Schulze Voting as Evidence Carrying Computation

Dirk Pattinson[1] and Mukesh Tiwari[1]

The Australian National University

Abstract. The correctness of vote counting in electronic election is one of the main pillars that engenders trust in electronic elections. However, the present state of the art in vote counting leaves much to be desired: while some jurisdictions publish the source code of vote counting code, others treat the code as commercial in confidence. None of the systems in use today applies any formal verification. In this paper, we formally specify the so-called Schulze method, a vote counting scheme that is gaining popularity on the open source community. The cornerstone of our formalisation is a (dependent, inductive) type that represents all correct executions of the vote counting scheme. Every inhabitant of this type not only gives a final result, but also all intermediate steps that lead to this result, and can so be externally verified. As a consequence, we do not even need trust the execution of the (verified) algorithm: the correctness of a particular run of the vote counting code can be verified on the basis of the evidence for correctness that is produced along with determination of election winners.

1 Introduction

The Schulze Method [15] is a vote counting scheme that elects a single winner, based on preferential votes. While no preferential voting scheme can guarantee *all* desirable properties that one would like to impose due to Arrow's theorem [2], the Schulze method offers a good compromise, with a number of important properties already established in Schulze's original paper. A quantitative comparison of voting methods [14] also shows that Schulze voting is better (in a game theoretic sense) than other, more established, systems, and the Schulze Method is rapidly gaining popularity in the open software community. The method itself rests on the relative *margins* between two candidates, i.e. the number of voters that prefer one candidate over another. The margin induces an ordering between candidates, where a candidate c is more preferred than d, if more voters prefer c over d than vice versa. One can construct simple examples (see e.g. [14]) where this order does not have a maximal element (a so-called *Condorcet Winner*). Schulze's observation is that this ordering *can* be made transitive by considering sequences of candidates (called *paths*). Given candidates c and d, a *path* between c and d is a sequence of candidates $p = (c, c_1, \ldots, c_n, d)$ that joins c and d, and the *strength* of a path is the minimal margin between adjacent nodes. This induces the *generalised margin* between candidates c and d as the strength of

the strongest path that joins c and d. A candidate c then wins a Schulze count if the generalised margin between c and any other candidate d is at least as large as the generalised margin between d and c.

This paper presents a formal specification of the Schulze method, together with the proof that winners can always be determined which we extract to obtain a provably correct implementation of the Schulze method. The crucial aspect of our formalisation is that the vote counting protocol itself is represented as a dependent inductive type that represents all (correct) partial executions of the protocol. A complete execution is can then be understood as a state of vote counting where election winners have been determined. Our main theorem then asserts that an inhabitant of this type exists, for all possible sets of incoming ballots. Crucially, every such inhabitant contains enough information to (independently) verify the correctness of the election result, and can be thought of as a *certificate* for the count.

From a computational perspective, we view tallying not merely as a function that delivers a result, but instead as a function that delivers a result, *together* with evidence that allows us to verify correctness. In other words, we augment verified correctness of an algorithm with the means to verify each particular *execution*.

From the perspective of electronic voting, this means that we no longer need to trust the hardware and software that was employed to obtain the election result, as the generated certificate can be verified independently. In the literature on electronic voting, this is known as *verifiability* and has been recognised as one of the cornerstones for building trust in election outcomes [7], and is the only answer to key questions such as the possibility of hardware malfunctions, or indeed running the very software that has been claimed to count votes correctly.

The certificate that is produced by each run of our extracted Schulze vote tallying algorithm consists of two parts. The first part details the individual steps of constructing the margin function, based on the set of all ballots cast. The second part presents evidence for the determination of winners, based on generalised margins. For the construction of the margin function, every ballot is processed in turn, with the margin between each pair of votes updated accordingly. The heart of our work lies in this second part of the certificate. To demonstrate that candidate c is an election winner, we have to demonstrate that the generalised margin between c and every other candidate d is at least as large than the generalised margin between d and c. Given that the generalised margin between two candidates c and d is determined in terms of paths $c, c_1, \ldots, c_n, d$ that join c and d, we need to exhibit

- evidence for the existence of a path p from c to d
- evidence for the fact that *no* path q from d to c is stronger than p

where the strength of a path $p = (c_0, \ldots, c_{n+1})$ is the minimum $\min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq n\}$ of the margins between adjacent nodes. While evidently a path itself is evidence for its existence, the *non-existence* of paths with certain properties is more difficult to establish. Here, we use a coinductive approach. As existence of a path with a given strength between two candidates can be easily phrased

as an inductive definition, the *complement* of this predicate arises as a greatest fixpoint, or equivalently as a coinductively defined predicate (see e.g. [10]). This allows us to witness the non-existence of paths by exhibiting co-closed sets.

