Verified Certificate Checking for Counting Votes

Milad K. Ghale[1(✉)], Dirk Pattinson[1], Ramana Kumar[2], and Michael Norrish[3]

[1] Australian National University, Canberra, Australia
{milad.ketabghale,dirk.pattinson}@anu.edu.au
[2] Data61, CSIRO and UNSW, Kensington, Australia
ramana.kumar@cl.cam.ac.uk
[3] Data61, CSIRO, and ANU, Canberra, Australia
michael.norrish@data61.csiro.au

Abstract. We introduce a new framework for verifying electronic vote counting results that are based on the Single Transferable Vote scheme (STV). Our approach frames electronic vote counting as certified computation where each execution of the counting algorithm is accompanied by a certificate that witnesses the correctness of the output. These certificates are then checked for correctness independently of how they are produced. We advocate verification of the verifier rather than the software used to produce the result. We use the theorem prover HOL4 to formalise the STV vote counting scheme, and obtain a fully verified certificate checker. By connecting HOL4 to the verified CakeML compiler, we then extract an executable that is guaranteed to behave correctly with respect to the formal specification of the protocol down to machine level. We demonstrate that our verifier can check certificates of real-size elections efficiently. Our encoding is modular, so repeating the same process for another different STV scheme would require a minimal amount of additional work.

1 Introduction

The main contribution of this paper is a new framework for verifiably correct vote counting. Electronic voting is becoming more and more prevalent worldwide. But almost scandalously, the current state of affairs leaves much to be desired, given that the public vote is a cornerstone of modern democracy. Indeed electronic techniques as they are used now may be seen as a step back from traditional paper based elections.

For example, the vote counting software that is used in Australia's most populous state, New South Wales, was found to contain errors that had an impact in at least one seat that was wrongly filled with high probability. This was reported in specialist publications [5] as well as the national press [3].

When counting ballots by hand, the counting is monitored by scrutineers, usually members of the general public or stakeholders such as party representatives. In contrast, computer software that is used to count ballots merely produces a final result. Moreover, in many cases, the source code of these programs

R. Piskac and P. Rümmer (Eds.): VSTTE 2018, LNCS 11294, pp. 69–87, 2018.
https://doi.org/10.1007/978-3-030-03592-1_5

is commercial in confidence, and there is no evidence of the correctness of the count that could be seen as analogous to scrutineers in traditional, paper-based vote counting.

It is universally recognised that transparent and verifiable vote counting is a key constituent to establish trustworthiness, and subsequently trust, in the final outcome. The computer-based methods currently in use fail to meet both expectations.

In the literature on electronic voting, the notion of *universal verifiability* of vote counting (any voter can check that the announced result is correct on the basis of the published ballots [14]) has long been recognised as being central, both for guaranteeing correctness, and building trust, in electronic elections. This notion has three subproperties; verifiability of casting votes as intended by voters, recording votes as intended, and counting votes as recorded [8]. The aim of this paper is only to address the last property, namely *verifiability of the tallying process*.

The approach presented here combines the concept of certifying algorithms [17] with formal specification and theorem proving to address this challenge. In a nutshell, a *certifying algorithm* is an algorithm that produces, with every execution, an easily-verifiable certificate of the correctness of the computed result. This certificate can then be scrutinised by a verifier, independently of the tools, hardware or software that were used to create the certificate.

Our focus in this paper is on the *certificate verifier*. We briefly discuss a concise formal specification of *single transferable vote* (STV), a complex, preferential voting system used e.g. in Ireland, Malta, New Zealand and Australia for multi-seat constituencies. From this specification, we develop a notion of *certificate* so that correct certificates guarantee correctness of election results. The main body of our work concerns the *verifier* (certificate checker), and we present a synthesis of the verifier that is itself fully verified down to the machine-code level.

This proceeds in four steps.[1] First, we formalise the vote counting protocol as a sequence of steps inside the HOL theorem prover where every step corresponds to an action taken by a counting officer in a paper-based setting. There are two kinds of stages that we call *judgements* in analogy to typing assertions in type theory. Final judgements just declare the set of winners. NonFinal judgements represent the current state of the count as a tuple, corresponding to a snapshot of the stage of the count in a paper-based setting. The formalisation of the voting protocol then takes the form of *rules* that specify how to advance from one judgement to the next, thereby progressing the count. The applicability of particular rules are described by side conditions that are in turn formalised by HOL predicates. A correct certificate is then simply a sequence of judgements where each judgement is justified through its predecessor by means of a correct rule application. The task of the verifier is then simply to process a list of judge-

[1] Source code of the formalisation can be found at https://github.com/MiladKetabGhale/Checker.

ments and ascertain that this is indeed the case. In particular, our specification of rules is purely descriptive.

Second, in tandem with the logical specification of each rule, we define a boolean-valued function that checks whether or not the rule has been applied correctly. This then directly gives rise to the verifier that, at every step, just checks whether any of the rules is applicable, using the corresponding boolean-valued function.

Third, we establish correspondence between the logical definitions and their computational counterparts. This boils down to formally establishing that the logical specification holds if and only if the boolean-valued function returns true, which in turn implies the correctness of the certificate verifier. This allows us to conclude that a valid certificate indeed implies that the election protocol has been carried out in accordance to the specification.

In the fourth, and last step, we synthesise an implementation of the verifier and produce a proof of this implementation's correctness. This is achieved by using proof-producing synthesis [18] of CakeML code from the HOL definitions, then using the verified CakeML compiler [22] to produce the machine code. To perform computation on an actual certificate, we define the formal syntax for certificates, and a parser in HOL, that we combine with the I/O mechanisms of CakeML to obtain the verifier. The result is an executable verifier that provably validates a certificate if and only if the certificate witnesses a correct execution of the vote counting protocol.

In summary, our slogan is "verify the verifier". Rather than verifying the *program* that performs an election count, we demand that a program produces a certificate that we can then independently verify. This has several advantages. For one, it is much less labour intensive to verify the verifier, compared with verifying the counting program. Second, having verifiable certificates at hand establishes the count-as-recorded property [8]. Third, we achieve correctness over a minimal trust base through the use of CakeML.

In the remainder of the paper, we describe our framework in detail and demonstrate that it can handle real-world size elections by evaluating it on historical data of elections conducted in Australia.

