want of better terms.

Light-weight methods range from the fully automatic methods like software bounded model checking (SBMC) to full functional software verification using automatic annotation-based program verification tools such as VCC [6]. Both SBMC and annotation-based program verification tools involve adding the properties to be checked as pre and post condition annotations to the actual code, turning these annotations automatically into proof obligations by a compiler, and discharging the proof obligations automatically by some theorem prover. Their main advantage is that the proof-obligations are discharged fully automatically. Thus the user may have to learn some basics of how to annotate programs with pre- and post-conditions, and how to operate the verification tool, but the user does not have to be an expert in logic and formal proof. Their biggest disadvantage is that there is usually little that can be done when the verification tool fails to discharge the required proof obligations automatically. Even when the proof obligations are discharged automatically, there is no guarantee that the tool itself is sound or complete, lowering the trust that can be placed in the correctness of the program.

Heavy-weight verification involves encoding both the implementation and the specification into the logic of some theorem prover, and then proving that the encoding of the implementation implies the encoding of the specification using that theorem prover, usually interactively. The biggest advantage of this method is that we can trust the final proof completely. The disadvantage is that the user has to be expert in logic and formal proof.

III. Heavy-weight Verification Using HOL4

The verification process explored here falls under the rubric of heavy-weight verification. It involves producing a logical formalisation of both the program's requirements and the program itself in the HOL4 theorem proving assistant, then constructing a formal proof showing that the program matches the requirements. Why should we trust the HOL4 theorem proving assistant?

HOL4 is an (interactive) theorem prover based upon Dana Scott's "Logic for Computable Functions" (LCF), a mathematically rigorous logic engine consisting of 8 primitive inference rules which have been proven to be mathematically correct [7]. HOL4 implements this logic engine using approximately 3000 lines of ML code. This code has been scrutinised by experts in LCF to ensure that it correctly implements the 8 inference rules. Any complex inference rules must be constructed from the core primitive rules only. This means that proofs produced in HOL4 are highly trustworthy.

A side-effect of using an LCF-style proof assistant is that the program must be in represented in higher-order logic. It thus becomes possible to prove various results about the program. This can be used to verify the voting scheme itself with respect to various desiderata. For example it would be possible to prove that the voting scheme in question adheres to the independence of irrelevant alternatives (see [8]). It is also possible to prove comparative results between different voting schemes: for instance that voting scheme A differs from voting scheme B in only x specific situations. The ability to reason about the program in this manner is what makes this process suited to the design of fairer voting schemes which can be rigorously tested against any desired properties.

IV. Case Study

As a case study, we implement a plurality voting program and verify that it adheres to the monotonicity criterion.

A. Plurality Voting

First-past-the-post plurality voting is a voting scheme wherein each voter may vote for one candidate only, usually by marking a cross or a tick next to the desired candidate on the ballot paper. The number of votes for each candidate is tallied, and the candidate with the most votes (a relative majority) is declared elected. Note that the candidate does not need an absolute majority. Real-world voting systems vary in the way they deal with a tie, but in our simple case, no candidate is elected in the case of a tie.

B. The Monotonicity Criterion (MC)

The monotonicity criterion was originally posited by Arrow as a property of social welfare functions as follows [8]:

> "If an alternative social state x rises or does not fall in the ordering of each individual without any other change in those orderings and if x was preferred to another alternative y before the change in individual orderings, then x is still preferred to y."

A social choice procedure, such as a voting scheme or a market mechanism, can be said to either satisfy this condition or not. Reducing the available social choice procedures to preferential voting schemes or a subset thereof allows us to narrow the definition and put it in more tractable language. Thus for our purpose: "social state" is the election of a particular candidate; and "x is preferred to y" refers to a societal preference and can be changed to "x is elected".

In our plurality system, voters may only vote for one candidate, ie. rank one candidate above all others (rejecting all others equally). Thus monotonicity can be rewritten as:

> If each voter either changes his or her vote to a vote for candidate x or maintains his or her vote unchanged, and x won before any votes changed, then x will still win after the changes.

C. Verification

The verification method involves producing a logical formalisation of both the program's requirements (the vote-counting legislation) and the program itself, then constructing a formal proof showing that the software matches the specification, using HOL4.

In other words, the proof procedure involves producing the following, step-by-step:

1) Implementation: An implementation in SML of the plurality vote-counting scheme.
2) Translation: A translation of the implementation into HOL4's formal logic.
3) Specification: An encoding of MC in HOL4's logic.
4) Proof: A proof acceptable to the HOL4 theorem prover that the specification (3) holds of the translation (2).