Our formalisation takes place inside the Coq proof assistant [5] that we chose mainly because its well-developed extraction mechanism and because it allows us to represent the Schulze voting scheme very concisely as a dependent inductive type. Interestingly, we make no use of Coq's mechanism of defining coinductive types [4]: as we are dealing with decidable predicates (formulated as boolean valued functions) only, it is simpler to directly introduce co-closed sets and establish their respective properties.

We take a propositions-as-types approach to synthesising a programme that computes election winners, together with accompanying evidence. In other words, our main theorem states that that winners (and certificates) can be computed for any set of initial ballots. We then use Coq's program extraction facility [12] to generate Haskell code from which we can then generate an executable. We report on experimental result and conclude with further work and a general reflection on our method.

Related Work. The idea of requiring that computations provide not only results, but also proofs attesting to the correctness of the computation is not new, and has been put forward in [1] for computations in general, and in [16] in the context of electronic voting. The general difficulty here is the precise nature of certificates, as different computations require a different type of evidence, and our conceptual contribution is to harness coinduction, more precisely co-closed sets as evidence for membership in the complement of inductively defined sets. Our approach is orthogonal to Necula's proof carrying code, where every executable (not every execution) is equipped with formal guarantees. Formal specification and verification of vote counting schemes has been done e.g. in [3, 8] but none of the methods produce independently verifiable results. The idea of formalising a voting protocol as a type has been put forward in [13] where a variant of single transferable vote has been analysed. While the Schulze method has been analysed e.g. from the point of manipulation [9], this paper appears to be the first to present a formal specification (and a certificate-producing, verified implementation) of the Schulze method in a theorem prover.

Coq Sources. All Coq sources and the benchmarks used in the preparation of this paper are at `http://users.cecs.anu.edu.au/~dpattinson/Sofware/`.

2 Formal Specification of Schulze Voting

We begin with an informal description of Schulze voting. Schulze voting is *preferential* in the sense that every voter gets to express their preference about candidates in the form of a rank ordered list. Here, we allow voters to be indifferent about candidates but require voters to express preferences over *all* candidates. This requirement can be relaxed and we can consider e.g. unranked candidates as tied for the last position.

Given a set s of ballots, one constructs the margin function $m : C \times C \to \mathbb{Z}$. Given two candidates $c, d \in C$, the *margin* of c over d is the number of voters that prefer c over d, minus the number of voters that prefer d over c. In symbols

$$m(c, d) = \sharp\{b \in s \mid c >_b d\} - \sharp\{b \in s \mid d >_b c\}$$

where $\sharp$ denotes cardinality and $<_b$ is the strict ordering given by the ballot $b \in s$. A (directed) *path* from candidate c to candidate d is a sequence $c_0, \ldots, c_{n+1}$ of candidates with $c_0 = c$ and $c_{n+1} = d$ ($n \geq 0$), and the *strength* of this path is the minimum margin of adjacent nodes, i.e.

$$\mathsf{st}(c_0, \ldots, c_{n+1}) = \min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq n\}.$$

Note that the strength of a path may be negative. The Schulze method stipulates that a candidate $c \in C$ is a *winner* of the election with margin function m if, for all other candidates $d \in C$, there exists a number $k \in Z$ such that

- there is a path p from c to d with strength $\mathsf{st}(p) \geq k$
- all paths q from d to c have strength $\mathsf{st}(q) \leq k$.

Informally speaking, we can say that candidate c *beats* candidate d if there's a path p from c to d which stronger than any path from d to c. Using this terminology, a candidate c is a winner if c cannot be beaten by any (other) candidate.

Remark 1. There are multiple formulations of the Schulze method in the literature. Schulze's original paper [15] only considers paths where adjacent nodes have to be distinct, and [9] only considers simple paths, i.e. paths without repeated nodes. Here, we consider *all* paths. It is easy to see that all three definitions are equivalent, i.e. they produce the same set of winners.

Our (Coq) formalisation takes a finite and non-empty type of candidates as given which we assume has decidable equality. For our purposes, the easiest way of stipulating that a type be finite is to require existence of a list containing all inhabitants of this type.

```
Parameter cand : Type.
Parameter cand_all : list cand.
Hypothesis cand_fin : forall c: cand, In c cand_all.
Hypothesis dec_cand : forall n m : cand, {n = m} + {n <> m}.
Hypothesis cand_inh : cand_all <> nil.
```

For the specification of winners of Schulze elections, we take the margin function as given for the moment (and later construct it from the incoming ballots). In Coq, this is conveniently expressed as a variable:

```
Variable marg : cand -> cand -> Z.
```

We formalise the notion of path and strength of a path by means of a single (but ternary) inductive proposition that asserts the existence of a path of strength $\geq k$ between two candidates, for $k \in \mathbb{Z}$.

```
Inductive Path (k: Z) : cand -> cand -> Prop :=
  | unit c d : marg c d >= k -> Path k c d
  | cons  c d e : marg c d >= k -> Path k d e -> Path k c e.
```

