Electronic vote counting as mathematical proof

Florence Verity

May 2016

A thesis submitted for the degree of Bachelor of Philosophy (Honours)
of the Australian National University

Australian
National
University

Declaration

The work in this thesis is my own except where otherwise stated.

Florence Verity

Acknowledgements

I wish to acknowledge the help of my primary supervisor, Dirk Pattinson. Thank you for sharing such an enjoyable and interesting project with me, and for your generous support. I wish to thank my supervisor from the Mathematical Sciences Institute, Scott Morrison, for his interest in the project and helpful feedback. I also acknowledge Katya Lebedeva for sharing her voting system expertise when it came to formalising a real-world protocol. Finally, I thank my family and friends for their curiosity and support throughout.

Abstract

Electronic vote counting is replacing manual counting, and with this comes the problem of verification to provide trust in the outcome of an electronic count. In this thesis, I explain and build on the 'vote counting as mathematical proof' approach to the problem by first presenting the underpinning theory of proofs and types. I then prove that the formalisation of a particular vote counting protocol under this approach satisfies the majority criterion. I also develop a generic approach to formalising vote counting protocols as natural deduction systems, called 'generic termination', and demonstrate that it can be readily applied to simple and real-world vote counting protocols.

Contents

Acknowledgements **v**

Abstract **vii**

Notation and terminology **xi**

Introduction **1**

1 Context **3**

1.1 Electronic vote counting . . . 3
1.1.1 Formal verification . . . 4
1.1.2 Electronic vote counting in Australia . . . 6
1.2 Vote counting as mathematical proof . . . 6
1.2.1 Vote counting rules as proof rules . . . 7
1.2.2 Implementation . . . 10
1.3 Related work . . . 11

2 Type theory **13**

2.1 Types . . . 13
2.1.1 Types versus sets . . . 14
2.2 Constructive logic . . . 15
2.2.1 BHK-interpretation . . . 16
2.3 Curry-Howard isomorphism . . . 21
2.3.1 Function type . . . 22
2.3.2 Product type . . . 22
2.3.3 Sum type . . . 23
2.4 Extending to first-order logic . . . 23
2.4.1 Universal quantifier . . . 24
2.4.2 Existential quantifier . . . 24

2.4.3 Dependent types . . . 25
2.5 More features of type theory . . . 25
2.5.1 Universes . . . 25
2.5.2 Inductive types . . . 26
2.5.3 Dependent inductive types . . . 27
2.6 The *Coq* proof assistant . . . 27
2.6.1 Goals and tactics . . . 27
2.6.2 Program extraction . . . 28

3 Majority Criterion 29
3.1 Simple STV . . . 30
3.1.1 Mathematical formalisation . . . 30
3.1.2 Formalisation in *Coq* . . . 35
3.2 Proof of the majority criterion . . . 40
3.2.1 Mathematical formalisation . . . 40
3.2.2 Formalisation in *Coq* . . . 45

4 Generic Termination 49
4.1 Method . . . 50
4.1.1 Formalisation in *Coq* . . . 52
4.2 First past the post . . . 56
4.3 Simple STV . . . 59
4.4 The *ANU Union* vote counting protocol . . . 62
4.4.1 Mathematical formalisation . . . 63
4.4.2 Formalisation in *Coq* . . . 68

Concluding remarks 71

A The Union constitution, sections 22-23 73

Bibliography 77

Notation and terminology

The notation used is primarily standard notation from mathematical logic and is explained where necessary. List notation is used that is mostly standard except for some abuse of set notation, and so is given here.

Notation

$\mathsf{List}(\mathrm{X})$	the set of lists with elements in X.
$[a_0, a_1, \ldots, a_n]$	the list with elements $a_0, a_1, \ldots, a_n$.
$f : fs$	the list with first element f and remainder the list fs.
$\lvert l \rvert$	the length of list l.
$l \mathbin{;} m$	the concatenation of two lists l and m.
$c \in l$	c occurs in the list l.
$\bigcup l$	the set of elements of the list l.

Terminology

first-order logic	an extension of propositional logic to the universal quantifier $\forall$ and the existential quantifier $\exists$.
formal language	a set of strings of symbols, often described by syntactical rules.
formula	a string of symbols that is part of a formal language.
judgement	an assertion in a formal system.

preferential voting	a voting system in which voters rank candidates in order of preference, rather than voting for a single candidate.
propositional logic	logic concerned with propositions formed from atomic propositions and logical connectives.
vote counting protocol	a procedure for counting votes in an election, which may be expressed in an informal or a formal language.

Introduction

The aim of this thesis is to build on work by Pattinson in [16], which established an approach to electronic vote counting called 'vote counting as mathematical proof'. This relies on the observation that vote counting rules may be given the same status as deduction rules in proof theory. Such a rules-based formalisation may be implemented inside the interactive theorem prover *Coq*, allowing the extraction of a provably correct vote counting program. Along with the outcome of the count, this program produces a proof of the result in the form of a sequence of rule applications, which may then be independently verified for correctness.

In this thesis, I prove that the formalisation of a single transferable vote (STV) counting protocol in [16] satisfies the majority criterion, a mathematical property used to compare voting systems. I address the problem of abstracting the properties common to all vote counting protocols under the proof rules approach by building a generic framework. I prove that in this framework, there are properties local to the list of rules that if satisfied, ensure a provable outcome of the count. This makes it easier to implement different vote counting protocols under the mathematical proof approach, which I demonstrate by applying the framework to the first past the past (FPTP) and STV protocols in [16], as well as a more complex real-world protocol.

Chapter 1 provides the context to the problem of trust in electronic vote counting and describes the status of electronic vote counting in Australia. It introduces the approach taken in this thesis by illustrating the analogy to proof theory by formalising FPTP as deduction rules. Finally, it compares vote counting as mathematical proof to related work. Chapter 2 presents the theory of types that underpins the approach. This focuses on replacing truth conditions in natural deduction by proof conditions to get a constructive logic, which we then compare to type theory by the Curry-Howard isomorphism.

Chapter 3 describes the specification and *Coq* implementation of STV as proof rules, before proving that it satisfies the majority criterion. Finally, Chapter 4 develops the generic framework, called 'generic termination'. It includes the implementation of FPTP and STV inside this framework, as well as the specification and implementation of the ANU Union vote counting protocol. Appendix A gives the relevant part of the ANU Union constitution specifying the vote counting procedure that is formalised in Chapter 4.

Accompanying this thesis is *Coq* code of the implementations. Throughout the thesis, the definitions made and theorems proved in the code are explained, both mathematically and with snippets of code. While the nature of an interactive theorem prover means that provided the theorems have been given correctly, the proofs are correct, it is recommended to open the code and step through some of it to gain a sense of the formal proof environment. Instructions on how to do this, as well as the code, are included on an accompanying USB drive. They may also be downloaded at `https://github.com/floverity/coq-vote-counting`.

Chapter 1

Context

This section motivates the shift from traditional elections to electronic alternatives and discusses how trust can be established in the outcome of an electronic election. It includes an introduction to the approach adopted in this thesis of 'vote counting as mathematical proof'. Finally, it provides a brief comparison to related work.

1.1 Electronic vote counting

Belief in the legitimacy of an election outcome requires trust that the votes have been counted correctly, that is, according to the legislated vote counting protocol. In the case of traditional, paper-based elections, this trust is founded in the role of election scrutineers who observe the vote count as it is conducted. This is not a mechanism for picking up the small errors to which complex manual tasks are prone; rather, it is a widely-accepted means of keeping the process transparent, ridding the need for blind faith in the count being run according to protocol.

Paper-based elections are progressively being replaced by electronic alternatives at every stage, from vote-casting to vote counting. The traditional process – printing and transporting the ballots to polling stations, conducting the polling, collecting the ballots and transporting them to a central location to perform the manual count – is labour-intensive, expensive and time-consuming. As it stands, the vote count for the Australian Senate remains unfinished until several weeks after the election is held.[2] Furthermore, traditional elections deny the privacy of voters who require assistance to complete a paper ballot, such as someone with a vision impairment. Electronic alternatives can provide this assistance anonymously. Electronic elections also have the potential to increase trust in the

voting process, for example, by keeping a record of the votes cast. The potential for paper ballots to go missing was evidenced in the 2013 Western Australian Senate election, in which 1139 votes were misplaced and an expensive recount was required.[1]

Electronic elections can also accommodate more complex vote counting protocols. The design of a voting system takes into account the ease with which ballots may be counted manually, rather than just considered the fairest way to conduct the tally.[12] In the case of preferential voting, this usually means that simplifications in the process of transferring ballots from one candidate to another are made. Should a tie between candidates occur, it is necessary to trace the previous rounds of transfers to break the tie. When done by hand, this process is an approximation that can lead to unfairness. Electronic elections allow for more complex yet fairer voting systems to be used in real world.

1.1.1 Formal verification

To realise these benefits, electronic elections need to be conducted in a trustworthy manner. The traditional scrutiny measures, such as allowing members of the public to observe the count, must be replaced by alternatives that perform the same role for voting software. The need for new scrutiny measures applies to both vote-casting and vote counting technologies; this thesis is concerned only with the latter.

This is not the first time assurance that a program performs a desired task correctly has been required. The field of *formal verification* in computer science is concerned with techniques for proving the *correctness* of a program. Here, 'correct' is only understood with respect to a *formal specification* that details the required properties of a program, and is *formal* as it is given in a language defined by mathematical logic (or mathematical logic itself).

Figure 1.1 illustrates the process from a legal document specifying a vote counting procedure to the actual vote count being conducted. Each arrow represents a step away from the original text, with the left-hand side corresponding to a traditional, manual count and the right-hand side corresponding to an electronic count.

To have trust in the outcome of the count, each arrow should be justified with a process verifying that the original legal text is still correctly captured. In the case of a manual count, the legal text is distilled into a practical, manual procedure for the vote counter to follow, referred to as the *manual specification*.