Each of these steps is explored individually below.

1) Implementation: A plurality vote-counting program has been written in StandardML (SML), a strict functional programming language. The SML code for the plurality counting program is given in Figure 1.

This implementation makes use of the `option` type operator. Specifically, `ELECT` returns a value of type `num option`. `WINNER` also makes use of the `num option` datatype. The `option` type operator is acting in both cases as a wrapper around type `num` to allow the program to return either a number (as $\texttt{SOME}\, c$) or the lack thereof (`NONE`). The statement $\texttt{SOME}\, c$ is *not* shorthand for "there exists some c".

For simplicity, each candidate is represented by a number from 0 to $(C-1)$, and the set of votes by a list of numbers: each representing a vote for the numbered candidate. Let c_i be the i^{th} candidate and v_j be the j^{th} vote. A vote v_j is a vote for c_i iff the j^{th} member of the list `v` is equal to i. If $v_j < 0$ or $v_j \geq n$ where n is the number of candidates, then v_j is invalid.

Our implementation runs in $O(cv)$ time with number of candidates c and number of votes v. A $O(c+v)$ implementation is possible, but it was kept this way in order to maintain the program's functional purity and simplicity (thereby making it easier to reason about). Theoretically, the same results are provable of a $O(c+v)$ implementation but this is not explored here.

2) Translation into HOL4: Figure 1 shows the implementation translated into recursive definitions in HOL4. The translation between SML and HOL4 was done by hand, but was a purely mechanical process. Bar a few small syntactic differences, the translation clearly syntactically matches the SML implementation. Whether the HOL4 translation matches the SML implementation semantically is somewhat less clear. This issue is explored in more detail in section VI.

Note that the translation is a statement in higher order logic, not a program in the traditional sense. This is why the HOL4 function definitions consist of conjunctions (`/\` is the HOL4 syntax for logical 'and').

3) Specification: Formally stated in higher-order logic, the definition of monotonicity given on page 3 becomes:

$$\begin{aligned}\forall C\, w\, v\, v'.\ \Big(&(\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v)\\ \wedge\ \big(&\forall n.\, n < \texttt{LENGTH}\, v \Rightarrow (\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v')\big)\\ &\wedge\ (\texttt{ELECT}\, C\, v = \texttt{SOME}\, w)\Big)\\ &\Rightarrow (\texttt{ELECT}\, C\, v' = \texttt{SOME}\, w) \quad (1)\end{aligned}$$

where:

- v is a list representing the set of initial votes;
- v' is a list representing the set of changed votes;
- w is a number representing the winning candidate;
- C represents the number of candidates;
- $\texttt{LENGTH}\, l$ is the length of list l; and
- $\texttt{EL}\, n\, l$ is the n^{th} element of list l, where $0 \leq n < \texttt{LENGTH}\, l$.

Note that `LENGTH` and `EL` are predefined recursive functions in HOL4 and $\texttt{EL}\, 0\, (h :: t) = h$. That is, the members of the list are numbered from 0, not 1.

The first conjunct in the antecedents of the implication (the first line) states that the number of votes cannot change. The second conjunct (second line) states that each vote in the set of changed votes must be a vote for the winner, or the same as the corresponding initial vote, or both. The third conjunct (third line) states that there is a winner from the set of initial votes. The final line states that these conjuncts together imply that the winner still wins with the changed votes.

```
local
  (* Counts the number of votes in the
     given list for candidate c. *)
  fun COUNTVOTES c []     = 0
    | COUNTVOTES c (h::t) = if h = c
                            then 1 + COUNTVOTES c t
                            else 0 + COUNTVOTES c t;

  (* Finds winner from all candidates
     numbered c or lower. *)
  fun WINNER 0 v = (SOME 0, COUNTVOTES 0 v)
    | WINNER c v =
      let
        val numvotes = COUNTVOTES c v
      in
        let
          val (w, max) = WINNER (c-1) v
        in
          if numvotes > max
            then (SOME c, numvotes)
          else if numvotes = max
            then (NONE, max)
          else (w, max)
        end
      end;
in
  (* C is the number of candidates, v is the
     list of votes *)
  fun ELECT C v = if C <= 0 then NONE
                  else #1 (WINNER (C-1) v)
end;
```