Using these definitions, we obtain the following notion of winning (and dually, losing) a Schulze election:

```
Definition wins_prop (c: cand) := forall d : cand, exists k : Z,
  Path k c d /\ (forall l, Path l d c -> l <= k).

Definition loses_prop (c : cand) := exists k: Z, exists  d: cand,
  Path k d c /\ (forall l, Path l c d -> l < k).
```

We reflect the fact that the above are *propositions* in the name of the definitions, in anticipation of type-level definitions of these notions later. The main reason for having equivalent type-level versions of the above is that purely propositional information is discarded during program extraction, unlike the type-level notions of winning and losing that represent evidence of the correctness of the determination of winners.

That is, our goal is to not only compute winners and losers according to the definition above, but also to provide independently verifiable evidence of the correctness of our computation. The propositional definitions of winning and losing above serve as a reference to calibrate their type level counterparts, and we demonstrate the equivalence between propositional and type-level conditions in the next section.

3 A Scrutiny Sheet for the Schulze Method

How can we *know* that, say, a candidate c in fact wins a Schulze election, and that, say, d is not a winner? One way would be to simply re-run an independent implementation of the method (usually hoping that results would be confirmed). But what happens if results diverge?

One major aspect of this paper is that we can answer this question by not only computing the set of winners, but in fact presenting *evidence* for the fact that a particular candidate does or does not win. In the context of electronic vote counting, this is known as a *scrutiny sheet*: a tabulation of all relevant data that allows us to verify the election outcome. Again drawing on an already computed margin function, to demonstrate that a candidate c wins, we need to exhibit an integer k for all competitors d, together with

- evidence for the existence of a path from c to d with strength $\geq k$
- evidence for the non-existence of a path from d to c that is stronger than k

The first item is straight forward, as a path itself is evidence for the existence of a path, and the notion of path is inductively defined. For the second item, we need to produce evidence of membership in the *complement* of an inductively defined set.

Mathematically, given $k \in Z$ and a margin function $m : C \times C \to Z$, the pairs $(c, d) \in C \times C$ for which there exists a path of strength $\geq k$ that joins both are precisely the elements of the least fixpoint $\mathsf{LFP}(V_k)$ of the monotone operator $V_k : \mathsf{Pow}(C \times C) \to \mathsf{Pow}(C \times C)$, defined by

$$V_k(R) = \{(c, e) \in C \mid m(c, e) \geq k \text{ or } (m(c, d) \geq k \text{ and } (d, e) \in R \text{ for some } d \in C)\}$$

It is easy to see that this operator is indeed monotone, and that the least fixpoint exists, e.g. using Kleene's theorem [17]. To show that there is *no* path between d and c of strength $> k$, we therefore need to establish that $(d, c) \notin \mathsf{LFP}(V_{k+1})$.

By duality between least and greatest fixpoints, we have that

$$(c, d) \in C \times C \setminus \mathsf{LFP}(V_{k+1}) \iff (c, d) \in \mathsf{GFP}(W_{k+1})$$

where for arbitrary k, $W_k : \mathsf{Pow}(C \times C) \to \mathsf{Pow}(C \times C)$ is the operator dual to V_k, i.e.

$$W_k(R) = C \times C \setminus (V_k(C \times C \setminus R))$$

and $\mathsf{GFP}(W_k)$ is the greatest fixpoint of W_k. As a consequence, to demonstrate that there is *no* path of strength $\geq k$ between candidates d and c, we need to demonstrate that $(d, c) \in \mathsf{GFP}(W_{k+1})$. By the Knaster-Tarski fixpoint theorem [18], this greatest fixpoint is the supremum of all W_{k+1}-coclosed sets, that is, sets $R \subseteq C \times C$ for which $R \subseteq W_{k+1}(R)$. That is, to demonstrate that $(d, c) \in \mathsf{GFP}(W_{k+1})$, we need to exhibit a W_{k+1}-coclosed set R with $(d, c) \in R$. If we unfold the definitions, we have

$$W_k(R) = \{(c, e) \in C^2 \mid m(c, e) < k \text{ and } (m(c, d) < k \text{ or } (d, c) \in R \text{ for all } d \in C)\}$$

so that given *any* fixpoint R of W_k and $(c, e) \in W$, we know that (i) the margin between c and e is $< k$ so that there's no path of length 1 between c and e, and (ii) for any choice of midpoint d, either the margin between c and d is $< k$ (so that $c, d, \ldots$ cannot be the start of a path of strength $\geq k$) or we don't have a path between d and e of strength $\geq k$. We use the following terminology:

Definition 1. *Let $R \subseteq C \times C$ be a subset and $k \in \mathbb{Z}$. Then R is W_k-*coclosed*, or simply k-*coclosed*, if $R \subseteq W_k(R)$.*