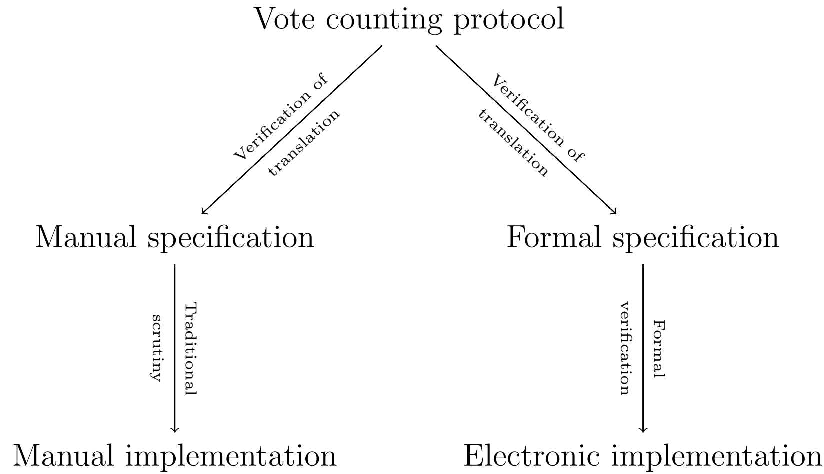


Figure 1.1: Verification steps for a traditional election and an electronic election.

Verification of this step lies in trusting the electoral officials who are familiar with both texts. The *manual implementation* then refers to the actual counting of the votes, where trust comes from the aforementioned role of election scrutineer.

In the case of an electronic vote count, translation into a formal specification is verified by convincing oneself it expresses the same process as the original text, as in the case of a manual count. Trust comes with many people being confident that they are one and the same, and so relies on using a language that is well-known or high-level, that is, close to natural language. The specification can also be validated by proving certain theorems or 'sanity checks' that are intended to be true of the specification. A method of formal verification is then used for trust in the implementation, that is, the actual piece of vote counting software. So while 'formal verification' has ther technical meaning of proving the correctness of a program with respect to a formal specification, in the case of vote counting, we are interested in a two-stage verification process of specification and implementation. Proving the correctness of a program does not verify that the formal specification correctly describes the problem.

A side-issue to updating the traditional scrutiny measures to the electronic case is communicating how these measures work to convince the public that they are adequate. This requires explaining how formal verification works and why it should be trusted to replace traditional scrutiny. It is also advantageous that the method of formal verification requires as few technical skills as possible, so that the pool of potential electronic scrutineers is as wide as possible.

It is also worth noting that there are alternative scrutiny measures to formal verification, however these are generally inferior as they don't rely on mathemat-

ical proof. One option is to publish the votes so that voters may rerun the count with their own software. However, this is open to a particular coercion tactic, sometimes called the “Italian attack”, whereby a voter is instructed to vote in a particular way and the coercer may then search to see whether this vote has been recorded. In a preferential voting system, the permutations of possible ballots can be large, meaning there is no guarantee that the particular vote will be otherwise cast.

1.1.2 Electronic vote counting in Australia

Electronic vote counting programs are used in Australia. Australian elections are conducted according to versions of single transferable vote (STV) counting. In this system, voters rank the candidates on the ballot in order of preference. The count proceeds by electing candidates who have reached a *quota* of votes required to be elected, and eliminating candidates when seats still remain and no candidate can be elected. When a candidate is eliminated, the ballots counting towards them are then ‘transferred’ to the next candidate given preference on the ballot.

The vote counting programs used in Australia have not been formally verified. In Victoria and the ACT, digitised ballots are counted using *open-source* programs, meaning everyone has access to scrutinise the code, however it has not been proved mathematically that the code correctly adheres to the legislated protocol. This is problematic because programming errors are commonly made and difficult to detect. This was evidenced by the three bugs found in the counting module of the ACT’s Electronic Voting and Counting System (EVACS) by the ANU Logic and Computation Group. Had these bugs manifested in the five elections using EVACS, they could have changed the election outcome.[14, p.7] The electoral body of NSW and the Australian Electoral Commission both use unverified, closed-source code.[13, p.9]

1.2 Vote counting as mathematical proof

The approach adopted in this thesis, variously referred to as “vote counting as mathematical proof” and “vote counting rules as proof rules”, comes from [16] and is based on the observation that vote counting procedures are comprised of rules analogous to proof rules in mathematical logic. Treating vote counting rules as proof rules then enables the appropriation of concepts from proof theory to

build an approach to electronic vote counting that is naturally mathematical, and thus only a small step away from formal verification.

This inherent mathematical content comes from proof theory, a branch of mathematical logic concerned with proofs as formal mathematical objects, as opposed to the more informal notion of what constitutes a proof in standard mathematical practice. In particular, we illustrate the analogy using rules from *natural deduction*, one of the two main systems of proof in contemporary structural proof theory, alongside the sequent calculus. A familiarity with the system of natural deduction is assumed for what follows, however the relevant features are explained and the reader is referred to [17] for more details.

Natural deduction is an example of a class of formal systems called *deductive systems*. A deductive system consists of *judgements* and *rules*. It specifies the forms a valid judgement may take and provides a collection of rules consisting of axioms and inference rules, where a rule is given by zero or more premises and a conclusion. In comparison to an algebraic theory such as group theory, we can think of the judgements as the elements of a group and the rules as the group action.

1.2.1 Vote counting rules as proof rules

We delineate three important similarities between proof rules from natural deduction and vote counting rules. To illustrate these similarities, we use the simplest system of vote counting called 'first past the post' (FPTP). In FPTP, a vote is for a single candidate and the candidate who receives the most votes wins. In what follows, we use the formalisation of FPTP rules from [16].

Form

The following is an example of a rule from natural deduction, namely the introduction of a disjunction:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \ (\vee \mathsf{I})$$

In this rule, $(\vee \mathsf{I})$ is the name of the rule, A and B are formulae, Γ is a set of formulae called the *assumptions*, and $\Gamma \vdash A$ and $\Gamma \vdash A \vee B$ are *judgements*. For now, a judgement $\Gamma \vdash X$ is read as 'Γ entails X'. This is not the only interpretation we can assign to the symbols; in the next section, we will see the significance of considering different judgements. The judgement above the line

is called a premise and the judgement below the line is called the conclusion. This is a single-premise rule, we may also have multi-premise rules and rules with no premises, however all of the vote counting rules defined in this thesis are single-premise rules.

Vote counting rules can be written in the same form. To illustrate this, consider the rule for counting a voting in FPTP:

> "To count a single vote, pick an uncounted ballot, mark it as counted and increase the tally for the candidate on the ballot by one."

To formalise this rule as a proof rule, let C be a set of candidates in the running to be elected, and identify a vote for candidate $c \in C$ with their name c. For a set X, let $\mathsf{List}(X)$ be the set of all lists with elements in X. Then the list of ballots cast can be represented by $b \in \mathsf{List}(C)$ and the list of uncounted ballots can be represented by $u \in \mathsf{List}(C)$. Let $t : C \to \mathbb{N}$ be the running tally, a function that maps a candidate c to the number of votes that have been counted in their favour. Then an intermediate stage of the vote count can be represented by a list of uncounted votes and a tally, namely a pair (u, t). A judgement takes the form

$$b \vdash (u, t)$$

interpreted as "the intermediate state (u, t) is correct state of vote counting from the assumption of the list of ballots cast b". Then we may express the 'count one' rule as:

$$\frac{b \vdash (u_1 \mathbin{;} [c] \mathbin{;} u_2, t)}{b \vdash (u_1 \mathbin{;} u_2, t[c \mapsto t(c) + 1])} \ (\mathsf{C1})$$

The judgement above the line is the premise, corresponding to an intermediate stage of the vote count in which the uncounted votes contain a vote for c, and the judgement below the line is the conclusion, corresponding to an intermediate stage of the vote count in which the vote for c has been removed from the uncounted votes, and the tally has been updated by increasing the value for c by one. The rule of inference is labelled on the right the name $\mathsf{C1}$, read 'count one'.

Axioms

An *axiom* in proof theory is a judgement that can be introduced without any formal justification, as it is accepted to be self-evident. It is a zero-premise rule. An example of an axiom in natural deduction is:

$$\frac{}{\Gamma \cup A \vdash A} \text{(Ax)}$$

This says that we can deduce, from nothing, the judgement "a set of assumptions containing A entails A". The start of a FPTP count is given by the stipulation:

"To start the count, mark all ballots as being uncounted votes."

This can be expressed as the following axiom:

$$\frac{}{b \vdash (b, \mathsf{nty})} \text{(Ax)}$$

where $\mathsf{nty} : C \to \mathbb{N}$ is the null tally, returning zero for every candidate. This says that taking all the ballots cast as the uncounted votes and the null tally is a correct state of vote counting.

Provability

There is a specific notion of *provability* in proof theory.

Definition 1.1. *A judgement is* provable *if it can be obtained as the final judgement in a finite sequence of judgements, where the sequence begins with an axiom and each subsequent judgement is either an axiom or the result of a valid rule application to a prior judgement.*

We refer to the sequence of judgements and rule applications as a *proof sequence*. If multi-premise rules are involved, the proof-sequence is part of a branching *proof tree*, however since single premise rules are sufficient for all of the vote counting rules in this thesis we will only be concerned with proof sequences. For example, we have the follow proof sequence showing that $\Gamma \cup A \vdash A \vee B$ is provable:

$$\frac{\frac{}{\Gamma \cup A \vdash A} \text{(Ax)}}{\Gamma \cup A \vdash A \vee B} (\vee\text{I})$$

Similarly, we can provide a proof sequence to show that a vote counting state is provable:

$$\frac{\frac{}{b1 \fatsemi [c] \fatsemi b2 \vdash (b1 \fatsemi [c] \fatsemi b2, \mathsf{nty})} \text{(Ax)}}{b1 \fatsemi [c] \fatsemi b2 \vdash (b1 \fatsemi b2, \mathsf{nty}[c \mapsto \mathsf{nty}(c) + 1])} \text{(C1)}$$

This says that if the ballots cast contains a vote for candidate c, we may start counting with these ballots and the null tally, and then count the vote for c to obtain a provable intermediate state of the count.

We can also use this notion of provability to define the outcome of a vote count. Consider the final rule in a FPTP count:

> "If no uncounted votes remain, the candidate with the highest tally will be declared the winner."