2 The Protocol and Its HOL Formalisation

Single Transferable Vote is a preferential voting scheme that is used in multi-seat constituencies. Voters rank (possibly a subset of) candidates by assigning numerical preferences to candidates where no two candidates may be given the same preference. This allows us to represent ballots as duplicate free lists of candidates where the list order reflects the preference order.

Each election defines a *quota*, i.e. a minimal set of votes that a candidate must receive in order to be elected. The count starts by counting all voters' *first preferences*, and candidates who reach the quota are elected, but in general there will still be seats to fill. This is effected by two mechanisms:

1. Transfer of surplus votes. Votes for elected candidates in excess of the quota are distributed to (and counted in favour of) the next preference listed on the ballot.
2. Elimination of candidates. The candidate with the *least* number of first preferences is eliminated from the election, and their votes are then distributed to the next listed preference on the ballot.

We give a precise definition of STV below. The main idea of distributing surplus votes is to account for *additional* candidates favoured by a voter if their first preference is already elected, whereas elimination honours voters' follow-on preferences if their first preference *cannot* be elected. Of course, the key question is precisely which ballots should be considered surplus and distributed to the next preferences, as follow-on preferences will generally differ. This is the purpose of a third mechanism:

3. Fractional Transfer. *All* surplus votes are transfered to the next preference, but at a *reduced* weight that is proportional to the size of the surplus.

For example, if a candidate exceeds the quota by 20%, all first preference votes for that candidate are re-assigned a weight of 0.2 and injected back into the count, and regarded as first-preference votes for the subsequently listed candidate. In other words, the number of first preference votes for a candidate is the sum of the *weights* of ballots where that candidate is listed as first preference. The initial weight of all ballots is 1.

There are various versions of STV. They mainly differ in how and when ballots are transferred and candidates are elected, the calculation of the transfer value, and the various tie breaking methods used to determine which candidate is to be excluded, and the quota being used. Here, we deal with a generic version of STV that incorporates all three mechanisms outlined above, and is very similar to the method used to elect members of the Australian Senate, and incidentally also to the scheme used elect the representatives of the student union at the Australian National University. Throughout, we do not assume a particular definition of the quota, but take this as a parameter. Design decisions in the precise formulation of the scheme are resolved as follows:

Step-by-step surplus transfer. Surplus votes of elected candidates that have exceeded the quota are transferred in order of number of first preferences received. That is, surplus votes of the candidate with the largest number of first preferences are transferred first.

Electing after each transfer. After each transfer of surpluses, candidates that reach the quota after surplus votes are being elected immediately.

The description of the formal protocol that we are analysing uses the following terminology. A *continuing candidate* is a candidate that has neither been elected nor eliminated. The *first preference* of a ballot is the most preferred continuing candidate, and the *transfer value* of a ballot is the fractional weight of a ballot. We keep track of the following data throughout:

- the set of *uncounted ballots*
- a *tally* for each candidate, the sum of the transfer values of all ballots counted in the candidate's favour
- a *pile* for each candidate that contains all ballots counted in favour of the respective candidate
- an queue of candidates that await surplus transfer

Initially, the queue for surplus transfer, as well as the piles associated to the individual candidates are empty, all ballots are uncounted, and all candidates are continuing. From this initial state, the protocol proceeds as follows:

1. determine the set of *formal* ballots, i.e. those ballots that represent a total order of preferences over a subset of candidates, each of which receives an initial transfer value of 1.
2. determine the number of first preference votes (the tally) for each continuing candidate. In doing this, record which vote is counted for which candidate by adding the ballot paper to the respective pile.
3. if there are unfilled seats, all candidates that have reached the quota are elected, and are added to the transfer queue in order of their tally.
4. if all the vacancies have been filled, counting terminates and the result is announced.
5. if the number of unfilled vacancies equals or exceeds the number of continuing candidates, all continuing candidates are elected and the result is announced.
6. if there are still vacancies, all ballots are counted, and the transfer queue is not empty, remove the first candidate from the transfer queue and transfer their votes (the votes on their pile) to the next preference by declaring these votes to be uncounted and the transfer value given by

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

   Subsequent transfer values are computed as the product of the current value with previous transfer value.
7. if there are still vacancies, all ballots are counted, and all surplus votes are transferred, choose the candidate with the least amount of first preference votes and exclude that candidate from the set of continuing candidates. All votes counted in favour of the eliminated candidate are transferred to the next preference (with unchanged transfer value).

The purpose of setting aside the ballots counted for particular candidates in the second step is precisely for the purpose of possibly transferring these ballots later, in case the candidate is either elected or eliminated.

2.1 An Example Certificate

As argued in the introduction, in the framework of certified computation, each step of the protocol is evidenced. Each of the steps outlined above is formalised as

a rule that progresses the count. Rules have side conditions (e.g. candidates having reached the quota, or all ballots being counted) and rule application changes the data that we track throughout the count (e.g. updating the tally, or removing a candidate from the set of continuing candidates). We given an example certificate in Fig. 1. Here, we have a small election with three candidates A, B and C, and an initial set of ballots containing $b_1 = ([A, C], 1)$, $b_2 = ([A, B, C], 1)$, $b_3 = ([A, C, B], 1)$, $b_4 = ([B, A], 1)$, $b_5 = ([C, B, A], 1)$. Each ballot is a pair, where the first component is a preference-ordered list of candidates, and the second is the transfer value (initially set to 1). The certificate consists of a *header* that specifies the quota as a fraction (computed according to the Droop quota [10]), the number of seats to be filled, and the list of candidates being voted on. The fourth line is the election result, and the remainder of the certificate consists of the intermediate steps that lead to this outcome.

The certificate records every step of the count, where a step corresponds to a rule application, and the rules themselves are modelled on valid actions that of counting officers to progress the count. The inspection of a certificate therefore corresponds to witnessing all the individual steps that take place in a hypothetical counting station.