Mathematically, the operator W_k acts on subsets of $C \times C$ that we think of as predicates. In Coq, we formalise these predicates as boolean valued functions and obtain the following definitions where we isolate the function `marg_lt` (that determines whether the margin between two candidates is less than a given integer) for clarity:

```
Definition marg_lt (k : Z) (p : (cand * cand)) :=
  Zlt_bool (marg (fst p) (snd p)) k.

Definition W (k : Z) (p: cand * cand -> bool) (x: cand * cand) :=
  andb
    (marg_lt k x)
    (forallb (fun m => orb (marg_lt k (fst x, m)) (p (m, snd x))) cand_all).
```

In order to formulate type-level definitions, we need to promote the notion of path from a Coq proposition to a proper type, and formulate the notion of k-coclosed predicate.

```
Definition coclosed (k : Z) (f : (cand * cand) -> bool) :=
  forall x, f x = true -> W k f x = true.

Inductive PathT (k: Z) : cand -> cand -> Type :=
| unitT : forall c d, marg c d >= k -> PathT k c d
| consT : forall c d e, marg c d >= k -> PathT k d e -> PathT k c e.
```

The only difference between type level paths (of type `PathT`) and (propositional) paths defined earlier is the fact that the former are proper types, not propositions, and are therefore not erased during extraction. Given the above, we have the following type-level definitions of winning (and dually, non-winning) for Schulze counting:

```
Definition wins_type c := forall d : cand, existsT (k : Z),
  ((PathT k c d) * (existsT (f : (cand * cand) -> bool),
    f (d, c) = true /\ coclosed (k + 1) f))%type.

Definition loses_type (c : cand) := existsT (k : Z) (d : cand),
  ((PathT k d c) * (existsT (f : (cand * cand) -> bool),
    f (c, d) = true /\ coclosed k f))%type.
```

The main result of this section is that type level and propositional evidence for winning (and dually, not winning) a Schulze election are in fact equivalent.

```
Lemma wins_type_prop : forall c, wins_type c -> wins_prop c.

Lemma wins_prop_type : forall c, wins_prop c -> wins_type c.
```

Note that the different nature of the two propositions doesn't allow us to claim an equivalence between both notions, as Coq defines biimplication only on propositions.

The proof of the first statement is completely straight forward, as the type carries all the information needed to establish the propositional winning condition. For the second statement above, we introduce an intermediate lemma

based on the *iterated margin function* $M_k : C \times C \to \mathbb{Z}$. Intuitively, $M_k(c,d)$ is the strength of the strongest path between c and d of length $\leq k+1$. Formally, $M_0(c,d) = m(c,d)$ and

$$M_{i+1}(c,d) = \max\{M_i(c,d), \max\{\min\{m(c,e), M_i(e,d) \mid e \in C\}\}\}$$

for $i \geq 0$. It is intuitively clear (and we establish this fact formally) that the iterated margin function stabilises at the n-th iteration (where n is the number of candidates), as paths with repeated nodes don't contribute to maximising the strength of a path. This proof loosely follows the evident pen-and-paper proof given for example in [6] that is based on cutting out segments of paths between repeated nodes and so reaches a fixed point.

```
Lemma iterated_marg_fp: forall (c d : cand) (n : nat),
     M n c d <= M (length cand_all) c d.
```

That is, the *generalised margin*, i.e. the strength of the strongest (possibly infinite) path between two candidates is effectively computable.

This allows us to relate the propositional winning conditions to the iterated margin function and showing that a candidate c is winning implies that the generalised margin between this candidate and any other candidate sd is at least as large as the generalised margin between d and c.

```
   Lemma wins_prop_iterated_marg (c : cand) : wins_prop c ->
     forall d, M (length cand_all) d c <= M (length cand_all) c d.
```

This condition on iterated margins can in turn be to establish the type-level winning condition, thus closing the loop to the type level winning condition.

```
 Lemma iterated_marg_wins_type (c : cand) : (forall d,
     M (length cand_all) d c <= M (length cand_all) c d) ->
     wins_type c.
```

The crucial part of establishing the type-level winning conditions in the proof of the lemma above is the construction of a co-closed set. First note that `M (length cand_all)` is precisely the generalised margin function. Writing g for this function, we assume that $g(c,d) \geq g(d,c)$ for all candidates d, and given d, we need to construct a $k+1$-coclosed set S where $k = g(c,d)$. One option is to put $S = \{(x,y) \mid g(x,y) < k+1\}$. As every i-coclosed set is also j-coclosed for $i \leq j$, the set $S' = \{(x,y) \mid g(x,y) < g(d,c)+1\}$ is also $k+1$-coclosed and (in general) of smaller cardinality. We therefore witness the existence of a $k+1$-coclosed set with S' as this leads to certificates that are smaller in size and therefore easier to check.