If we introduce a new judgement of the form $b \vdash \mathsf{winner}(c)$, we can define a rule that declares winners:

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

The premise is an intermediate state of the count in which no uncounted votes remain, and expresses as a side condition the property that the tally of every candidate in the running is no greater than the tally of c. The conclusion is a declaration of c as the winner. In this system, we may then define the outcome of a vote count to be a *provable* declaration of a winner.

Given these parallels, we have modelled a simple vote counting protocol as a deductive system.

1.2.2 Implementation

Giving vote counting rules the same status as proof rules acts as a formal specification for a vote counting protocol. Much more than that, we will see that the mathematical nature of this formalisation has inherent computational content. This allows us to move easily from the logical specification to implementing an actual vote counting program.

The theory behind this far-reaching concept – type theory and the Curry-Howard isomorphism – will be the focus of next chapter. It will allow us to automatically generate a vote counting program that not only gives the outcome of a count, but also constructs a proof sequence to show the provability of the judgement declaring the outcome. Not only does this manner of generation ensure the program is provably correct with respect to the specification, the proof sequence may be independently verified by checking that the rule applications between judgements are valid. Just like a pen and paper proof is an entity in its own right, independent of the process that went into writing it, the proof sequence is entirely independent from the means of generation. This kind of independently verifiable evidence is called a *certificate*, and allows voters to verify the count themselves, in addition to having formally verified software.

1.3 Related work

Considering related work, we claim that this approach is an improvement in both stages of verification illustrated in Figure 1.1. In a similar approach taken in [10], a non-classical logic called linear logic is used for the specification. This is based on the observation that just like ballots, assumptions in linear logic (unlike classical logic) may only be used once. The specification is then automatically translated into executable code, and the program generates a certificate in the form of a linear logic proof which may be independently verified to ascertain the correct of the count using a proof checker.

However, to verify that the natural language specification is captured by the formalisation requires a considerable level of familiarity with linear logic. We claim that the gap between the original text and the specification is smaller under the proof rules approach, and written in a more familiar system of logic.

We also claim that while the approach in [10] relies on using a pre-existing proof checker, in which trust of correctness must be placed, the proof sequence generated in our approach is simple enough to check that the technical skills required for the individual to write their own proof-checker in a main stream programming language are not beyond a first year programming course.[16, p.3] This also opens up a wide pool of potential electronic scrutineers.

In verifying the step from specification to implementation, other approaches tend not to generate independent certificates, such as [3] and [9]. This means voters must rely on the correctness of a whole chain of tools used in the code and verification, without a secondary piece of evidence.

Chapter 2

Type theory

In this section, we introduce the theory that allows us to implement vote counting rules as proof rules. This begins with a discussion of the concept of a type, before establishing its position at the intersection of mathematical logic and programming by considering how we interpret classical logic. The basic features of type theory are presented, as well as other features specific to the type theory used for our implementation.

2.1 Types

The concept of a type originates with the interest in foundational mathematics at the turn of the 20th century. In particular, it was part of a response to Russell's paradox, a contradiction that arises by considering the set of all sets not containing themselves. To formalise this idea, let ϕ be the predicate holding for a set X if $X \notin X$. If we allow for *unrestricted set comprehension*, that is, for any predicate we can form the set of objects for which the predicate holds, then we can form Russell's set,

$$R = \{X \mid \phi(X)\}.$$

Russell's paradox arises by asking whether or not R is a member of itself. By definition of ϕ, if R is not a member of itself then $\phi(R)$ holds, therefore R must contain itself by definition conditionsof R. Conversely, by definition of R, if R is a member of itself then $\phi(R)$ holds, therefore R must not contain itself by definition of ϕ. As such, we have the paradox expressed symbolically as $R \in R \Leftrightarrow R \notin R$.

One significance of this paradox is that it indicates an inconsistency in the set

theory of the time. An inconsistency in set theory, where both P and $\neg P$ may be deduced, is said to undermine mathematics as the logical consequence is an *explosion* – the deduction of any proposition Q.

The significance in this context is Russell's response to the paradox. Identifying that the paradox stemmed from a lack of restriction on the predicates that could define a set, Russell defined a *type structure*. This placed 'primary objects' at the base, in 'type 0', and then arranged predicates in a hierarchy. A property predicated of a primary object exists in 'type 1', a property predicated of a property of a primary object exists in 'type 2', and so on. Under this stipulation, the set R may not be defined and the paradox is blocked.

2.1.1 Types versus sets

This solution to Russell's paradox does not suggest that sets must be replaced by types for a consistent foundation, and of course a revised set theory with restricted set comprehension serves well as a foundation for mathematics. However, developing the notion of a type leads to some very useful mathematics that, due to its *computational content*, is well-suited to applications from the viewpoint of logic. The specification of vote-counting rules will be a demonstration of how using types can be more convenient, and fruitful, than using sets. Before establishing the link to computation, we begin by building familiarity with the primitive concept of types by comparison to sets, in order to gain an intuition of types beyond a first impression as 'strange sets'.

Mathematical objects are, in a sense, naturally 'typed'. Although a pair is defined by an implementation, for example as a Kuratowski pair,

$$(a, b) := \{\{a\}, \{a, b\}\},$$

it is standard practice outside of set theory to treat it more like a synthetic object, without regard for its implementation. Another example of this practice is a function $f : A \to B$, given formally as a *functional relation* – a subset of the cartesian product $A \times B$ satisfying certain properties. It is more customary to regard the definition of a function $f : A \to B$ as a *typing judgement*, saying the object f has the type of a function with domain A and codomain B, where 'type of a function' assigns to f the familiar operational behaviour of a function. In type theory, this working intuition is captured formally. Pairs and functions are given as types directly though rules that specify the primitive operations used to

manipulate an object. In this sense, types are higher-level than sets and because they are defined by rules, which by nature are *procedural*, this enables a close connection to computation.

Type theories are deductive systems, as discussed in Chapter 1, and so built by specifying judgements and rules. In contrast, set theory consists of two layers – the deductive system of first-order logic and inside this system, a collection of axioms of the theory, such as ZFC. This means mathematical objects and propositions about objects are distinct in set theory, since the former are sets and the latter are judgements in a deductive system. In type theory, there is a single basic notion, meaning propositions are also types.

The basic judgement in type theory is of the form $a : A$, read as 'a is a term of type A'. Although in certain contexts this may be thought of as the judgement 'a is a member of A', it is not equivalent to the set-theoretic $a \in A$, which rather than being a judgement is a *proposition* giving a relation between two objects. As a proposition, $a \in A$ may be true or false, while the judgement $a : A$ cannot be proved true or false – it is either a valid judgement in the system or not. Furthermore, every object must have a type whereas in set theory, an object can exist independently of a set.

The interpretation of the judgement $a : A$ is important in understanding type theory, and forms the focus of the next section. In particular, we will see how changing the classical interpretation of judgements in propositional logic leads to a system of logic isomorphic to a theory of computation.

2.2 Constructive logic

A formal language can be considered purely syntactic manipulation in the absence of an *interpretation* – the assignment of meaning to symbols. In the case of propositional logic, an interpretation is classically provided by assigning to each atomic proposition a truth value, so that 'A' becomes the judgment 'A is true'. Connectives may then be understood in terms of the constituent propositions via a truth table, as in Figure 2.1.

These are known as the Tarski semantics for propositional logic, and ask the question that mathematical logic is traditionally concerned with – 'when is A true?'. However, this is not the only judgement that may be considered.

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$	$\neg A$
T	T	T	T	T	T	F
F	T	F	T	T	F	T
T	F	F	T	F	F	
F	F	F	F	T	T	

Table 2.1: Truth table providing an interpretation for propositional logic.

2.2.1 BHK-interpretation

In the context of mathematics, there is another important judgement that comes from asking the question 'what is a proof of A?'. The Brouwer-Heyting-Kolmogorov (BHK) interpretation is concerned with the judgement 'p is a proof of A', and so gives an account of propositional and first-order logic in terms of *proof conditions*, rather than *truth conditions*. This interpretation leads to a subsystem of classical logic, in the sense that certain classical principles are no longer valid, known as constructive logic.

The BHK-interpretation first assumes an intrinsic understanding of a proof for an atomic proposition, where proof means some kind of convincing mathematical argument. Proof conditions may then be given for each connective in terms of the constituent propositions. We now develop propositional constructive logic by reinterpreting the judgements in the natural deduction rules for the main connectives – conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$ and the logical constant for falsehood $\perp$, read 'absurdity'. Conditions for the remaining connectives, negation $\neg$ and the biconditional $\Leftrightarrow$, are then supplied by the following definition.

Definition 2.1. *Let $\neg$ and $\Leftrightarrow$ be abbreviations given by*

$$\neg A := A \Rightarrow \perp$$
$$A \Leftrightarrow B := (A \Rightarrow B) \wedge (B \Rightarrow A)$$

Remark 2.2. *The BHK-interpretation generally refers to the fragment of rules below called* introduction rules*, saying what constitutes a proof involving a connective. We provide the proof conditions interpretation of all the deduction rules, using [21] and [11] for reference.*

Conjunction

To illustrate the shift to a proof conditions interpretation, we begin with the truth conditions for the rules governing conjunction. Firstly, note that implicit in the judgement 'A is true' is a second judgement, saying that the identifier A is a proposition. This means we also require the more primitive judgement 'A is a proposition'. Thus the first rule is a *formation rule*, asserting an identifier to be a proposition:

$$\frac{\Gamma \vdash A \text{ is a proposition} \qquad \Gamma \vdash B \text{ is a proposition}}{\Gamma \vdash A \wedge B \text{ is a proposition}} \wedge_F$$

This says that the conjunction of two propositions is also a proposition. As in the deductive systems of Chapter 1, Γ is the *context*, a set of zero or more assumed propositions, and $\vdash$ is the *entails* relation, saying the righthand side follows logically from the lefthand side.