(a) SML

```
val COUNTVOTES_def = Define `
  (COUNTVOTES c []     = 0) /\
  (COUNTVOTES c (h::t) = if (h = c)
                         then 1 + COUNTVOTES c t
                         else 0 + COUNTVOTES c t)`;


val WINNER_def = Define `
  (WINNER 0 v = (SOME 0, COUNTVOTES 0 v)) /\
  (WINNER c v =
    let
      numvotes = COUNTVOTES c v
    in
      let
        (w, max) = WINNER (c-1) v
      in
        if numvotes > max
          then (SOME c, numvotes)
        else if numvotes = max
          then (NONE, max)
        else (w, max))`;



val ELECT_def = Define `
  ELECT C v = if C <= 0 then NONE
              else FST (WINNER (C-1) v)`;
```

(b) HOL4

Fig. 1: Implementation of a plurality counting algorithm (a) in SML, and (b) translated into HOL4.

4) Proof: The entire proof was completed using the HOL4 theorem prover. Rather than explaining the syntax of HOL4 and how it corresponds to higher-order logic, all of the formulae in this section are given using standard higher-order logic syntax.

Let ϕ be defined as follows:

$$\phi = \Big((\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\, n < \texttt{LENGTH}\, v \Rightarrow (\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v')\big)\Big) \tag{2}$$

This allows us to rewrite the proof obligation (1) as:

$$\forall C\, w\, v\, v'.\, \big(\phi \wedge (\texttt{ELECT}\, C\, v = \texttt{SOME}\, w)\big) \Rightarrow (\texttt{ELECT}\, C\, v' = \texttt{SOME}\, w) \tag{3}$$

C is either 0 or the successor to some number (ie. $\texttt{SUC}\, x$). Examining these cases and applying some basic substitution allows us to rewrite the proof obligation (3) in terms of WINNER:

$$\forall c\, w\, v\, v'.\, \big(\phi \wedge (\texttt{FST}\, (\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w)\big) \Rightarrow (\texttt{FST}\, (\texttt{WINNER}\, c\, v') = \texttt{SOME}\, w) \tag{4}$$

The new proof obligation is that at any stage of the recursion: if w beats all other candidates examined so far with the initial votes, then w beats the same candidates with the changed votes.

To get to the core of the problem, it is desirable to go one step further and rewrite the proof obligation in terms of COUNTVOTES. In order to do this, we need a formula relating WINNER and COUNTVOTES. The following lemma states that if w beats all candidates numbered c or less, then w also has more votes than all of the said candidates and vice versa. The proof of this lemma relies upon inductive proofs of various properties of WINNER:

$$\forall c\, v\, w.\, w \leq c \Rightarrow \big((\texttt{FST}\, (\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w) \iff \forall c'.\, c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v\big) \tag{5}$$

The proof obligation (4) can thus be rewritten in terms of COUNTVOTES as follows:

$$\forall c\, w\, v\, v'.\, \big(\phi \wedge (\forall c'.\, c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v)\big) \Rightarrow (\forall c'.\, c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v' > \texttt{COUNTVOTES}\, c'\, v') \tag{6}$$

In other words we need to prove that if w has more votes than the set of lesser-numbered candidates using the initial votes, and the conditions in ϕ hold, then w also has more votes than all the aforementioned candidates using the changed votes. A structural case analysis of v and v' can now be performed (the lists being either empty or having a head and tail).

In order to make the proof fall all the way through it is necessary to prove the following properties of `COUNTVOTES`:

$$\forall w\, v\, v'.\ \phi \Rightarrow \texttt{COUNTVOTES}\ w\ v' \geq \texttt{COUNTVOTES}\ w\ v \quad (7)$$

$$\forall w\, v\, v'.\ \phi \Rightarrow (\forall c.\ c \neq w \Rightarrow \texttt{COUNTVOTES}\ c\ v \geq \texttt{COUNTVOTES}\ c\ v') \quad (8)$$

Appendix A lists all the lemmas involved in the proof and a diagram of their inter-dependencies.

V. Correctness

The astute reader will have noticed that we have not proved the correctness of our encoding of our implementation by proving that the winner is the candidate with the most number of votes. The HOL4 formula to capture this correctness statement is:

$$\forall C\, v\, w.\ w < C \Rightarrow (\texttt{ELECT}\ C\ v = w \iff \forall c'.c' \neq w \wedge c' < C \Rightarrow \texttt{COUNTVOTES}\ w > \texttt{COUNTVOTES}\ c') \quad (9)$$

Given the lemmas proved during the proof process for the monotonicity criterion, this is a quick and easy process. It has been left out for brevity.