We note that the difference between the type-level and the propositional definition of winning is in fact more than a mere reformulation. As remarked before, one difference is that purely propositional evidence is erased during program extraction so that using just the propositional definitions, we would obtain a

determination of election winners, but no additional information that substantiates this (and that can be verified independently). The second difference is conceptual: it is easy to verify that a set is indeed coclosed as this just involves a finite (and small) amount of data, whereas the fact that *all* paths between two candidates don't exceed a certain strength is impossible to ascertain, given that there are infinitely many paths.

In summary, determining that a particular candidate wins an election based on the `wins_type` notion of winning, the extracted program will *additionally* deliver, for all other candidates,

- an integer k and a path from the winning candidate to the other candidate
- a co-closed set that witnesses that no path of strength $\geq k$ exists that goes the other way.

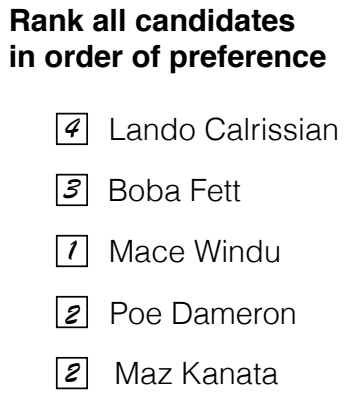
It is precisely this additional data (on top of merely declaring a set of election winners) that allows for scrutiny of the process, as it provides an orthogonal approach to verifying the correctness of the computation: both checking that the given path has a certain strength, and that a set is indeed coclosed, is easy to verify. We reflect more on this in Section 7, and present an example of a full scrutiny sheet in the next section, when we join the type-level winning condition with the construction of the margin function from the given ballots.

4 Schulze Voting as Inductive Type

Up to now, we have described the specification of Schulze voting relative to a given margin function. We now describe the specification (and computation) of the margin function given a profile (set) of ballots. Our formalisation describes an individual *count* as a type with the interpretation that all inhabitants of this type are correct executions of the vote counting algorithm. In the original paper describing the Schulze method [15], a ballot is a linear preorder over the set of candidates.

In practice, ballots are implemented by asking voters to put numerical preferences against the names of candidates as represented by the image on the right. The most natural representation of a ballot is therefore a function $b : C \to \mathbb{N}$ that assigns a natural number (the preference) for each candidate, and we recover a strict linear preorder $<_b$ on candidates by setting $c <_b d$ if $b(c) > b(d)$.

As preferences are usually numbered beginning with 1, we interpret a preference of 0 as the voter failing to designate a preference for a candidate as this allows us to also accommodate incomplete ballots. This is clearly a design decision, and we could have formalised



ballots as functions $b : C \to 1 + \mathbb{N}$ (with 1 being the unit type) but it would add little to our analysis.

```
Definition ballot := cand -> nat.
```

The count of an individual election is then parameterised by the list of ballots cast, and is represented as a dependent inductive type. More precisely, we have a type `State` that represents either an intermediate stages of constructing the margin function or the determination of the final election result:

```
Inductive State: Type :=
| partial: (list ballot * list ballot)  -> (cand -> cand -> Z) -> State
| winners: (cand -> bool) ->  State.
```

The interpretation of this type is that a state either consists of two lists of ballots and a margin function, representing

- the set of ballots counted so far, and the set of invalid ballots seen so far
- the margin function constructed so far

or, to signify that winners have been determined, a boolean function that determines the set of winners.

The type that formalises correct counting of votes according to the Schulze method is parameterised by the profile of ballots cast (that we formalise as a list), and depends on the type `State`. That is to say, an inhabitant of the type `Count n`, for `n` of type `State`, represents a correct execution of the voting protocol up to reaching state `n`. This state generally represents intermediate stages of the construction of the margin function, with the exception of the final step where the election winners are being determined. The inductive type takes the following shape:

```
Inductive Count (bs : list ballot) : State -> Type :=
| ax us m : us = bs -> (forall c d, m c d = 0) ->
  Count bs (partial (us, []) m)         (* zero margin      *)
| cvalid u us m nm inbs : Count bs (partial (u :: us, inbs) m) ->
  (forall c, (u c > 0)%nat) ->          (* u is valid       *)
  (forall c d : cand,
    ((u c < u d) -> nm c d = m c d + 1) (* c preferred to d *) /\
    ((u c = u d) -> nm c d = m c d)     (* c, d rank equal  *) /\
    ((u c > u d) -> nm c d = m c d - 1))(* d preferred to c *) ->
  Count bs (partial (us, inbs) nm)
| cinvalid u us m inbs : Count bs (partial (u :: us, inbs) m) ->
  (exists c, (u c = 0)%nat)             (* u is invalid     *) ->
  Count bs (partial (us, u :: inbs) m)
| fin m inbs w (d : (forall c, (wins_type m c) + (loses_type m c))):
  Count bs (partial ([], inbs) m)       (* no ballots left  *) ->
  (forall c, w c = true <-> (exists x, d c = inl x)) ->
```

```
(forall c, w c = false <-> (exists x, d c = inr x)) ->
Count bs (winners w).
```