Intermediate stages of the count record six pieces of information, separated by semicolons: the ballots the are still to be counted, the tallies of all candidates, the ballots counted in favour of each candidate, the transfer queue, and finally the sets of continuing and elected candidates. We briefly illustrate the protocol using the certificate in Fig. 1 as an example, going though the protocol step-by-step.

count. First preferences for each candidates are computed, and ballots counted in favour of particular candidates are placed onto that candidate's pile. Here, A is the first preference on b_1, b_2, and b_3 (leading to a tally of 3), and B receives b_4, and C receives b_5. Tallies are updated so that tally of A becomes 3, and B and C each reach 1.

elect. Candidate A exceeds the quota, and is elected. The transfer value of all ballots counted in A's favour changes to $1/9$ according to formula (1). The updated pile of A reflects this change in transfer values, and now contains ($[A, C]$,

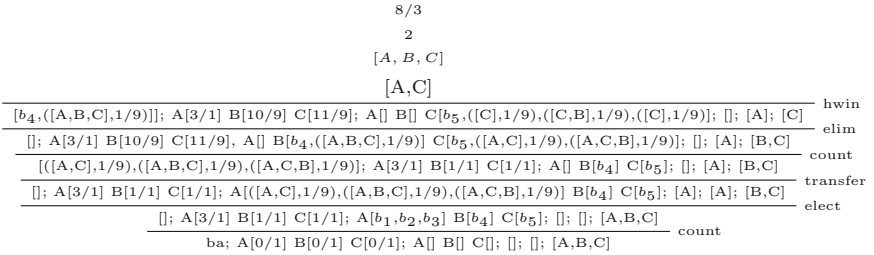


Fig. 1. Example certificate

1/9), ([A, B, C], 1/9), and ([A, C, B], 1/9). The data associated with B and C doesn't change.

transfer. As there are vacancies and no one else has reached or exceeded the quota, the surplus of A is dealt with. The list of uncounted ballots is updated to contain the votes for A (with transfer values updated in the previous step).

count. As there are uncounted ballots (again), tallies are updated. As A is no longer a continuing candidate, the votes are counted in favour of the highest-ranked continuing candidate. That is, C receives two new votes (each of value 1/9) which are ([A, C], 1/9) and ([A, C, B], 1/9). Candidate B receives one vote, which is ([A, B, C], 1/9).

elim. No continuing candidate has reached the quota, one vacancy is still unfilled, and all ballots are (again) counted. Hence the candidate with the lowest tally is eliminated (in this case, B) and their votes (with unchanged transfer values) are again injected into the count.

hwin. The only continuing candidate, that is C, is elected and as we have filled all the vacancies, a final stage has been obtained.

To validate certificates of this form, we first parse the textual representation into actual data structures, and then check the certificates for correctness on the basis of a HOL formalisation that we now describe.

2.2 The HOL Formalisation

Elections are *parameterised* by the data in the header (candidates, quota and number of vacancies) that remain constant throughout the count. We use the term *judgement* for the data-structure representation of the various stages of the count. They come in two flavours: final judgements announce the winners, and non-final judgements are intermediate stages of the execution of the protocol.

Definition 1 (Judgements). *We formalise judgements as a datatype with two constructors. The first constructor,* Final *w represents a final stage of the computation, where w is the final list consisting of all of the declared elected candidates. The second constructor,* NonFinal *(ba,t,p,bl,e,h) is an intermediate stage of the computation, where ba is the list of uncounted ballots at this point, t is the tally list recording the number of votes of each candidate has received up to this point, p is the pile list of votes assigned to each candidate, bl is the list of elected whose surplus have not yet been transferred, e is the list of elected candidates by this point, and h is the list of continuing (hopeful) candidates up to this stage.*

$$
\begin{aligned}
&\mathit{judgement} = \\
&\quad \mathsf{NonFinal}\ (\mathit{ballots} \times \mathit{tallies} \times \mathit{piles} \times \mathit{cand\ list} \times \mathit{cand\ list} \times \mathit{cand\ list}) \\
&\quad \mid \mathsf{Final}\ (\mathit{cand\ list})
\end{aligned}
$$

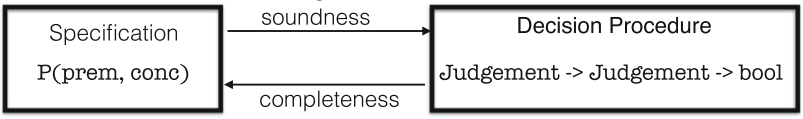
We use lists (instead of sets, or multisets) mainly for convenience of formalisation in HOL, but this is not used in an essential way either in the definition, or

in the formalisation, of the counting rules that we give later. By choosing to formalise the tally and pile as lists rather than functions operating on the list of candidates, judgements become an instance of the equality type class which we use later on in specification and reasoning about counting rules. Additionally, this formulation reduces the gap between an actual certificate and its abstract syntactic representation which we refer to as a *formal certificate*.
As a (formal) certificate consists of a sequence of judgements, each of which represents a state of the count, we need to verify the correctness of the transitions between successive judgements. Each rule consists of three main components:

- a specification of how premiss and conclusion relate
- side conditions that specifies when a rule is applicable
- a number of implicit assertions that guarantee the integrity of the data.

For example, we expect a valid certificate to have no duplication in the list of elected or continuing candidates, and every candidate must have only one tally and one pile at every non-final judgement.

Crucially, the specification of the counting rules is purely *descriptive*. To effectively check certificates, we augment each (specification of a) rule with an actual decision procedure that, given two judgements, returns either true or false, depending on whether the rule is applicable or not. The decision procedure and the formal specification are connected by (formal) proofs of soundess and completeness, as shown in the figure below.



Here, *soundness* refers to the fact that the decision procedure only stipulates that a rule has been correctly applied if the application is in accordance with the specification and *completeness* says that this will happen whenever the rule is applicable. The decision procedures are actual functions in HOL that we then translate and extract using CakeML to guarantee machine-level correctness, and both soundness and completeness are established formally in HOL. We illustrate this in detail with the elimination rule.