Given the simplicity of the algorithm for plurality voting, it is questionable whether our formal proof of correctness is significant. Note, however, that the proof that our plurality voting algorithm obeys monotonicity is far from trivial.

VI. Summary and Discussion

There are two aspects worth considering when evaluating the feasibility of our verification process: the effort involved and whether the proof actually covers everything that is required. We address each in turn.

We have proved that our recursive definitions in HOL4 match our encoding of MC. Syntactically, our SML program appears equivalent to our recursive definitions. Semantic equivalence is another matter. We have no formal guarantee that our SML implementation is equivalent to our HOL4 translation, except for their syntactic similarity.

A particularly illuminating example of this conundrum is the difference between HOL4's and SML's handling of numerical types. In both programs, the candidates are represented by numbers. SML uses integers by default, which can be positive or negative: `~1`, `0`, `1`, `2` etc. HOL4, on the other hand, uses Peano numbers, which can only be `0` or the successor to some number. That is, they can only be positive: `0`, `SUC 0`, `SUC (SUC 0)` etc. The underlying representation would not matter if the same operations were defined and those operations had the same effect. This is not the case, however. $0 - 1 = 0$ is provably correct in HOL4, whilst `0 - 1` will result in `~1` in SML (`~` is unary negation in SML so `~1` means -1). We are safe however, since our SML implementation deals only with positive integers.

One way to get around this is to execute the HOL4 definitions directly. After all, the encoding in HOL4 is itself executable using HOL4's deductive rewriting engine. Unfortunately there is a large loss in efficiency when using this method. The SML implementation takes less than 7 minutes, using less than 10.5 GiB of memory, to count 250 *million* votes with 160 candidates. By contrast, with the same number of candidates, the HOL4 translation takes 40 minutes, using 14 GiB of memory, to count 25 *thousand* votes. Also, since the logical statements must be built up using the primitive core rules of logic, it is impractical to convert a list of votes into a logical statement acceptable to HOL4.

Another way would be to write the HOL4 specification first, and automatically produce the SML implementation using a verified compiler. This is a non-trivial task. There is, in fact, a project underway aimed at automating this translation: CakeML (https://cakeml.org/) [9]. It is currently under development so is not explored here, but may in future provide the missing link required.

Currently, our confidence in the correctness of our SML program rests completely on the syntactic similarity between the SML code and its HOL4 encoding, and the assumption that syntactic similarity implies semantic equivalence. As explained above, this holds for the case study explored here. For more complex voting schemes, we envisage that an iterative process may be necessary to reduce the syntactic differences between the SML code and its encoding in HOL4 (under the assumption that syntactic similarity implies semantic equivalence). This may require extending the HOL4 theorem prover to include more complex constructs from SML which may be needed to efficiently implement more complex voting schemes.

The entire process from implementation to complete verification took 3 weeks. Bear in mind that this was a learning process, with only 1–2 months-worth of prior experience with HOL4. Ultimately, 3 weeks is a short time to spend producing a piece of fully formally verified software. How this scales to more complex problems remains to be seen.

Another measure of the effort involved is the proof-to-implementation ratio, measured in lines of code (LoC). The implemented algorithm spans 24 lines whilst the proof spans 590. This gives at least 24 lines of proof for each line of implementation. Unfortunately, the final LoC measurement does not take into account the effort expended in exploring unproductive proof strategies. This makes its applicability here questionable. Nevertheless, it may be helpful when comparing the procedure to other verification methods. Assuming the ratio can be extrapolated to larger programs, verifying a 100-line program would require 2400 lines of verification.

It is also worth noting that the methodology here is not well suited to rapid prototyping. In particular, an indeterminate amount of time can be spent attempting to prove an invalid property before realising it is impossible.

VII. Conclusion

The procedure for fully formally verifying properties of vote counting algorithms is clearly feasible for small simple algorithms. It remains to be seen whether the procedure will scale to complex proportional representation systems.

The verification approach took roughly 10 weeks of full time work: 7 weeks of learning HOL4 and 3 weeks to specify and verify the code. Given the trustworthiness of the HOL4 proof assistant and the associated rigorousness of the proof, this seems a small price to pay. However, the following caveats apply. We verified a HOL-encoding of an SML program, not the SML program itself, so we have no proof of their equivalence. A visual comparison is compelling for the simple case we examined here, but might not be for a complex STV voting scheme used in real elections. The HOL4 encoding of plurality voting is itself executable, but is only feasible for small-scale elections. The CakeML project, currently under active development, may provide a solution that could be used to bridge this gap. Also, the interactive proof methodology does not lend itself to rapid prototyping since it does not provide counter-examples. Indeed, one can spend an inordinate amount of time trying to prove false conjectures before realising that they are indeed false.