The intuition here is simple: the first constructor, `ax`, initiates the construction of the margin function, and we ensure that all ballots are uncounted, no ballot are invalid (yet), and the margin function is constantly zero. The second constructor, `cvalid`, updates the margin function according to a valid ballot (all candidates have preferences marked against their name), and removes the ballot from the list of uncounted ballots. The constructor `cinvalid` moves an invalid ballot to the list of invalid ballots, and the last constructor `fin` applies only if the margin function is completely constructed (no more uncounted ballots). In its arguments, `w: cand -> bool` is the function that determines election winners, and `d` is a function that delivers, for every candidate, type-level evidence of winning or losing, consistent with `w`. Given this, we can conclude the count and declare `w` to be the set of winners (or more precisely, those candidates for which `w` evaluates to `true`).

Together with the equivalence of the propositional notions of winning or losing a Schulze count with their type-level counterparts, every inhabitant of the type `Count b (winners w)` then represents a correct count of ballots `b` leading to the boolean predicate `w: Cand -> bool` that determines the winners of the election with initial set `b` of ballots.

The crucial aspect of our formalisation of executions of Schulze counting is that the transcript of the count is represented by a type that is *not* a proposition. As a consequence, extraction delivers a program that produces the (set of) election winner(s), *together* with the evidence recorded in the type to enable independent verification.

5 All Schulze Election Have Winners

The main theorem, the proof of which we describe in this section, is that all elections according to the Schulze method engender a boolean-valued function `w: Cand -> bool` that determines precisely which candidates are winners of the election, together with type-level evidence of this. Note that a Schulze election can have more than one winner, the simplest (but not the only) example being when no ballots at all have been cast. The theorem that we establish (and later extract as a program) simply states that for every incoming set of ballots, there is a boolean function that determines the election winners, together with an inhabitant of the type `Count` that witnesses the correctness of the execution of the count. In Coq, we use a type-level existential quantifier `existsT` where `existsT (x:A), P` stands for $\Sigma_{\mathtt{x:A}}\mathtt{P}$.

```
Theorem schulze_winners: forall (bs : list ballot),
  existsT (f : cand -> bool) (p : Count bs (winners f)), True.
```

The first step in the proof is elementary: We show that for any given list of ballots we can reach a state of the count where there are no more uncounted ballots, i.e. the margin function has been fully constructed.

The second step relies on the iterated margin function already discussed in Section 3. As $M_n(c,d)$ (for n being the number of candidates) is the strength of the strongest path between c and d, we construct a boolean function w such that $w(c) = \texttt{true}$ if and only if $M_n(c,d) \geq M_n(d,c)$ for all $d \in C$. We then construct the type-level evidence required in the constructor `fin` using the function (or proposition) `iterated_marg_wins_type` described earlier.

Coq's extraction mechanism then allows us to turn this into a provably correct program. When extracting, all purely propositional information is erased and given a set of incoming ballots, the ensuing program produces an inhabitant of the (extracted) type `Count` that records the construction of the margin function, together with (type level) evidence of correctness of the determination of winners. That is, we see the individual steps of the construction of the margin function (one step per ballot) and once all ballots are exhausted, the determination of winners, together with paths and co-closed sets. The following is the transcript of a Schulze election where we have added wrappers to pretty-print the information content. This is the (full) scrutiny sheet promised in Section 3.

```
V: [A3 B1 C2 D4,..], I: [], M: [AB:0 AC:0 AD:0 BC:0 BD:0 CD:0]
-------------------------------------------------------------
V: [A1 B0 C4 D3,..], I: [], M: [AB:-1 AC:-1 AD:1 BC:1 BD:1 CD:1]
-----------------------------------------------------------------
V: [A3 B1 C2 D4,..], I: [A1 B0 C4 D3], M: [AB:-1 AC:-1 AD:1 BC:1 BD:1 CD:1]
-------------------------------------------------------------------------
                      . . .
-------------------------------------------------------------------------
V: [A1 B3 C2 D4], I: [A1 B0 C4 D3], M: [AB:2 AC:2 AD:8 BC:5 BD:8 CD:8]
-------------------------------------------------------------------------
V: [], I: [A1 B0 C4 D3], M: [AB:3 AC:3 AD:9 BC:4 BD:9 CD:9]
-----------------------------------------------------------
winning: A
  for B: path A --> B of strenght 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for C: path A --> C of strenght 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for D: path A --> D of strenght 9, 10-coclosed set:
    [(D,A),(D,B),(D,C)]
losing: B
  exists A: path A --> B of strength 3, 3-coclosed set:
    [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C),(D,A),(D,B),(D,C),(D,D)]
losing: C
  exists A: path A --> C of strength 3, 3-coclosed set:
    [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C),(D,A),(D,B),(D,C),(D,D)]
losing: D
  exists A: path A --> D of strength 9, 9-coclosed set:
    [(A,A),(A,B),(A,C),(B,A),(B,B),(B,C),(C,A),(C,B),(C,C),(D,A),(D,B),
     (D,C),(D,D)]
```