Integrity Constraints. The integrity constraints for the elimination rule are identical to those of other rules. For example, the name of each candidate appears only once in the initial list of competing candidates. Also, at every stage of the count, every candidate has exactly one tally and one pile (of votes counted in their favour). Therefore, if a judgement in a certificate maliciously allocates no tally, or more than one tally for a single candidate, this error is detected and the certificate is rejected as invalid. We express the fact that tallies need to be recorded for every candidate as follows:

$$\mathsf{Valid\_PileTally}\ t\ l \iff \forall c.\ \mathsf{mem}\ c\ l \iff \mathsf{mem}\ c\ (\mathsf{map\ fst}\ t)$$

The above predicate is paired with computational twins, and soundness and completeness connect both. Here, given lists t and l, the function

Valid_PileTally_dec1 decides if every first element of each pair in t is a member of l.

$$\begin{array}{l}\mathsf{Valid\_PileTally\_dec1}\ [\,]\ l \iff \mathsf{true}\\ \mathsf{Valid\_PileTally\_dec1}\ (h{::}t)\ l \iff \mathsf{mem}\ (\mathsf{fst}\ h)\ l \wedge \mathsf{Valid\_PileTally\_dec1}\ t\ l\end{array}$$

Additionally, the function Valid_PileTally_dec2 determines if each element of l appears as the first component of a pair in t.

$$\begin{array}{l}\mathsf{Valid\_PileTally\_dec2}\ t\ [\,] \iff \mathsf{true}\\ \mathsf{Valid\_PileTally\_dec2}\ t\ (l_0{::}ls) \iff\\ \quad \mathsf{if\ mem}\ l_0\ (\mathsf{map\ fst}\ t)\ \mathsf{then\ Valid\_PileTally\_dec2}\ t\ ls\\ \mathsf{else\ false}\end{array}$$

We prove that the formal specification Valid_PileTally corresponds with the functions Valid_PileTally_dec1 and Valid_PileTally_dec2. Therefore we ensure that tallies and piles are distinctively allocated to candidates.

$$\vdash \mathsf{Valid\_PileTally}\ t\ l \iff \mathsf{Valid\_PileTally\_dec1}\ t\ l \wedge \mathsf{Valid\_PileTally\_dec2}\ t\ l$$

Side Conditions. Item 7 of the protocol on Page 5 specifies when and how a candidate shall be eliminated from the election. It stipulates that

a. there are still seats to fill
b. there are no votes to count at this stage and there are no pending transfers
c. the candidate c has the least tally
d. eliminate the candidate c
e. votes of the eliminated candidate c are transferred according to the next preference with the same transfer value.

To illustrate how clauses of the protocol are formalised explicitly, we explain the way that we have specified item (d) inside HOL. We introduce the predicate equal_except which formally asserts when two lists are equal except for one exact element.

$$\begin{array}{l}\mathsf{equal\_except}\ c\ l\ nl \iff\\ \quad \exists\, l_1\ l_2.\ l = l_1 +\!\!+ l_2 \wedge nl = l_1 +\!\!+ [c] +\!\!+ l_2 \wedge \neg\mathsf{mem}\ c\ l_1 \wedge \neg\mathsf{mem}\ c\ l_2\end{array}$$

The computational twin of this definition decides whether two list match with the exception of one element. This is the function equal_except_dec.

$$\begin{array}{l}\mathsf{equal\_except\_dec}\ c\ [\,] = [\,]\\ \mathsf{equal\_except\_dec}\ c\ (h{::}t) = \mathsf{if}\ c = h\ \mathsf{then}\ t\ \mathsf{else}\ h{::}\mathsf{equal\_except\_dec}\ c\ t\end{array}$$

We formally establish that this function implements the specification given by the equal_except predicate.

$$\vdash \mathsf{mem}\ c\ h \wedge \mathsf{distinct}\ h \Rightarrow \mathsf{equal\_except}\ c\ (\mathsf{equal\_except\_dec}\ c\ h)\ h$$

Moreover, modulo extensional equality, the function equal_except_dec is unique.

$$\vdash \mathsf{mem}\ c\ h_2 \wedge \mathsf{equal\_except}\ c\ h_1\ h_2 \Rightarrow h_1 = \mathsf{equal\_except\_dec}\ c\ h_2$$

Having defined the implicit integrity constraints, and the explicit side conditions in the definition of *elimination*, we can present the formalisation of this rule in HOL as a predicate.

The ELIM_CAND rule specifies what it means to legitimately eliminate a given candidate c. It relates three data items: a candidate, a triple composed of three fixed parameters which are the quota, vacancies, and the initial list of candidates, and two judgements j_1 and j_2 (the premiss and the conclusion of the rule).

$$\begin{array}{l}
\mathsf{ELIM_CAND}\ c\ (qu,st,l)\ j_1\ j_2 \iff \\
\quad \exists\, t\ p\ e\ h\ nh\ nba\ np. \\
\qquad j_1 = \mathsf{NonFinal}\ ([],t,p,[],e,h) \wedge \mathsf{Valid_Init_CandList}\ l \wedge \\
\qquad (\forall\, c'.\ \mathsf{mem}\ c'\ (h +\!\!+ e) \Rightarrow \mathsf{mem}\ c'\ l) \wedge \mathsf{distinct}\ (h +\!\!+ e) \wedge \mathsf{Valid_PileTally}\ p\ l \wedge \\
\qquad \mathsf{Valid_PileTally}\ np\ l \wedge \mathsf{length}\ (e +\!\!+ h) > st \wedge \mathsf{length}\ e < st \wedge \\
\qquad \mathsf{distinct}\ (\mathsf{map\ fst}\ t) \wedge \mathsf{Valid_PileTally}\ t\ l \wedge \\
\qquad (\forall\, c'.\ \mathsf{mem}\ c'\ h \Rightarrow \exists\, x.\ \mathsf{mem}\ (c',x)\ t \wedge x < qu) \wedge \mathsf{mem}\ c\ h \wedge \\
\qquad (\forall\, d.\ \mathsf{mem}\ d\ h \Rightarrow \exists\, x\ y.\ \mathsf{mem}\ (c,x)\ t \wedge \mathsf{mem}\ (d,y)\ t \wedge x \leq y) \wedge \mathsf{equal_except}\ c\ nh\ h \wedge \\
\qquad nba = \mathsf{get_cand_pile}\ c\ p \wedge \mathsf{mem}\ (c,[])\ np \wedge \\
\qquad (\forall\, d'. \\
\qquad\quad d' \neq c \Rightarrow \\
\qquad\quad\ \forall\, l.\ (\mathsf{mem}\ (d',l)\ p \Rightarrow \mathsf{mem}\ (d',l)\ np) \wedge (\mathsf{mem}\ (d',l)\ np \Rightarrow \mathsf{mem}\ (d',l)\ p)) \wedge \\
\qquad j_2 = \mathsf{NonFinal}\ (nba,t,np,[],e,nh)
\end{array}$$

The first and the fourth component of j_1 which correspond to the list of uncounted ballots and the backlog are both empty. This realises the condition (a) stated above. It is also required that h the list of continuing candidates in the premise j_1, and nh the updated list of continuing candidates in j_2 satisfy the predicate equal_except so that condition (d) is met. Each of the conjuncts in the definition of ELIM_CAND encapsulates part of the item 7 in the protocol.