The remainder of the rules concern the main judgement 'A is true'. The *introduction rule* says that if we have two propositions that are both true then their conjunct, which is a proposition by the formation rule, is also true. Since it is understood that truth judgements are only made about propositions, this is given as:

$$\frac{\Gamma \vdash A \text{ is true} \qquad \Gamma \vdash B \text{ is true}}{\Gamma \vdash A \wedge B \text{ is true}} \wedge_I$$

Two *elimination rules* allow the deduction of each conjunct separately from the conjunction:

$$\frac{\Gamma \vdash A \wedge B \text{ is true}}{\Gamma \vdash A \text{ is true}} \wedge_{E1} \qquad\qquad \frac{\Gamma \vdash A \wedge B \text{ is true}}{\Gamma \vdash B \text{ is true}} \wedge_{E2}$$

Shifting now to a proof conditions interpretation for $\wedge$, the formation rule remains the same but we update the introduction rule:

$$\frac{\Gamma \vdash p \text{ is a proof of } A \qquad \Gamma \vdash q \text{ is a proof of } B}{\Gamma \vdash (p, q) \text{ is a proof of } A \wedge B} \wedge_I$$

This stipulates what a proof of $A \wedge B$ looks like, and it is what we would expect – namely a pair consisting of a proof of A and a proof of B. Henceforth, we abbreviate the judgement 'p is a proof of A' using the notation $p : A$. The elimination rules become:

$$\frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash \pi_1(p) : A} \wedge_{E1} \qquad \frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash \pi_2(p) : B} \wedge_{E2}$$

where π_1 is the first projection and π_2 is the second projection. Again, this is what we would expect. A proof of A is the first projection of a pair consisting of a proof of A and a proof of B, in order, while a proof of B is the second projection. Read together, the introduction rule says pairs of proofs are proofs of the conjunction, and the elimination rule gives the *closure*, ruling out any other kind of proof of a conjunction.

In changing the judgement, the rules include a new object – not just the proposition A but the proof object p. As such, we have a new kind of rule governing how a proof of a proposition can be simplified, accounting for the possibility of different proofs of the same proposition. They are called *computation rules*, and for conjunction they take the form:

$$\pi_1(p, q) \rightsquigarrow p$$
$$\pi_2(p, q) \rightsquigarrow q$$

The symbol $\rightsquigarrow$ is read as 'reduces to'. This specifies the computational behaviour of π_1 and π_2. The computation rule can also be seen as describing the interaction between the proof term in the conclusion of the introduction rule and the proof terms in the conclusions of elimination rules.

Implication

We continue by presenting the deduction rules for the remaining connectives with this interpretation. The formation rule for implication says that from two propositions A and B, we can form the proposition $A \Rightarrow B$:

$$\frac{\Gamma \vdash A \text{ is a proposition} \qquad \Gamma \vdash B \text{ is a proposition}}{\Gamma \vdash A \Rightarrow B \text{ is a proposition}} \Rightarrow_F$$

The introduction rule says that if a proof x of A entails a proof p of B, then there is a proof of $A \Rightarrow B$. In other words, a proof of $A \Rightarrow B$ is a function mapping a proof of A to a proof of B.

$$\frac{\Gamma, x : A \vdash p : B}{\Gamma \vdash \lambda x.p : A \Rightarrow B} \Rightarrow_I$$

The function is given using *λ-abstraction*, a means defining a function anonymously. For example, $\lambda x.p$ expresses the function given by the more familiar notation $f(x) = p$, except without ascribing it the name f. We can also specify the type of x by writing $(\lambda x : A).p$. The elimination rule stipulates going from a proof of $A \Rightarrow B$ and a proof of A, to a proof of B. Since a proof of $A \Rightarrow B$ is a function, this is just function application.

$$\frac{\Gamma \vdash q : A \Rightarrow B \qquad \Gamma \vdash a : A}{\Gamma \vdash q(a) : B} \Rightarrow_E$$

Finally, the computation rule relates the proof term in conclusion of the introduction rule to the proof term in the conclusion of the elimination rule. It says that the application of a function $(\lambda x : A).p)$ to a reduces to p with a substituted for every instance of x in p. The notation for this is $p[a/x]$ and so the rule is given:

$$((\lambda x : A).p)a \rightsquigarrow p[a/x]$$

Absurdity

The absurdity, $\bot$, behaves differently to the other connectives. There is the obvious formation rule:

$$\frac{}{\bot \text{ is a proposition}} \bot_F$$

however there is no introduction rule. This reflects the meaning of absurdity – it does not make sense to have a proof of the absurd proposition and so $\bot$ is defined by having no possible proof. The elimination rule is a closure condition saying not only is there no way to get a proof of $\bot$, but if there is a proof of $\bot$ then the system must crash and so anything can be proved. This is expressed by $abort_A p$.

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash abort_A p : A} \bot_E$$

There are no computation rules for $\bot$.

Disjunction

Disjunction is left until last as it has the most significant change in interpretation. The formation rule is as expected:

$$\frac{\Gamma \vdash A \text{ is a proposition} \qquad \Gamma \vdash B \text{ is a proposition}}{\Gamma \vdash A \vee B \text{ is a proposition}} \vee_F$$

The introduction rule says that a proof of $A \vee B$ is a pair (i, q) where either $i = 0$ and q is a proof of A, or $i = 1$ and q is a proof of B. In general, we may not be able to determine which disjunct is being proved, so we require the proof to carry that information. The rules are written:

$$\frac{\Gamma \vdash p : A}{\Gamma \vdash (0, p) : A \vee B} \vee_{Il} \qquad \frac{\Gamma \vdash q : B}{\Gamma \vdash (1, p) : A \vee B} \vee_{Ir}$$

Where the proof conditions interpretation of the other connectives gave us something that behaved the same way as the truth interpretation, but carried more information in the form of proof objects, disjunction is different. Consider the structure of a proof by contradiction, used to show that there exists an object x with certain properties given by P. The law of excluded middle, informally called a 'law' but either given directly as a rule in classical propositional logic or derived from a *double negation elimination* rule, says that either $\exists x.P(x)$ is true or $\neg\exists x.P(x)$ is true. Suppose that assuming $\neg\exists x.P(x)$ entails a contradiction, $Q \wedge \neg Q$. From the law of non-contradiction, again only informally called a law but either captured in a rule or deduced from other rules, $\neg\exists x.P(x)$ cannot be true. Therefore $\exists x.P(x)$ must be true.

Reading this in terms of proofs, if $\neg\exists x.P(x)$ entails a contradiction then there is no proof of $\neg\exists x.P(x)$, however this does not provide a proof of $\exists x.P(x)$. In other words, the law of excluded middle is denied since a disjunction requires a proof of one of the disjuncts, and the knowledge of which disjunct is being proved. This is the sense in which a proof conditions interpretation gives a logic that is *constructive*. It is not sufficient to show the non-existence of an object leads to a contradiction; rather, to prove there exists an object x with certain properties requires a finite procedure to construct the object.

Returning to the rules for $\vee$, the elimination rule says that if there is a proof of a disjunction, and a proof of another proposition C is entailed by a proof of each of the disjuncts, then this forms a proof of C:

$$\frac{\Gamma \vdash p : A \vee B \qquad \Gamma \vdash f : A \Rightarrow C \qquad \Gamma \vdash g : A \Rightarrow C}{\Gamma \vdash \mathit{cases}\ p\ f\ g : C} \vee_E$$

The notation *cases* is just used to capture how the proof of C is built from the proofs p, f and g, depending on which disjunct p proves. This becomes clear with the computation rules specifying the evaluation of *cases*:

$$\mathit{cases}\ (0, p)\ f\ g \rightsquigarrow f(p)$$
$$\mathit{cases}\ (1, p)\ f\ g \rightsquigarrow g(p)$$

The first rule says that if p is a proof of the first disjunct A, then a proof of C is obtained by applying f, a function from A to C, to p. The second rule says that if p is a proof of the second disjunct B, then a proof of$C$ is obtained by applying g, a function from B to C, to p.

2.3 Curry-Howard isomorphism

Note that by simply changing the main judgement in our propositional calculus, we have described familiar mathematical objects as proofs of propositions, for example pairs and functions. The history of type theory continued from Bertrand Russell to Alonzo Church, who developed the concept of a type into a formal system called the 'Simply Typed λ-calculus',[8] intended as a foundation for mathematics. Such a foundation defined familiar mathematical objects, such as pairs and functions, in terms of the rules governing their behaviour. Due to this procedural nature, it also served as a model of computation.

Now suppose we change the judgement 'A is a proposition' to 'A is a type', and the judgement 'p is a proof of A' to 'p is a term of type A'. It turns out that this is sufficient to give a type theory – a formal system based on types. Changing the judgements in this way brings together two systems, constructive logic and type theory, whose initial development was unrelated. This is known as the Curry-Howard isomorphism, and says we can interpret propositions as types and proofs as terms, also called 'programs'. It is this correspondence between logical operators and type-theoretic operators that allows us to directly translate the mathematical formalisation of vote counting rules as proof rules into a type theory, with minimal gap between the two.

Considering the rules in terms of types, the formation rules explain what the types of the system are, and the introduction and elimination rules provide rules for an expression to be well-typed. These are said to describe the *static* part of the language.[21, p.78] The new rule introduced under the proof theoretic interpretation, the computation rule, introduces a new *dynamic* component to the language specifying how expressions may be reduced to simpler forms, or in this sense, *evaluated*.

We now present the main rules of type theory according to Martin-Löf type theory (1972) given in [15], using [20] as a reference. It was developed with the Curry-Howard isomorphism in mind as a foundation for constructive mathematics, and forms the basis of the system we will use. While it would be sufficient to

simply change the judgements in the rules governing the connectives of propositional logic, for the sake of being convinced that the same system was developed independently, we focus on how familiar mathematical objects might be given a rule-based formalisation.