Here, we assume four candidates, `A`, `B`, `C` and `D` and a ballot of the form `A3 B2 C4 D1` signifies that `D` is the most preferred candidate (the first preference), followed by `B` (second preference), `A` and `C`. In every line, we only display the first

uncounted ballot (condensing the remainder of the ballots to an ellipsis), followed by votes that we have deemed to be invalid. We display the partially constructed margin function on the right. Note that the margin function satisfies $m(x, y) = -m(y, x)$ and $m(x, x) = 0$ so that the margins displayed allow us to reconstruct the entire margin function. In the construction of the margin function, we begin with the constant zero function, and going from one line to the next, the new margin function arises by updating according to the first ballot. This corresponds to the constructor `cvalid` and `cinvalid` being applied recursively: we see an invalid ballot being set aside in the step from the second to the third line, all other ballots are valid. Once the margin function is fully constructed (there are no more uncounted ballots), we display the evidence provided in the constructor `fin`: we present evidence of winning (losing) for all winning (losing) candidates. In order to actually verify the computed result, a third party observer would have to

1. Check the correctness of the individual steps of computing the margin function
2. For winners, verify that the claimed paths exist with the claimed strength, and check that the claimed sets are indeed coclosed.

Contrary to re-running a different implementation on the same ballots, our scrutiny sheet provides an *orthogonal* perspective on the data and how it was used to determine the election result.

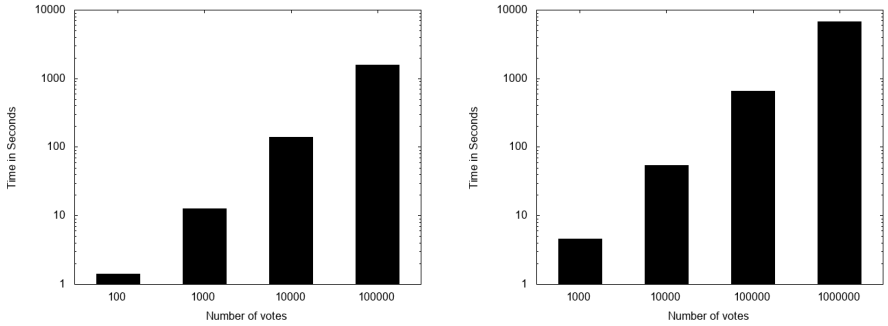
6 Experimental Results

Coq's built in extraction mechanism extracts into both Haskell and Ocaml, and allows to extract Coq types into built in (or user defined) types in the target programming language.

We have evaluated our approach by extracting the entire Coq development into Haskell, with all types defined by Coq extracted as is, i.e. in particular using Coq's unary representation of natural numbers. The results are displayed in Figure 1a using a logarithmic scale.
Profiling the executable reveals that a large portion of time is being spent comparing natural numbers (that Coq represents in unary) for size. In Figure 1b, we have extracted Coq's natural number type to the (native) Haskell type `Int` of integers, and the comparison function to the Haskell native comparison operator (`<=`). The use of native integers has resulted in a nearly tenfold speedup as seen in the figure on the right.

While extraction of Coq data types into their Haskell counterparts potentially jeopardises correctness of the code, the fact that we produce a transcript of the code (a scrutiny sheet) that can (and should!) be checked for correctness externally alleviates the risk of erroneous results that can be produced that way.

Both graphs have been produced assuming four candidates and (the same) randomly generated ballots on an Intel i7 2.6 GHz Linux desktop computer with 8GB of ram. We have not analysed the memory consumption for either benchmark as it appeared to be minimal.



(a) Direct Extraction (b) Extraction using Haskell Integers

Fig. 1: Experimental Results

7 Discussion

Our paper takes the approach that computation of winners in electronic voting (and in situations where correctness is key in general) should not only produce an end result, but an end result, *together* with a verifiable justification of the correctness of the computed result. In this paper, we have exemplified this approach by providing a provably correct, and evidence-producing implementation of vote counting according to the Schulze method.

While the Schulze method is not difficult to implement, and indeed there are many freely available implementations, comparing the results between different implementations can give some level of assurance for correctness only in case the results agree. If there is a discrepancy, a certificate for the correctness of the count allows to adjudicate between different implementations, as the certificate can be checked with relatively little computational effort.