Similar to the case of equal_except, for each of the conjuncts, we define a computational counterpart and prove the equivalence of the conjunct with its computational realisation. Conjunction of these computational definitions is ELIM_CAND_dec, which is the computational equivalent of the predicate ELIM_CAND.

$$\begin{array}{l}
\mathsf{ELIM_CAND_dec}\ c\ (qu,st,l)\ (\mathsf{NonFinal}\ (ba,t,p,bl,e,h))\ (\mathsf{NonFinal}\ (ba',t',p',bl',e',h')) \iff \\
\quad \mathsf{null}\ ba \wedge \mathsf{null}\ bl \wedge \mathsf{null}\ bl' \wedge t = t' \wedge e = e' \wedge \mathsf{length}\ (e +\!\!+ h) > st \wedge \mathsf{length}\ e < st \wedge \\
\quad \neg\mathsf{null}\ l \wedge \mathsf{distinct}\ l \wedge \mathsf{list_MEM_dec}\ (h +\!\!+ e)\ l \wedge \mathsf{distinct}\ (h +\!\!+ e) \wedge \\
\quad \mathsf{Valid_PileTally_dec1}\ p\ l \wedge \mathsf{Valid_PileTally_dec2}\ p\ l \wedge \mathsf{Valid_PileTally_dec1}\ p'\ l \wedge \\
\quad \mathsf{Valid_PileTally_dec2}\ p'\ l \wedge \mathsf{distinct}\ (\mathsf{map\ fst}\ t) \wedge \mathsf{Valid_PileTally_dec1}\ t\ l \wedge \\
\quad \mathsf{Valid_PileTally_dec2}\ t\ l \wedge \mathsf{mem}\ c\ h \wedge \mathsf{less_than_quota}\ qu\ t\ h \wedge \\
\quad h' = \mathsf{equal_except_dec}\ c\ h \wedge \mathsf{bigger_than_cand}\ c\ t\ h \wedge ba' = \mathsf{get_cand_pile}\ c\ p \wedge \\
\quad \mathsf{mem}\ (c,[])\ p' \wedge \mathsf{subpile1}\ c\ p\ p' \wedge \mathsf{subpile2}\ c\ p'\ p \\
\mathsf{ELIM_CAND_dec}\ c\ v_0\ (\mathsf{Final}\ v_1)\ v_2 \iff \mathsf{false} \\
\mathsf{ELIM_CAND_dec}\ c\ v_3\ (\mathsf{NonFinal}\ v_{11})\ (\mathsf{Final}\ v_5) \iff \mathsf{false}
\end{array}$$

By drawing upon the correspondence established between conjuncts of the elimination specification and computational counterpart, we prove that $\vdash$ELIM_CAND_dec = ELIM_CAND. The same procedure is followed to achieve formal specification, computational definitions, and their correspondence for the rest of counting rules.

2.3 The Certificate Verifier

Clearly, one way to verify the result of a computation is to simply re-compute (possibly using a verified program) [2]. While this makes perfect sense for a *deterministic* program, voting protocols generally employ tie-breaking techniques that lead to non-determinism. In the case of STV, for example, this applies when two candidates are tied for exclusion. In this situation it is permissible to eliminate *either* of the candidates. From the perspective of certified computation, this is a non-issue, as the certificate simply records which choice has been made (and why this choice is permissible). Compared to simply re-computing, the verification of a certificate provides another significant advantage: in case of diverging results, we gain information on precisely what step of the (incorrect) computation is to blame for the wrong result. Computationally, the additional advantage is simplicity and speed: the verification of the verifier is considerably simpler than that of a fully-fledged implementation, and certificate checking is also generally faster than re-computing.

The verification of certificates comprises two steps. First we need to validate whether the first judgement of the certificate is a valid initial state of the count. A valid initial judgement is one where candidate's tally is zero, their piles are empty, and both the transfer queue and the list of elected candidates are both empty as well.

$$\begin{aligned}&\mathsf{initial\_judgement}\ l\ j\ \Longleftrightarrow\\ &\ \exists\, ba\ t\ p\ bl\ e\ h.\\ &\quad j = \mathsf{NonFinal}\ (ba,t,p,bl,e,h) \wedge (\forall\, c.\ \mathsf{mem}\ c\ (\mathsf{map\ snd}\ t) \Rightarrow c = 0) \wedge\\ &\quad (\forall\, c.\ \mathsf{mem}\ c\ (\mathsf{map\ snd}\ p) \Rightarrow c = [\,]) \wedge bl = [\,] \wedge e = [\,] \wedge h = l\end{aligned}$$

Second, we check whether transitions from one judgement to the next is according to one of the rules that define the count.

$$\begin{aligned}&\mathsf{Valid\_Step\_Spec}\ params\ j_0\ j_1\ \Longleftrightarrow\\ &\ \mathsf{HWIN}\ params\ j_0\ j_1 \vee \mathsf{EWIN}\ params\ j_0\ j_1 \vee \mathsf{COUNT}\ params\ j_0\ j_1 \vee\\ &\ \mathsf{TRANSFER}\ params\ j_0\ j_1 \vee \mathsf{ELECT}\ params\ j_0\ j_1 \vee\\ &\ \exists\, c.\ \mathsf{mem}\ c\ (\mathsf{snd}\ (\mathsf{snd}\ params)) \wedge \mathsf{ELIM\_CAND}\ c\ params\ j_0\ j_1\end{aligned}$$

We can therefore check whether a transition from one judgement to the next is correct by simply considering the disjunction of all rules.

$$\begin{aligned}&\mathsf{Valid\_intermediate\_judgements}\ params\ J\ \Longleftrightarrow\\ &\ J \neq [\,] \wedge (\exists\, w.\ \mathsf{last}\ J = \mathsf{Final}\ w) \wedge\\ &\ \forall\, J_0\ J_1\ j_0\ j_1.\ J = J_0 +\!\!+ [j_0;\ j_1] +\!\!+ J_1 \Rightarrow \mathsf{Valid\_Step\_Spec}\ params\ j_0\ j_1\end{aligned}$$