2.3.1 Function type

The most basic type is the function type, corresponding to implication under the Curry-Howard isomorphism. Suppose we have primitive types A and B, and we want to form the type of functions with domain A and codomain B. This is given by the formation rule:

$$\frac{\Gamma \vdash A \text{ is a type} \qquad \Gamma \vdash B \text{ is a type}}{\Gamma \vdash A \to B \text{ is a type}} \to_F$$

By this rule, a function is given as a primitive type, rather than being define via an implementation as a functional relation. A function is constructed by λ-abstraction according to the introduction rule, which in the context of types, is also known as the *constructor*:

$$\frac{\Gamma, x : A \vdash p : B}{\Gamma \vdash \lambda x.p : A \to B} \to_I$$

The elimination rule expresses the application of a function $f : A \to B$ to a term in the domain to get a term in the codomain, $f(a)$:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \to_E$$

The computation rule then specifies the action of the elimination rule on a constructor:

$$((\lambda x : A).p)a \rightsquigarrow p[a/x]$$

This is just evaluation of a function by replaced every occurrence of x in p by a.

2.3.2 Product type

The product type corresponds to conjunction under Curry-Howard. Suppose A and B are types, then we can form the type $A \times B$, called the cartesian product. The terms are intended to be ordered pairs, and once more, they are given as a primitive concept rather than an implementation in sets.

The obvious way to construct pairs is to take an $a : A$ and a $b : B$, and form $(a, b) : A \times B$. This is the familiar introduction rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

The elimination rule describes how pairs are used. Pairs are used by defining functions out of them – a first projection and a second projection:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_2(p) : B}$$

The computation rule then relate the constructor and eliminators, that is, specifies the expected operation of π_1 and π_2:

$$\pi_1(p, q) \rightsquigarrow p$$
$$\pi_2(p, q) \rightsquigarrow q$$

2.3.3 Sum type

Given A, B types we can form the sum or coproduct type $A + B$. This corresponds to disjoint union in set theory, and disjunction under the Curry-Howard isomorphism. As expected, there area two ways to form a disjoint union:

$$\frac{\Gamma \vdash p : A}{\Gamma \vdash \mathsf{inl}(p) : A + B} \qquad \frac{\Gamma \vdash q : B}{\Gamma \vdash \mathsf{inr}(p) : A + B}$$

We change $(0, p)$ to inl and $(1, p)$ to inr, for left injection and right injection respectively. To eliminate a sum type, we construct a function out of it:

$$\frac{\Gamma \vdash p : A + B \qquad \Gamma \vdash f : A \Rightarrow C \qquad \Gamma \vdash g : A \Rightarrow C}{\Gamma \vdash \mathit{cases}\ p\ f\ g : C}$$

Then the computation rules describe the operational behaviour of *cases* by relating it to left injection and right injection from the introduction rules:

$$\mathit{cases}\ \mathsf{inl}(p)\ f\ g \rightsquigarrow f(p)$$
$$\mathit{cases}\ \mathsf{inr}(p)\ f\ g \rightsquigarrow g(p)$$

2.4 Extending to first-order logic

So far we have only looked at the propositional fragment of logic. The propositions as types interpretation also extends to first-order logic. We begin from the viewpoint of logic before considering their counterparts in type theory.

2.4.1 Universal quantifier

Under the BHK-interpretation, a proof of $(\forall x : A).P$ is a function f, mapping each $a : A$ to a proof $f(a)$ of $P[a/x]$, where every occurrence of x in P is replaced by a. To make sense of this, we mix the two interpretations of propositions and types, naturally thinking of x as a term of type A and P as a proposition. In the case where P does not contain any occurrences of x, this interpretation is the same as for the connective $\Rightarrow$.

We have the introduction rule:

$$\frac{\Gamma, x : A \vdash p : P}{\Gamma \vdash (\lambda x : A).p : (\forall x : A).P}\ \forall_I$$

The type $(\forall x : A).P$ may be thought of as an indexed family of proofs. The elimination rule corresponds to obtaining a single proof from this indexed familiy:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : (\forall x : A).P}{\Gamma \vdash f(a) : P[a/x]}\ \forall_E$$

and the computation rule relates the introduction and elimination rules:

$$((\lambda x : A).p)a \rightsquigarrow p[a/x]$$

2.4.2 Existential quantifier

Under the BHK-interpretation, a proof of $(\exists x : A).P$ is a pair (a, p), where $a : A$ and p is a proof of $P[a/x]$. We think of this as providing a *witness* for the existential claim, that is, an object a, along with a proof that the proposition P holds for a. This corresponds to the strengthened notion of existence in constructive logic previously discussed. The introduction rule is given as:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash p : P[a/x]}{\Gamma \vdash (a, p) : (\exists x : A).P}\ \exists_I$$

The elimination rules project the first and second components of a pair:

$$\frac{\Gamma \vdash p : (\exists x : A).P}{\Gamma \vdash Fst\ p : A}\ \exists_{E1} \qquad\qquad \frac{\Gamma \vdash p : (\exists x.A).P}{\Gamma \vdash Snd\ p : P[Fst\ p/x]}\ \exists_{E2}$$

The operational behaviour of Fst and Snd are given by the computation rules:

$$Fst(p, q) \mapsto p$$
$$Snd(p, q) \mapsto q$$

Note that these rules are very similar to those for conjunction – if P does not depend on x these are the conjunction rules.

2.4.3 Dependent types

The types corresponding to the universal and existential quantifiers are known collectively as *dependent types*. The universal quantifier, given by $(\forall x : A).P$ or $\prod_{x:A} P$, is a *dependent function type*, a generalisation of a function type to an indexed family of functions. The existential quantifier, given by $(\exists x : A).P$ or $\sum_{x:A} P$, is a *dependent pair type*, as the second component of the pair depends on the first.

Dependent pair types, in particular, are very expressive in the sense that they have multiple interpretations. In addition to being considered a generalisation of the product type, $(\exists x : A).P$ can be thought of as the *subset* type, expressing $\{a \in A | P(a)\}$. The dependent pair does not just pick out a subset, it also pairs each element a with evidence $P(a)$ for their inclusion in the subset. This subset interpretation will be used extensively in the implementation of vote counting as mathematical proof.

2.5 More features of type theory

The vote counting as mathematical proof approach is implemented in the interactive theorem prover *Coq*. An interactive theorem prover is a software tool that assists the user in developing formal proofs. *Coq* is based on a type theory called the Calculus of Inductive Constructions (CIC), of which Martin-Löf type theory is a fragment. There are some other features of CIC that will be relevant to our implementation.

2.5.1 Universes

It is the case that every term has a type. In particular, since a type is a term, this means that types also have types. An infinite hierarchy of types is thus required, known as a hierarchy of universes. In CIC, this is given by the following definition:

Definition 2.3. *The type of types are* sorts. *The set of sorts* $\mathcal{S}$ *is given by*

$$\mathcal{S} := \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}$$

where

$$\begin{aligned}\mathsf{Prop} &: \mathsf{Type}(1)\\ \mathsf{Set} &: \mathsf{Type}(1)\\ \mathsf{Type}(i) &: \mathsf{Type}(i+1).\end{aligned}$$

In CIC, while we operate with the Curry-Howard isomorphism in mind, a distinction is made between Prop and Set for technical reasons that seems to separate out propositions and types. Formally, Prop is the type of logical propositions with terms corresponding to proofs, while Set is the type of types as used in programming with terms corresponding to programs. Set may just be though of as the sort Type with a specified level in the hierarchy, while Prop is thought of in terms of its operational behaviour. Terms of type Prop have their computational content forgotten, and so correspond to a propositions under a truth conditions interpretation.

Remark 2.4. *This language can be confusing. In our use of CIC to come, we will use the terms 'propositions' and 'types' according to the Curry-Howard isomorphism, not to refer to the sorts* Prop*,* Set *or* Type*. The use of* Prop *and* Type *as sorts in our implementation does not correspond to our natural language use of 'propositions' and 'types', rather a consideration of the operational behaviour of* Prop *as proof-irrelevant.*

2.5.2 Inductive types

Just as types may be formed by the formation rules from before, types may also be formed by induction rules. An inductive type T is thought of as being 'freely generated' by a finite collection of constructors, where constructors are functions with zero or more arguments and codomain T. An inductive type is freely generated in the sense that it is built by repeated application of the constructors. In this way, infinite types are defined by recursion and have properties proved of them by induction. In type theory, recursion and induction are the same – a proof by induction is a proof object defined by recursion.

Some inductive types are associated with induction in mathematics, such as the natural numbers, and others are not, such as lists. For example, the type $\mathbb{N}$ of natural numbers is given as an inductive type with two constructors; $0 : \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. This says that every term of type $\mathbb{N}$ is either 0 or the constructor succ applied to a previously constructed term of the type. This builds

the terms $\mathsf{succ}(0), \mathsf{succ}(\mathsf{succ}(0)), \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0)))$ and so on. An example of a type removed from the mathematical notion of induction is the type of finite lists $\mathsf{List}(A)$, with elements terms of type A. This has two constructors: $\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$.

2.5.3 Dependent inductive types

The combination of inductive types and dependent types leads to a very precise and expressive language. They are combined by giving inductive types with dependently typed constructors, for example, the following definition in *Coq*:

```
Inductive even : nat -> Prop :=
  even0 : even 0
| even2 : forall x : nat, even x -> even (S (S x)).
```

where the second constructor, `even2`, is dependently typed. Thought of another way, this definition specifies two ways of giving evidence that a number is even. The first constructor `even0` is atomic evidence that zero is even, while `even2` allows us to construct evidence that `S (S x)` is even from evidence that `x` is even. It is the expressiveness of dependent inductive types that will allow us to implement vote counting as mathematical proof in *Coq*.

2.6 The *Coq* proof assistant

The *Coq* proof assistant will act as a logical framework in which we build the deductive system of voting counting as mathematical proof, discussed in Chapter 1. Before starting this implementation, we provide some remarks on how mathematics is undertaken in *Coq*, and how programs may automatically be generated from functions.

2.6.1 Goals and tactics

Under the propositions as types interpretation, proving a proposition corresponds to providing a term of the associated type. Mathematics in type theory is constructing terms of particular types. Such proof term can be very complicated, but since they governed by procedural rules, the *Coq* system is able to provide tools in the form of *goals* and *tactics* to assist in building terms piece by piece, as opposed to giving the correctly typed term in its entirety.

A goal refers to type of which the user wants to construct a term. Commands known as tactics can then be applied to break down the goal down into simpler *subgoals*.