VIII. Further Work

Our aim in the future is to extend this case study to formally verify the correctness of an SML implementation of Hare-Clark, a complex STV voting scheme used in a number of jurisdictions around the world, including Ireland, Australia and New Zealand.

Since submitting this paper, we have encoded the Hare-Clark Act which specifies the STV method used to count votes in the Australian state of Tasmania into approximately XXX lines of HOL. We have also written a matching program of approximately YYY lines of SML to count votes according to this method. We were able to keep the syntactic similarity between the HOL encoding and the SML program. Tests show that our SML program can easily count Z votes for C candidates in M minutes. We are therefore confident that the methodology outlined here will scale to allow us to formally verify complex real-world instances of STV as used in various jurisdictions around the world.

Acknowledgment

We thank Jeremy Dawson for his guidance in the use of the HOL4 theorem prover.

References

[1] D. W. Jones and B. Simons, *BROKEN BALLOTS: Will Your Vote Count?* CSLI Publications, Stanford, USA, 2012.

[2] Pret-A-Voter, "Prêt à Voter," http://www.pretavoter.com/, Accessed January 28, 2013.

[3] D. Chaum, "Secret-ballot receipts: True voter-verifiable elections," *IEEE Security and Privacy*, vol. 2, no. 1, pp. 38–47, 2004.

[4] Helios, "Helios," http://heliosvoting.org/.

[5] D. Chaum, A. Essex, R. T. C. III, J. Clark, S. Popoveniuc, A. T. Sherman, and P. Vora, "Scantegrity: End-to-end voter verifiable optical-scan voting," *IEEE Security & Privacy*, vol. 6, no. 3, pp. 40–46, 2008.

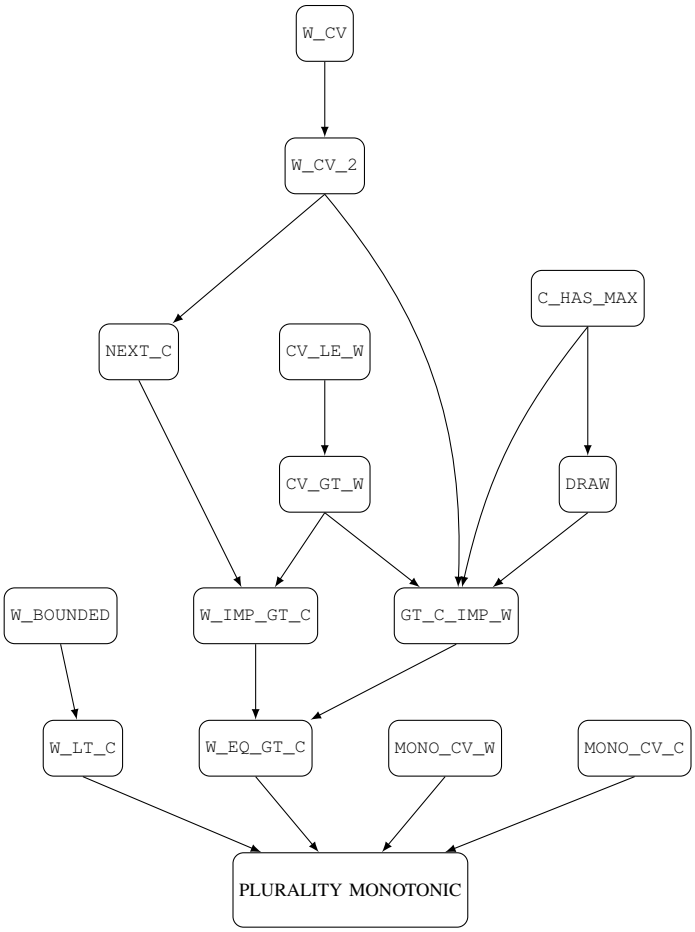

Fig. 2: Dependencies between lemmas. The proof of a lemma at the destination of an arrow relies upon the lemma at the arrow's origin.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Springer Berlin Heidelberg, 2009, vol. 5674, pp. 23–42. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03359-9_2

[7] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*. CUP, 1993.

[8] K. J. Arrow, "A difficulty in the concept of social welfare," *Journal of Political Economy*, vol. 58, no. 4, pp. pp. 328–346, 1950.