Putting the specification of a valid initial judgement with valid sequence of judgements together, we obtain the specification for a valid certificate:

$$\begin{aligned}&\mathsf{Valid\_Certificate}\ \mathit{params}\ [\,] \iff \mathsf{false}\\&\mathsf{Valid\_Certificate}\ \mathit{params}\ (\mathit{first\_judgement}::\mathit{rest\_judgements}) \iff\\&\ \mathsf{initial\_judgement}\ (\mathsf{snd}\ (\mathsf{snd}\ \mathit{params}))\ \mathit{first\_judgement} \wedge\\&\ \mathsf{Valid\_intermediate\_judgements}\ \mathit{params}\ (\mathit{first\_judgement}::\mathit{rest\_judgements})\end{aligned}$$

For checking a formal certificate we therefore first verify that certificate starts at a permissible initial stage. We then iteratively check that transitions have happened correctly, and that the terminating state is a final one where winners are declared. The above specification of a valid vertificate, corresponds to the following computational formal certificate checker.

$$\begin{aligned}&\mathsf{Check\_Parsed\_Certificate}\ \mathit{params}\ [\,] \iff \mathsf{false}\\&\mathsf{Check\_Parsed\_Certificate}\ \mathit{params}\ (\mathit{first\_judgement}::\mathit{rest\_judgements}) \iff\\&\ \mathsf{Initial\_Judgement\_dec}\ (\mathsf{snd}\ (\mathsf{snd}\ \mathit{params}))\ \mathit{first\_judgement} \wedge\\&\ \mathsf{valid\_judgements\_dec}\ \mathit{params}\ (\mathit{first\_judgement}::\mathit{rest\_judgements})\end{aligned}$$

The correctness of this definition rests on the equivalences we have already established between the specifications and their computational counterparts, namely, Initial_Judgement_dec and initial_judgement, and valid_judgements_dec and valid_judgements. Consequently a formal certificate is validated if and only if it is valid according to the HOL specification of Valid_Certificate.

$$\mathsf{Check\_Parsed\_Certificate}\ \mathit{params}\ J \iff \mathsf{Valid\_Certificate}\ \mathit{params}\ J$$

Since the HOL specification realises the protocol, a formal certificate is validated if and only if it meets the protocol's expectation.

3 Translation into CakeML and Code Extraction

The verified certificate-checking function, Check_Parsed_Certificate, described above, is a good starting point for a verifier, but still has two shortcomings: it is a function in logic rather than an executable program, and as a consequence, its inputs must be provided as elements of the respective data types, whereas certificates are purely textual. We now demonstrate how to address these shortcomings and obtain a verified executable for checking certificates. Our final theorem about the verifier executable is presented at the end of this section.

Parsing. The input to the verifier is a textual certificate file, in a format similar to Fig. 1. We specify this file format indirectly, by defining an executable

specification of a certificate parser.

```
Check_Certificate lines ⟺
 case lines of
  quota_line::seats_line::candidates_line::winners_line::jlines ⇒
      case
        (parse_quota quota_line,parse_seats seats_line,
         parse_candidates candidates_line,parse_candidates winners_line,
         mapm parse_judgement jlines)
       of
       (Some quota,Some seats,Some candidates,Some winners,Some judgements) ⇒
           Check_Parsed_Certificate (quota,seats,candidates)
            (rev (Final winners::judgements))
      | _ ⇒ false
 | _ ⇒ false
```

Specifically, we define functions that take a string representing a line in the file and return either None or Some x, where x is the parsed information from the line. Given these parsing functions—parse_quota, parse_seats, etc.—we write the verifier as a function, above, that parses lines from the file then calls Check_Parsed_Certificate to do the verification.

Translation into CakeML and I/O Wrapper. Using prior work on proof-producing synthesis [18] we can automatically synthesise an implementation of the function Check_Certificate in the programming language CakeML. The synthesis tool for CakeML produces a theorem relating the semantics of the synthesised program back to the logical function. However, the result is a *pure* function that expects the lines of a file as input. To actually open the file and read lines from it, we write the impure wrapper check_count (making use of the CakeML Basis Library) around the pure function, and verify the wrapper using Characteristic Formulae for CakeML, as described by Guéneau et al. [13]. The result is a complete CakeML program whose I/O semantics is verified, witnessed by the theorem check_count_compiled below, to implement Check_Certificate on lines from standard input.

To elaborate further on the above step, the impure wrapper check_count calls two impure functions parse_line and loop. The former, calls I/O functions to read one line at a time from the concrete certificate given as lines on the standard input and parse it. It comprises two phases; one for the header of the certificate file consisting of the quota, seats number, and initial list of candidates, and the other is for parsing judgement lines. If the parsing fails due to malformedness of a line, the parser messages the appropriate error on the standard output with the line number included. However, if it succeeds, the parsed line is fed to the loop function to check if the transition from two consecutive parsed judgement lines is a valid step. The parsing and checking of judgement lines continues until either all steps are verified as correct, or an incorrect step is encountered. The following theorem asserts that the loop function returns the correct output None if and only if the initial line of judgements in the certificate file is indeed valid

and all steps taken to move from one judgement line to its successor are correct.

```
loop params i (Final w) j0 js = None ⟺
  EVERY (IS_SOME ∘ parse_judgement) js ∧
  Check_Parsed_Certificate params (rev (Final w::j0::map (the ∘ parse_judgement) js))
```

Compilation in Logic. Finally, we would like an executable verifier in machine code (rather than CakeML code). To produce this, we use the verifed CakeML compiler [22], which can be executed within the theorem prover itself. This is a time-consuming process: compilation within logic can be a thousand times slower (e.g., half an hour) than running the compiler outside the logic (a second or two). But the payoff is a final theorem which only mentions the final generated machine-code implementation: all dependence on the CakeML language and implementation is discharged by proof.