Definition 2.5. *Let G be a goal. A tactic applied to G produces a (possibly empty) list of subgoals $G_1, \ldots, G_k$, and has an associated function taking terms $g_1 : G_1, \ldots, g_k : G_k$ to a term $g : G$, i.e. a solution of the goal.*

The construction of a proof term in *Coq* ends with the command `Qed`, which checks whether the proof is finished, saving the proof and generating the complete proof term. The term is submitted to *Coq*'s *type checker* to ensure that it is well-formed and typed before being accepted. The type checker is the very small, thoroughly tested, code base for *Coq* in which trust lies.

2.6.2 Program extraction

Functions written in *Coq* correspond to functions that could be written in an ordinary functional programming language. *Coq* has the ability to map functions developed in the system to functions in a specified programming language, and in this way, can produce software in which the behaviour of the extracted function is faithfully described by the function in *Coq*.[5, p.285] While the *Coq* extraction mechanism is highly trusted, our approach to electronic vote counting means also produces an independently verifiable certificate so it is not necessary to look further into the extraction mechanism here.

Chapter 3

Majority Criterion

To compare vote-counting protocols *objectively* – free from any personal bias towards methods producing a more politically favourable outcome – mathematically defined *voting system criteria* are used. These are statements of properties that are potentially desirable for a vote-counting protocol to satisfy. One such example is the *majority criterion*, commonly defined in the literature for a single-winner voting system, but easily generalised to the case of multiple winners and adapted to STV as follows.

Definition 3.1. *The majority criterion states that if candidate c receives more than 50% of the first preference votes, then c is a winner of the election.*

In this chapter, we prove that the formulation of Simple STV under 'vote-counting as mathematical proof' in [16] satisfies the majority criterion. This corresponds to proving the property for the specification inside *Coq*. If it is true of the specification, then it is true of any implementation conforming to this specification, such as the implementation by program extraction.

Proof of the majority criterion acts as a sanity check. It is not part of the formal verification from specification to implementation, but acts as a verification measure in the move from vote counting protocol to specification, to make sure the specification has accurately captured the protocol and so has the same properties.

It also serves as a proof of concept, demonstrating that the 'vote-counting as mathematical proof' approach readily accommodates comparison by voting system criteria. Since electronic vote-counting means the ease of manual counting is no longer a factor in the design of a voting systems, it allows for the development of more complex yet fairer voting systems. The ability to conveniently prove voting system criteria is important to compare any new systems.

To this end, we begin by describing the specification of Simple STV as logic-lookalike rules, first by providing the mathematical formalisation then the translation into *Coq*. This is not original work – the mathematical formalisation and *Coq* implementation come from [16]. Following the same two stages of development, we state and prove the majority criterion. This is an original addition to the *Coq* implementation from [16]. To prove the majority criterion, we first prove a property that holds at every stage of the count, known as an *invariant* of the system. The main theorem is then deduced used the invariant.

3.1 Simple STV

In an STV system, ballots are ranked lists of candidates ordered in terms of personal preference. The system relies on having a *quota* – a number of votes that a candidate must receive in order to be elected. The specification in [16] is fully general with respect to the quota. The a natural language specification of Simple STV is given as follows:

1. If a candidate has enough first preferences to meet the quota, they are declared elected and any votes for this candidate surplus to the quota are transferred.

2. If all first preferences are counted and the number of seats is strictly smaller than the sum of the number of candidates that are either still in the running or already elected, then a candidate with the least number of first preferences is eliminated and their votes are transferred.

3. If a vote is transferred, it is assigned to the next candidate on the ballot.

4. Vote counting is finished if either the number of elected candidates is equal to the number of available seats, or if the number of remaining hopeful candidates plus the number of elected candidates is less than or equal to the number of available seats.

3.1.1 Mathematical formalisation

Let C be a set of candidates running for election and $B \in \mathsf{List}(C)$ be a list of candidates with order corresponding to a preference ranking, representing a ballot. Note that this definition of a ballot is intended to be very general – it does not exclude multiple rankings for a single candidate, nor stipulate that all

candidates must be ranked. However, this is easily done, as will be seen in the formalisation of a real-world protocol in Chapter 4.

Definition 3.2. *If* $b \in \mathsf{List}(B)$ *represents the list of ballots cast,* $q \in \mathbb{N}$ *represents the quota and* $s \in \mathbb{N}$ *represents the number of seats available to be filled, then a judgement takes one of two forms:*

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

where $u \in \mathsf{List}(B)$ *represents the list of uncounted ballots,* $a : C \to \mathsf{List}(B)$ *the* assignment *recording for a particular candidate c the ballots with first preference c that have been counted,* $t : C \to \mathbb{N}$ *the running tally,* $h : \mathsf{List}(C)$ *the list of candidates still in the running and* $e : \mathsf{List}(C)$ *the list of already elected candidates; or*

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

where $w \in \mathsf{List}(C)$ *represents the list of winners of the election.*

The first judgement corresponds to an intermediate state of the count, while the second judgement corresponds to a final state of the count.

Remark 3.3. *Certain choices were made above with the eventual* Coq *formulation in mind, rather than purely mathematical reasons. For example, the hopeful candidates need not be ordered. It would make sense to represent them as a set, however it is easier to work with lists in* Coq*.*

Using the judgements, we define rules mimicking the form of deduction rules in proof theory. Side conditions are used to express relations between the vote counting state in the premise and the vote counting state in the conclusion.

Definition 3.4. *There are eight deduction rules.*

Axiom *describes the initial state of the vote-count. Let* $\mathsf{nty} : C \to \mathbb{N}$ *be the null tally given by* $\mathsf{nty}(c) = 0$ *for all* $c \in C$*, and let* $\mathsf{nas} : C \to \mathsf{List}(B)$ *be the null assignment given by* $\mathsf{nas}(c) = []$ *for all* $c \in C$*. Then define the rule:*

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

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

read as:

> *"At the start of the count all ballots are uncounted, no ballots have been assigned to a candidate, the running tally for every candidate is zero, no candidates have been elected, every candidate is in the list of hopeful candidates and the list of hopeful candidates is pairwise distinct."*

Count one vote *captures counting the first preference on a ballot. To specify that the list of uncounted votes is updates to remove the counted vote, let* eqe *be the relation*

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

holding for $x \in X$, a list $l \in \mathsf{List}(X)$ and a list $l' \in \mathsf{List}(X)$ when l' is equal to l except that l' additionally contains x at an arbitrary position. To express that the assignment is updated, let add *be the relation*

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

holding for a candidate c, a ballot v, an assignment a and an assignment a' when a equals a' except that the evaluation of the assignment a' for c includes v inserted at an arbitrary position. To express that the tally is updated, let inc *be the relation*

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

holding between a candidate c, an 'old' tally t and a 'new' tally t' when the new tally is the old tally with the value for c incremented by one. Then define the rule:

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

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

read as:

> *"If there is an uncounted vote with first preference c, c is a hopeful and the tally of c is below the quota, record this vote by adding it to the assignment for c and increase the tally for c by one."*

Elect *applies when a candidate has reached the quota. Let* eqe *be defined as above, then define the rule:*

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

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

read as:

> *"If there is a hopeful candidate who has reached the quota and there are still seats available, then this candidate is declared elected by moving them from the list of hopefuls to the list of elected candidates."*

Transfer votes *applies when a candidate is no longer in the running and the uncounted ballots are updated to transfer their votes to the next preference. To ensure the ballots in the list of uncounted votes are updated, let* repl *be the relation*

$$\mathsf{repl}(x,y,l,l') \equiv \exists l_1, l_2.\, (l = l_1 \,⨟ [x] ⨟\, l_2 \wedge l' = l_1 \,⨟ [y] ⨟\, l_2)$$

holding between $x, y \in X$ *and two lists* $l, l' \in \mathsf{List}(X)$ *when* l *is equal to* l' *except with one occurrence of* x *in* l *replaced by* y*. Then define the rule:*

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

- $c \notin h$
- $\mathsf{repl}((c{:}cs), cs, u, u')$

read as:

> *"If there is an uncounted vote with first preference c and c is not a hopeful candidate, then delete this first preference from the ballot."*

Empty votes *applies when there are empty votes in the list of uncounted ballots, either due to initially empty ballots or successive transfers. Define the rule:*

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

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

read as:

> *"If there are uncounted votes with no preferences, they are discarded."*

Transfer least *applies when all of the ballots have been counted but all of the seats have not been filled and a candidate with minimal number of first preferences is eliminated. To update the list of hopeful candidates, let* eqe *be the relation given earlier. Then define the rule:*

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

- $|e| + |h| > s,\ c \in h$
- $\forall d \in h.(tc \leq td),\ \mathsf{eqe}(c, h, h'),\ u = a(c)$

read as:

> *"If the number of available seats exceeds the sum of hopeful and elected candidates, no uncounted votes remain, and candidate c has a minimal number of votes, then c is removed from the list of hopefuls and all votes cast for c are transferred."*

Hopeful win *declares the winners of the election in the case where the number of elected plus hopeful candidates is no greater than the number of sets. Define the rule:*

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

- $|e| + |h| \leq s$
- $w = e \mathbin{;} h$

read as:

> *"If the number of candidates that are either hopeful or elected is less than or equal to the number of seats available, then scrutiny ceases and all candidates that are either elected or hopeful are declared winners of the election".*

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

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

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

read as:

> *"If the number of elected candidates equals the number of seats available, scrutiny ceases and the elected candidates are declared the winners of the election".*

Finally, we recall the definition of provability from Chapter 1 and place it the context of the deductive system just defined.

Definition 3.5. *A state of the count* $(b, q, s) \vdash n$*, where* n *is of the form* $\mathsf{state}(u, a, t, h, e)$ *or* $\mathsf{winners}(w)$ *is said to be* provable *if there exists a sequence of correct applications of the rules in Definition 3.4, ending with* $(b, q, s) \vdash n$ *and starting with* Ax.

3.1.2 Formalisation in *Coq*

We now translate the mathematical formalisation of judgements and rules into types in *Coq*, and explain how the *Coq* extraction mechanism can be applied to a theorem about the existence of winners in order to generate a vote-counting program. The code below comes from the file `STV_majority_criterion.v`.