From the perspective of computational complexity, checking a transcript for correctness is of the same complexity as computing the set of winners, as our certificates are cubic in size, so that certificate checking is not less complex than the actual computation.

However, publishing an independently verifiable certificate that attests the individual steps of the computation helps to increase *trust* in the computed election outcome. Typically, the use of technology in elections increases the amount of trust that we need to place both in technological artefacts, and in people. It raises questions that range from fundamental aspects, such as proper testing and/or verification of the software, to very practical questions, e.g. whether the correct version of the software has been run. On the contrast, publishing a certificate of the count dramatically reduces the amount of trust that we need to place into both people and technology: the ability to publish a verifiable justification of the correctness of the count allows a large number of individuals to scrutinise the count. While only moderate programming skills are required to check the validity of a certificate (the transcript of the count), even individuals

without any programming background can at least spot-check the transcript: for the construction of the margin function, everything that is needed is to show that the respective margins change according to the counted ballot. For the correctness of determination of winners, it is easy to verify existence of paths of a given strength, and also whether certain sets are co-closed – even by hand! This dramatically increases the class of people that can scrutinise the correctness of the count, and so helps to establish a trust basis that is much wider as no trust in election officials and software artefacts is required.

Technically, we do not *implement* an algorithm that counts votes according to the Schulze method. Instead, we give a specification of the Schulze winning conditions (`wins_prop` in Section 2) in terms of an already computed margin function that (we hope) can immediately be seen to be correct, and then show that those winning conditions are equivalent to the existence of inhabitants of types that carry verifiable evidence (`wins_type`). We then join the (type level) winning conditions with an inductive type that details the construction of the margin function in an inductive type. Via propositions-as-types, a provably correct vote counting function is then equivalent the proposition that there exists an inhabitant of `Count` for every set of ballots. Coq's extraction mechanism then allows us to extract a Haskell program that produces election winners, together with verifiable certificates.

8 Conclusion and Further Work

This paper has presented a formalisation of the Schulze method for counting preferential ballots. Our formalisation focuses on the correct execution of the method. One appealing aspect of the Schulze method is that it meets lots of desirable criteria of vote counting systems. We leave the verification of these for future work.

In our formalisation of vote counting, there is a one-to-one correspondence between correct executions of the protocol, and inhabitants of a (dependent) inductive type. In our Coq development, we have used the propositions-as-types approach, and have constructed an existence proof, from which we have generated Haskell code. An alternative approach would be to implement a function that directly constructs inhabitants, and obtain a detailed performance comparison between both approaches. While we anticipate that a direct implementation brings performance benefits, our experimental evaluation shows that even with very little optimisation (Section 6), extracting vote counting program from an existence proof allows us to count a relatively large number of ballots already.

Finally, we remark that extracting Coq developments into a programming language itself is a non-verified process which could still introduce errors in our code. The most promising way to alleviate this is to independently implement (and verify) a certificate verifier, possibly in a language such as CakeML [11] that is guaranteed to be correct to the machine level.

References

1. K. Arkoudas and M. C. Rinard. Deductive runtime certification. *Electr. Notes Theor. Comput. Sci.*, 113:45–63, 2005.
2. K. J. Arrow. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58(4):328–346, 1950.
3. B. Beckert, R. Goré, C. Schürmann, T. Bormer, and J. Wang. Verifying voting schemes. *J. Inf. Sec. Appl.*, 19(2):115–129, 2014.
4. Y. Bertot. Coinduction in coq. *CoRR*, abs/cs/0603119, 2006.
5. 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.
6. B. A. Carr'e. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7(3):273, 1971.
7. D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2(1):38–47, 2004.
8. D. Cochran and J. Kiniry. Votail: A formally specified and verified ballot counting system for irish PR-STV elections. In *Pre-proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, 2010.
9. L. A. Hemaspaandra, R. Lavaee, and C. Menton. Schulze and ranked-pairs voting are fixed-parameter tractable to bribe, manipulate, and control. *Ann. Math. Artif. Intell.*, 77(3-4):191–223, 2016.
10. D. Kozen and A. Silva. Practical coinduction. *Mathematical Structures in Computer Science*, pages 1–21, February 2016.
11. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *Proc. POPL 2014*, pages 179–192. ACM, 2014.
12. 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.
13. 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.
14. R. L. Rivest and E. Shen. An optimal single-winner preferential voting system based on game theory. 2010.
15. M. Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.
16. C. Schürmann. Electronic elections: Trust through engineering. In *Proc. RE-VOTE 2009*, pages 38–46. IEEE Computer Society, 2009.
17. V. Stoltenberg-Hansen, I. Lindström, and E. Griffor. *Mathematical Theory of Domains*. Number 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.
18. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.