Final Theorem. The final theorem, which we explain further below, is about the generated machine code, represented by the constant check_count_compiled.

```
⊢ wfCL cl ∧ wfFS fs ∧
  x64_installed check_count_compiled (basis_ffi cl fs) mc ms ⇒
   ∃ io_events fs′.
      machine_sem mc (basis_ffi cl fs) ms ⊆
      extend_with_resource_limit { Terminate Success io_events } ∧
      extract_fs fs io_events = Some fs′ ∧
      (stdout fs′ "Certificate OK\n" ⟺
        Check_Certificate (lines_of (get_stdin fs)))
```

We assume (x64_installed) that this code is loaded into memory in an x86-64 machine represented by *mc* and *ms*, and that the command line (*cl*) and file system (*fs*) are well-formed. The conclusion of the theorem concerns the semantics (machine_sem) of executing the machine: it will terminate successfully (or fail if there is not enough memory) with a trace of I/O events (*io_events*) such that if we replay those events on the initial file system, we obtain a resulting file system *fs′* for which the string "Certificate OK\n" is printed on standard output if and only if Check_Certificate succeeds on the lines of standard input.

4 Experimental Results

We have tested our approach against some of the past Australian Legislative Assembly elections in the Australian Capital Territory for years 2008 and 2012 (Fig. 2).[2] The certificates were produced by the Haskell program extracted from our previous formalisation of the same protocol in Coq [11].

We also evaluated the verifier on certificates obtained through randomly generated ballots. We vary two parameters: the number of ballots and the size of each ballot. Figure 3 shows the results on certificates where the number of candidates is fixed at 20, vacancies are 5, and the length of each ballot is 12. Also we

[2] Tests were conducted on one core of an Intel Core i7-7500U CPU 2.70 GHz × 4 Ubuntu 16.4 LTS.

electoral	ballots	vacancies	candidates	time (sec)	certificate size (MB)	year
Brindabella	63334	5	19	86	54.4	2008
Ginninderra	60049	5	27	118	83.0	2008
Molonglo	88266	7	40	329	211.2	2008
Brindabella	63562	5	20	75	74.5	2012
Ginninderra	66076	5	28	191	90.1	2012
Molonglo	91534	7	27	286	158.7	2012

Fig. 2. ACT Legislative Assembly 2008 and 2012

keep the number of ballots, vacancies, and length of each ballot fixed at 100000, 1, and 10 respectively, in order to see the effect of increase in the length of each ballot (Fig. 4). We have also implemented the protocol in an unverified certifying Haskell program.[3] The unverified program was then tested on ballots of the same ACT Legislative Assembly elections. We have then verified the certificates produced by this program for each of the districts. The result shows that the certificates of the districts for the year 2012 are valid. Also the certificate of Molonglo electorate 2008 is verified as correct. However, the two electorates of Brindabella and Ginninderra 2008, despite declaring the final winners correctly, were *invalid* as an error occurs in an intermediate transition on line 6 in both certificates.

ballots	certificate size	time (sec)
400000	523.6	4224
200000	253.3	938
100000	131.1	461

Fig. 3. Varying number of ballots

ballot length	certificate size	time (sec)
6	60.2	140
12	124.0	298
18	180.5	325

Fig. 4. Varying length of each ballot

Based on the aforementioned error message, we only need to inspect a very small part of the certificate. Upon closer inspection, we uncovered a subtle error in the implementation of the elimination rule. On the other hand, the same program successfully (and correctly) computes election results for other districts, substantiating the subtlety of the error. We argue that precisely because of such delicacies in the STV protocol and hence their implementation, we advocate that vote counting be carried out in a certified way, with a minimal trust base such as demonstrated in this paper.

5 Discussion

Universal verifiability is a security requirement introduced for measuring verifiability of an election result by any member of the public [8]. The literature

[3] Source code can be found in the Github repository given in the second page.

on election protocol design agrees on the textual formulation of the concept, despite the fact that they vary in the technical implementation of the property [8]. Moreover, it is accepted that satisfaction of the property rests on verifying three subproperties, namely cast-as-intended, recorded-as-intended, and count-as-recorded [8], and also demonstration of the eligibility verifiability as an explicitly or implicitly stated prerequisite [16].

Our framework only aims at addressing verification of the count-as-recorded subproperty. We do not attempt to introduce an election protocol for answering expectations of the universal verifiability. Therefore, verification of other two subproperties and the eligibility criterion falls outside the focus of the current work. However, our tool can be perfectly employed by any election protocol which accommodates STV scheme and uses Mixnets [4] for anonymising and decrypting ballots. For example, some protocols require authorities to produce a witness for tallying, and then verify it is a proof of correctness for the announced tallying result [6]. Such systems can adapt certification and the checker for (a) offering an independently checkable witness of tallying, and (b) verifying the certificate in a provably correct way. Finally, the certificates which the checker operates on include the exact ballots published by election authorities after the tallying is complete, and are therefore publicly available. Hence, the certificate would not compromise privacy concerns such as vote buying or voter coercion any more than the existing practice of ballot publication.

The framework employs CakeML to achieve an end-to-end verification of certificates. Therefore we prove that executable checker is verified to behave according to its specification in HOL, which operates in a different environment. To obtain this level of verification, we rely on the verified proof-synthesis tool of CakeML, the mechanism for producing deeply embedded equivalent assertions of HOL functions into CakeML environment, the Characteristic Formulae of CakeML to assert that the pure (deeply embedded functions) behave consistently with the impure I/O calls, and the verified compiler that generates executables that provably respect all of the above proofs.

Furthermore, the separation of the program from proofs offered in the combined CakeML and HOL environment makes our formalisation easier to understand. In particular, we believe that external scrutineers should be able to examine the specification of the framework to understand what it does, rather than having to also get to grips with CakeML proofs and computational components. HOL4's rich rewriting tactics and libraries also allow us to express the protocol and discharge related proofs with a minimum amount of lines of encoding.

We have demonstrated the practical feasibility of our approach by means of case studies. For example, the certificates of the Molonglo district, the biggest Legislative Assembly electorate in Australia, are checked in just five minutes.

Our framework is modular in two different ways. On the one hand, the formalisation realises the counting scheme as a set of standalone logical rules. On the other hand, each of the rules comprises independent assertions. Since every STV election consists of counting, elimination, transfer, electing and declaration of winners, we only need to change some of these rules locally to capture different

variants. For example, the STV version used in the Senate elections of Australia requires transfer of excess votes of an elected candidate before any other rule can apply. This difference can be formalised in our system simply by modifying a single component of the TRANSFER rule. So for establishing verification results, we simply have to discharge a few correspondences in HOL. Furthermore, the steps of translation into CakeML and the process of extracting a verified executable remains mostly unaffected.