Implementation 3.6. *We begin by defining the type of candidates and the list of all candidates running in the election. We use the keyword* `Variable`*, which has the expected mathematical meaning and allows us to work in full generality, instantiating with specific candidates later on.*

```
Variable cand: Type.
Variable cand_all: list cand.
```

These types are required to satisfy certain properties. Firstly, the list of all candidates must be pairwise distinct, that is, each member of the list is unique. Secondly, we must specify that every candidate appears in the list of all candidates. Finally, we require that equality of candidates is decidable*, that is, for any two candidates there is either a proof that they equal or a proof that they are not equal.*

```
Hypothesis cand_pd: PD cand_all.
Hypothesis cand_finite: forall c, In c cand_all.
Hypothesis cand_eq_dec: forall c d:cand, {c=d} + {c<>d}.
```

We use the keyword `Hypothesis`*. This defines an arbitrary proof term, for example* `cand_eq_dec` *is an unspecified proof of the proposition given by the type* `forall c d:cand, {c=d} + {c<>d}`*. Note also that in this proposition, instead of using the familiar disjunction* `\/` *we use* `+`*, which is defined in the same way except with values on the level of* `Type` *rather than* `Prop`*. This is the technicality in* Coq *remarked on in Chapter 2, the practical upshot being that values in* `Type` *are not just true or false but also carry* evidence *for the truth or falsity, whereas values in* `Prop` *are only the former.*

A ballot is defined according to the mathematical formalisation as a list of candidates.

```
Definition ballot := list cand.
```

A judgement takes one of two possible forms, so we encode them as an inductive type with two constructors. We call the type `Node`*, the name given to a valid judgement in a proof tree.*

```
Inductive Node :=
   state:                     (** intermediate states **)
       list ballot            (* uncounted votes *)
     * (cand -> list ballot)  (* assignment of counted votes to first preference candidate *)
     * (cand -> nat)          (* tally *)
     * (list cand)            (* hopeful candidates still in the running *)
     * (list cand)            (* elected candidates no longer in the running *)
     -> Node
 | winners:                   (** final state **)
    list cand -> Node.        (* election winners *)
```

We read this as saying there are two ways to construct a term of type `Node`*. The first way is to apply the* `state` *constructor to a five-tuple consisting of a list of uncounted votes, an assignment, a tally, a list of hopeful candidates and a list of elected candidates; the second way is to apply the constructor* `winners` *to a list of candidates.*

The rules are encoded in a single dependent inductive type, more specifically as a parametrised inductive type with dependently typed constructors. This type is referred to as 'type of proofs' in [16], and to minimise confusion we call it the type of proof sequences. It specifies what constitutes evidence for a judgement having the property of provability. The type of proof sequences is parametrised by the list of ballots cast, the quota and the number of seats, and it has seven constructors, one corresponding to each rule. For example, consider the constructor corresponding to the axiom:

```
ax :  forall  u a t h e,           (** start counting **)
(forall c: cand, In c h) ->        (* if all candidates are hopeful and *)
PD h ->                            (* hopefuls are pairwise distinct *)
u = b ->                           (* and the list of uncounted ballots contains all ballots *)
a = nas ->                         (* and the initial assignment is the null assignment *)
t = nty ->                         (* and we begin with the null tally *)
e = nbdy ->                        (* and nobody is elected initially *)
Pf b q s (state (u, a, t, h, e))   (* we start counting with this data *)
```

We read this as saying if we give a piece of evidence each that all candidates are hopeful, the hopeful candidates are pairwise distinct, the list of uncounted ballots contains all ballots, the initial assignment is the null assignment, the initial tally is the null tally and nobody has been elected, then we can construct a term of

the proof type. This corresponds directly to the mathematical formalisation given previously.

The complete type is given below. By comparing each constructor with the corresponding rule given mathematically in Definition 3.4, it is a straightforward exercise to convince oneself that they are the same. The functions described in the mathematical formalisation of Simple STV are all easily defined in Coq. *Their definitions are not provided here but are all included in the code.*

```
Inductive Pf (b: list ballot) (q: nat) (s: nat) : Node -> Type :=
  ax :  forall  u a t h e,            (** start counting **)
  (forall c: cand, In c h) ->         (* if all candidates are hopeful and *)
  PD h ->                             (* hopefuls are pairwise distinct *)
  u = b ->                            (* and the list of uncounted ballots contains all ballots *)
  a = nas ->                          (* and the initial assignment is the null assignment *)
  t = nty ->                          (* and we begin with the null tally *)
  e = nbdy ->                         (* and nobody is elected initially *)
  Pf b q s (state (u, a, t, h, e))    (* we start counting with this data *)
| c1 : forall u nu a na t nt h e f fs, (** count one vote **)
  Pf b q s (state (u, a, t, h, e )) -> (* if we are counting votes, *)
  eqe (f::fs) nu u ->                 (* and have an uncounted vote with first preference f removed *)
  In f h ->                           (* and f is a hopeful *)
  t f < q ->                          (* and this isn't surplus *)
  add f (f::fs) a na ->               (* and the new assignment records the vote for f *)
  inc f t nt ->                       (* and the new tally increments the votes for f by one *)
  Pf b q s (state (nu, na, nt, h, e)) (* we continue with the updated tally and assignment *)
| el : forall u a t h nh e ne c,      (** elect a candidate **)
  Pf b q s (state (u, a, t, h, e)) -> (* if we have an uncounted vote with first preference f *)
  In c h ->                           (* and c is a hopeful *)
  t(c) = q ->                         (* and c has enough votes *)
  length e < s ->                     (* and there are still unfilled seats *)
  eqe c nh h ->                       (* and f has been removed from the new list of hopefuls *)
  eqe c e ne ->                       (* and added to the new list of elected candidates *)
  Pf b q s (state (u, a, t, nh, ne))  (* then proceed with updated hopeful and elected candidates *)
| tv : forall u nu a t h e f fs,      (** transfer vote **)
  Pf b q s (state (u, a, t, h, e)) -> (* if we are counting votes *)
  ~(In f h) ->                        (* and f no longer in the running *)
  rep (f::fs) fs u nu ->              (* and f is being removed from an uncounted ballot *)
  Pf b q s (state (nu, a, t, h, e))   (* we continue with updated set of uncounted votes *)
| ey : forall u nu  a t h e,          (** empty vote **)
  Pf b q s (state (u, a, t, h, e)) -> (* if we are counting votes *)
  eqe [] nu u ->                      (* and an empty vote is removed from uncounted votes *)
  Pf b q s (state (nu, a, t, h, e))   (* continue with the updated set of uncounted votes *)
| tl : forall u a t h nh e c,         (** transfer least **)
  Pf b q s (state ([], a, t, h, e)) -> (* if we have no uncounted votes *)
  length e + length h > s ->          (* and there are still too many candidates *)
  In c h  ->                          (* and candidate c is still hopeful *)
  (forall d, In d h-> t c <= t d) ->  (* but all others have more votes *)
  eqe c nh h ->                       (* and c has been removed from the new list of hopefuls *)
  u = a(c) ->                         (* and marked to be transfered *)
  Pf b q s (state (u, a, t,nh, e))    (* transfer c's votes and proceed with new hopefuls *)
| hw : forall w u a t h e,            (** hopefuls win **)
  Pf b q s (state (u, a, t, h, e)) -> (* if at any time *)
  length e + length h <= s ->         (* we have more hopeful and elected candidates than seats *)
```

```
  w = e ++ h ->                         (* and the winning candidates are their union *)
  Pf b q s (winners (w))                (* then they are declared winners *)
| ew : forall w u a t h e,              (** elected win **)
  Pf b q s (state (u, a, t, h, e)) ->   (* if at any time *)
  length e = s ->                       (* we have as many elected candidates as seats *)
  w = e ->                              (* and the winners are precisely the elected candidates *)
  Pf b q s (winners w).                 (* they are declared the winners *)
```

Comparing this to the mathematical formalisation, a term `p` of type `Pf b q s n` corresponds to a sequence of correct rule applications beginning with the Ax rule and ending in the judgement $(b, q, s) \vdash n$, where n is either of the form $\mathsf{state}(u, a, t, h, e)$ or $\mathsf{winners}(w)$.

Remark 3.7. *The law of excluded middle $A \vee \neg A$ is not constructively valid, since not having a proof of A does not ensure there is a proof $\neg Am$ and vice versa, and so it does not hold in the logic of* Coq*. However, we can prove $A \vee \neg A$ is true for certain A, as was the case for the decidability of equality of candidates.*

This completes the specification of Simple STV. An implementation can be produced directly from this specification by proving that every election has an outcome, that is, a proof sequence ending in a judgement declaring winners. The *Coq* program extraction mechanism then automatically constructs a provably correct program in Haskell that determines the election winners, while also producing a proof sequence that may be independently verified.

Implementation 3.8. *The theorem is formulated as follows.*

```
Theorem ex_winners_pf: forall b q s, q > 0 ->
  existsT w: list cand, Pf b q s (winners w).
```

Rather than `exists` *we use* `existsT`*, which is introduced notation for the type level existential quantifier. The reason for this choice is the same as the choice of* `+` *instead of* `\/` *discussed in Implementation 3.6. That is,* `existT` *carries the evidence of existence, in this case a* proof *of the election outcome, rather than solely the outcome.*

To demonstrate how we may use this theorem to extract a vote counting program, we construct a toy example.

Example 3.9. *The Simple STV implementation was given as a* section *in* Coq*. Within a section, the keywords* `Variable` *and* `Hypothesis` *are used to make definitions local to the section. On closing the section, the variables and hypotheses*

become extra function parameters for the global definitions made within the section, such as those using the keyword `Lemma`, `Theorem` *and* `Define`*.*

This means outside the section, the theorem about the existence of winners corresponds to a function with added parameters for the variables and hypotheses. We define four candidates for our example.

```
Inductive cand := Alice | Bob | Charlie | Deliah.
Definition cand_all := [Alice; Bob; Charlie; Deliah].
```

Proofs are then provided for the following lemmas, corresponding to the hypotheses before.

```
Lemma cand_pd: PD cand_all.
Lemma cand_finite: forall c, In c cand_all.
Lemma  cand_eq_dec : forall c d : cand, {c = d} + {c <> d}.
```

Then we define the main function.