[9] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: a verified implementation of ML," in *POPL*, 2014, pp. 179–192.

Appendix

The following is a full listing of each lemma proved during the HOL4 proof. Figure 2 shows the dependencies between the various lemmas. See Section IV-C4 for an explanation of the proof.

`CV_LE_W`:

$$\forall c\, v\, c'.\ c' \leq c \Rightarrow \texttt{COUNTVOTES}\ c'\ v \leq \texttt{SND}\ (\texttt{WINNER}\ c\ v) \quad (10)$$

CV_GT_W:

$$\forall v\, c\, c'.\ c' < \texttt{SUC}\, c \wedge \texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v > \texttt{SND}\,(\texttt{WINNER}\, c\, v) \Rightarrow \texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v > \texttt{COUNTVOTES}\, c'\, v \quad (11)$$

W_BOUNDED:

$$\forall c\, v\, c'.\ c' > c \Rightarrow (\texttt{FST}\,(\texttt{COUNTVOTES}\, c\, v) \neq \texttt{SOME}\, c) \quad (12)$$

W_CV:

$$\forall c\, v\, w\, m.\ (\texttt{WINNER}\, c\, v = (\texttt{SOME}\, w, m)) \Rightarrow (\texttt{COUNTVOTES}\, w\, v = m) \quad (13)$$

W_CV_2:

$$\forall c\, v\, w.\ (\texttt{SOME}\, w = \texttt{FST}\,(\texttt{WINNER}\, c\, v)) \Rightarrow (\texttt{COUNTVOTES}\, w\, v = \texttt{SND}\,(\texttt{WINNER}\, c\, v)) \quad (14)$$

NEXT_C:

$$\forall v\, w\, c.\ (\texttt{SOME}\, w = \texttt{FST}\,(\texttt{WINNER}\, c\, v)) \wedge \texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v < \texttt{SND}\,(\texttt{WINNER}\, c\, v) \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v \quad (15)$$

W_IMP_GT_C:

$$\forall c\, v\, c'\, w.\ \big((c' \neq w) \wedge (c' \leq c) \wedge (\texttt{FST}\,(\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w)\big) \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v \quad (16)$$

C_HAS_MAX:

$$\forall v\, c.\ \exists c'.\ c' \leq c \wedge (\texttt{COUNTVOTES}\, c'\, v = \texttt{SND}\,(\texttt{WINNER}\, c\, v)) \quad (17)$$

DRAW:

$$\forall v\, c.\ (\texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v = \texttt{SND}\,(\texttt{WINNER}\, c\, v)) \Rightarrow \exists c'.\ c' \leq c \wedge (\texttt{COUNTVOTES}\,(\texttt{SUC}\, c)\, v = \texttt{COUNTVOTES}\, c'\, v) \quad (18)$$

GT_C_IMP_W:

$$\forall c\, v\, w.\ w \leq c \Rightarrow \big((\forall c'.\ c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v) \Rightarrow (\texttt{FST}\,(\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w)\big) \quad (19)$$

W_EQ_GT_C:

$$\forall c\, v\, w.\ w \leq c \Rightarrow \big((\texttt{FST}\,(\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w) = (\forall c'.\ c' \neq w \wedge c' \leq c \Rightarrow \texttt{COUNTVOTES}\, w\, v > \texttt{COUNTVOTES}\, c'\, v)\big) \quad (20)$$

W_LT_C:

$$\forall c\, v\, w.\ (\texttt{FST}\,(\texttt{WINNER}\, c\, v) = \texttt{SOME}\, w) \Rightarrow w \leq c \quad (21)$$

MONO_CV_W:

$$\forall w\, v\, v'.\ (\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\ (n < \texttt{LENGTH}\, v) \Rightarrow ((\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v'))\big) \Rightarrow \texttt{COUNTVOTES}\, w\, v' \leq \texttt{COUNTVOTES}\, w\, v \quad (22)$$

MONO_CV_C:

$$\forall w\, v\, v'.\ (\texttt{LENGTH}\, v' = \texttt{LENGTH}\, v) \wedge \big(\forall n.\ (n < \texttt{LENGTH}\, v) \Rightarrow ((\texttt{EL}\, n\, v' = w) \vee (\texttt{EL}\, n\, v = \texttt{EL}\, n\, v'))\big) \Rightarrow \forall c.\ c \neq w \Rightarrow \texttt{COUNTVOTES}\, c\, v \geq \texttt{COUNTVOTES}\, c\, v' \quad (23)$$