6 Related Work

Given that our main concern is with the count-as-recorded property, we provide an overview of existing work from the perspective of their tally verification methods. We also compare with related work that combines theorem proving and certified computation.

The existing certificate-producing implementations of vote counting mainly formalise a voting protocol inside the Coq theorem prover and then prove some desired properties about the formalised specification, and then extract the development into Haskell [11,20,23] or OCaml [21] programs. Since the semantics of the target and source of the extraction method differ, and there is no proof that the translation occurs in a semantic-preserving way, verification of the specification does not provably extend to the extracted program. Moreover, these work are either not accompanied by a checker [11], or their checker is an unverified Haskell/OCaml program [20,21,23]. One therefore has to trust both the extraction mechanism and the compiler used to produce the executable.

In the context of certified computation, Alkassar et al. [1] combine certified computation and theorem proving with methods of code verification to establish a framework for validation of certifying algorithms in the C programming language. With the help of the VCC tool [9], pre- and postconditions are generated that are syntactically generalized in the Isabelle theorem prover and then discharged. The user has to trust the VCC tool, and there is duplication of effort in that one has to generalise the conditions imposed by the VCC and then implement them manually in Isabelle to prove. To ameliorate this disadvantage, Noschinski et al. [19] replace the intermediate step where VCC is invoked by the AutoCorres [12] verifier which provably correctly translates (part of) the C language into Isabelle in a semantics-preserving manner. Nonetheless one has to trust that the machine code behaviour corresponds to its top-level C encoding.

Some election protocols [8,15] do require a witness for the tallying result, which should then be verified for correctness. Other work (e.g., [7]) implements algorithms in programming environments such as Python. However the algorithm, the correctness proof of the algorithm, and the implementation occur in different unverified environments. Finally, Cortier et al. [6] present simple formally stated pre- and post-conditions for elections that allow voting for one candidate. This is done inside the dependently-typed programming language $F^\star$. The $F^\star$ environment is implemented by a compiler that translates into RDCIL, a dialect of .NET bytecode. The verification also depends on the external SMT

solver Z3. The size of these tools' implementations makes for a very large trusted code base.

7 Conclusion

Correct, publicly verifiable, transparent election count is a key constituent of establishing trustworthiness in the final outcome. The tool developed here has clarity in encoding, precision in formulation, and modularity in implementation so that it can be taken as a framework for verifying STV election results down to machine level.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. J. Autom. Reason. **52**(3), 241–273 (2014)
2. Blum, M., Kannan, S.: Designing programs that check their work. In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 14–17 May 1989, Seattle, Washington, USA, pp. 86–97 (1989)
3. Brooks, L., Griffits, A.: NSW council elections: computer 'guesstimate' might have ignored your vote. ABC News, September 2017. http://www.abc.net.au/news/2017-09-14/computer-algorithms-may-sway-local-council-elections/8944186
4. Chaum, D.: Untraceable electronic mail return addresses and digital pseudonyms. In: Gritzalis, D.A. (ed.) Secure Electronic Voting, pp. 211–219. Springer, Boston (2003). https://doi.org/10.1007/978-1-4615-0239-5_14
5. Conway, A., Blom, M., Naish, L., Teague, V.: An analysis of New South Wales electronic vote counting. In: Proceedings of the ACSW 2017, pp. 24:1–24:5 (2017)
6. Cortier, V., Eigner, F., Kremer, S., Maffei, M., Wiedling, C.: Type-based verification of electronic voting protocols. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 303–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_16
7. Cortier, V., Galindo, D., Glondu, S., Izabachène, M.: Election verifiability for Helios under weaker trust assumptions. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8713, pp. 327–344. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_19
8. Cortier, V., Galindo, D., Küsters, R., Müller, J., Truderung, T.: Verifiability notions for e-voting protocols. IACR Cryptology ePrint Archive 2016, 287 (2016)
9. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: contract-based modular verification of concurrent C. In: 31st International Conference on Software Engineering, ICSE 2009, Vancouver, Canada, 16–24 May 2009, Companion Volume, pp. 429–430 (2009)
10. Droop, H.R.: On methods of electing representatives. J. Stat. Soc. Lond. **44**(2), 141–202 (1881). http://www.jstor.org/stable/2339223
11. Ghale, M.K., Goré, R., Pattinson, D.: A formally verified single transferable voting scheme with fractional values. In: Krimmer, R., Volkamer, M., Braun Binder, N., Kersting, N., Pereira, O., Schürmann, C. (eds.) E-Vote-ID 2017. LNCS, vol. 10615, pp. 163–182. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68687-5_10

12. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: automatic verified abstraction of C. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 99–115. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_8
13. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 584–610. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_22
14. Kremer, S., Ryan, M., Smyth, B.: Election verifiability in electronic voting protocols. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 389–404. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15497-3_24
15. Küsters, R., Truderung, T., Vogt, A.: Accountability: definition and relationship to verifiability. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, 4–8 October 2010, pp. 526–535 (2010)
16. Küsters, R., Truderung, T., Vogt, A.: Verifiability, privacy, and coercion-resistance: new insights from a case study. In: 32nd IEEE Symposium on Security and Privacy, S&P 2011, 22–25 May 2011, Berkeley, California, USA, pp. 538–553 (2011)
17. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
18. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. J. Funct. Program. **24**(2–3), 284–315 (2014)
19. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through autocorres and simpl. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 46–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_4
20. Pattinson, D., Schürmann, C.: Vote counting as mathematical proof. In: Pfahringer, B., Renz, J. (eds.) AI 2015. LNCS (LNAI), vol. 9457, pp. 464–475. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26350-2_41
21. Pattinson, D., Tiwari, M.: Schulze voting as evidence carrying computation. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 410–426. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_26
22. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, 18–22 September 2016, pp. 60–73. ACM (2016). http://doi.acm.org/10.1145/2951913.2951924
23. Verity, F., Pattinson, D.: Formally verified invariants of vote counting schemes. In: Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2017, Geelong, Australia, 31 January–3 February 2017, pp. 31:1–31:10 (2017). http://doi.acm.org/10.1145/3014812.3014845