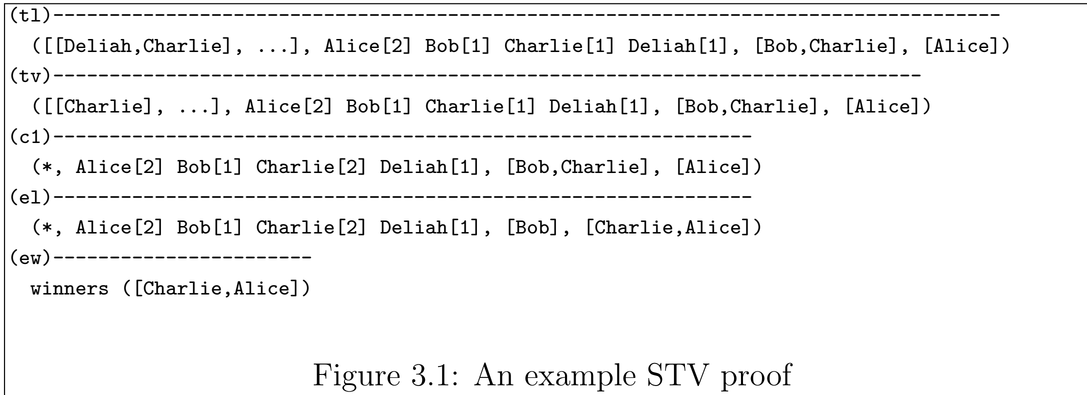
```
Definition cand_ex_winners_pf :=
  ex_winners_pf cand cand_all cand_pd cand_finite cand_eq_dec.
```

Program extraction is then given by the following.

```
Extraction Language Haskell.
Extraction "STVCode" cand_ex_winners_pf.
```

In addition to the extracted code, a 'wrapper' piece of code is written to visualise the proof tree. It is important to note that this does not alter the extracted program, as the code would no longer necessarily be provably correct according to the specification. Figure 3.1 shows a sample output from the extracted program with the wrapper, with horizontal lines representing a deduction from one correct vote-counting state to the next, sanctioned by the rule at the lefthand side.

```
(ax)--------------------------------------------------------------------------------------
  ([[Alice,Bob], ...], Alice[0] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(c1)------------------------------------------------------------------------------------------
  ([[Alice,Charlie], ...], Alice[1] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(c1)-------------------------------------------------------------------------------------------
  ([[Deliah,Charlie], ...], Alice[2] Bob[0] Charlie[0] Deliah[0], [Alice,Bob,Charlie,Deliah], [])
(el)------------------------------------------------------------------------------------------
  ([[Deliah,Charlie], ...], Alice[2] Bob[0] Charlie[0] Deliah[0], [Bob,Charlie,Deliah], [Alice])
(c1)-------------------------------------------------------------------------------------
  ([[Bob,Alice], ...], Alice[2] Bob[0] Charlie[0] Deliah[1], [Bob,Charlie,Deliah], [Alice])
(c1)-----------------------------------------------------------------------------------
  ([[Charlie], ...], Alice[2] Bob[1] Charlie[0] Deliah[1], [Bob,Charlie,Deliah], [Alice])
(c1)-----------------------------------------------------------------------
  (*, Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie,Deliah], [Alice])
```



```
(tl)----------------------------------------------------------------------------------
  ([[Deliah,Charlie], ...], Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie], [Alice])
(tv)-------------------------------------------------------------------------
  ([[Charlie], ...], Alice[2] Bob[1] Charlie[1] Deliah[1], [Bob,Charlie], [Alice])
(c1)------------------------------------------------------------
  (*, Alice[2] Bob[1] Charlie[2] Deliah[1], [Bob,Charlie], [Alice])
(el)------------------------------------------------------------
  (*, Alice[2] Bob[1] Charlie[2] Deliah[1], [Bob], [Charlie,Alice])
(ew)----------------------
  winners ([Charlie,Alice])
```

Figure 3.1: An example STV proof

3.2 Proof of the majority criterion

Now that we have developed the specification of Simple STV, we prove it satisfies the majority criterion. Rather than approaching this proof directly, we use the standard technique of strengthening the induction hypothesis to an *invariant* – a statement that holds at every intermediate state of the count – and proving this first. A suitable invariant is one which is correct, that is, holds at every stage, and is able to be used to prove the theorem.

We begin by providing a mathematical formalisation of the invariant and the theorem, using the same notation as previously introduced for Simple STV.

3.2.1 Mathematical formalisation

Suppose there is a function $\mathsf{cfp} : \mathsf{cand} \times \mathsf{List}(B) \to \mathbb{N}$, read *count first preferences*, that takes a candidate c and a list of ballots l and returns the number of ballots in l for which c is the first preference. As before, let $b \in \mathsf{List}(B)$ represent the list of ballots cast. Then we can state the theorem.

Definition 3.10. *The majority criterion says that candidate c is a winner if* $\mathsf{cfp}(c, b) > |b|/2$*.*

Further, suppose there is a function $\mathsf{ct} : (\mathsf{cand} \to \mathbb{N}) \times \mathsf{List}(C) \to \mathbb{N}$, read *sum of current tallies*, which takes a running tally and a list of candidates $l \in \mathsf{List}(C)$ and returns the sum of the tallies for each candidate in the list, namely

$$\mathsf{ct}(t, l) = \sum_{c \in l} t(c).$$

Then we can state the following lemma.

Lemma 3.11. *If a candidate $c \in C$ satisfies the hypothesis of the majority criterion, $\mathsf{cfp}(c,b) > |b|/2$, then at an intermediate state of the count given by $(b,q,s) \vdash \mathsf{state}(u,a,t,h,e)$, either c is an elected candidate, or c is a hopeful candidate and the following inequalities hold:*

(i) $t(c) + \mathsf{cfp}(c,u) > |b|/2$

(ii) $\mathsf{ct}(t, h \mathbin{;} e) + |u| \leq |b|$

The instance where c is elected is the same statement as the majority criterion. In the case where c is a hopeful candidate, together (i) and (ii) describe an early state of the count in which not enough first preferences have been counted for c to reach the quota and be elected. Specifically, condition (ii) captures that not enough first preferences have been counted, and condition (i) says that taking into account the first preferences for c left uncounted c will satisfy the hypothesis of the majority criterion.

To prove the invariant, we show that it is true at the start of the count, and is then preserved by every valid rule application. Below is an outline of the proof, demonstrating the approach on three rules only, with the full formal proof to be found in the file `STV_majority_criterion.v`, line 1451. Note that this proof is constructive – we prove the disjunction by giving a proof of one of the disjuncts, including the information of which disjunct it proves.

Proof (Lemma 3.13). Consider any $c \in C$ such that $\mathsf{cfp}(c,b) > |b|/2$. The count begins with the following provable state:

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

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

Since no candidate has been elected and every candidate is in the list of hopeful candidates, we want to show that inequalities (i) and (ii) hold. Using the side-conditions for Ax and the definition of nty,

$$\begin{aligned} t(c) + \mathsf{cfp}(c,u) &= \mathsf{nty}(c) + \mathsf{cfp}(c,b) \\ &= 0 + \mathsf{cfp}(c,b) \\ &> |b|/2 \end{aligned}$$

by the initial assumption, so (i) is true. Furthermore,

$$\begin{aligned}\mathsf{ct}(t, h \mathbin{;} e) + |u| &= \mathsf{ct}(\mathsf{nty}, h) + |b| \\ &= 0 + |b| \\ &\leq |b|\end{aligned}$$

so (ii) is also true and the lemma holds for the initial state of the count.

Now assume there is a provable intermediate state of the count given by $(b, q, s) \vdash \mathsf{state}(u, a, t, h, e)$ for which the invariant is true, that is

$$(c \in e) \;\vee\; \Big(c \in h \;\wedge\; \big(t(c) + \mathsf{cfp}(c, u) > |b|/2\big) \;\wedge\; \big(\mathsf{ct}(t, h \mathbin{;} e) + |u| \leq |b|\big)\Big) \quad (3.1)$$

We proceed by showing that if any of the remaining six rules may be applied, then the invariant is also true of the conclusion of the rule.

Suppose the 'count one vote' rule may be applied, corresponding to the following situation:

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

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

We want to show the invariant holds for the conclusion, namely:

$$(c \in e) \;\vee\; \Big(c \in h \;\wedge\; \big(t'(c) + \mathsf{cfp}(c, u') > |b|/2\big) \;\wedge\; \big(\mathsf{ct}(t', h \mathbin{;} e) + |u'| \leq |b|\big)\Big)$$

If the first disjunct of (3.1) is true, then we have the contradiction that c is both an elected candidate and a hopeful candidate, from which we may deduce anything. If the second disjunct of (3.1) is true, then observe that while $u \neq u'$ and $t \neq t'$, the following sums are equal:

$$\begin{aligned}t'(c) + \mathsf{cfp}(c, u') &= t(c) + \mathsf{cfp}(c, u) \\ \mathsf{ct}(t', h' \mathbin{;} e') + |u'| &= \mathsf{ct}(t, h' \mathbin{;} e') + |u|\end{aligned}$$

The desired result follows by substitution. See the code for proofs of these equalities.

Other rules are more involved. Consider, for example, the 'transfer least' rule:

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

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

Showing the invariant holds for the conclusion corresponds to proving

$$(c \in e) \ \vee \ \Big(c \in h' \ \wedge \ \big(t(c) + \mathsf{cfp}(c, u') > |b|/2\big) \ \wedge \ \big(\mathsf{ct}(t, h' \mathbin{;} e) + |u'| \leq |b|\big)\Big).$$

If the first disjunct of (3.1) is true, then the result is obvious. If the second disjunct of (3.1) holds, then there are two cases to consider, as c may or may not be the candidate having their votes transferred. Suppose c is not the candidate having their votes transferred, rather it is some distinct candidate $f \neq c$. Then $c \in h'$ and

$$\begin{aligned} t(c) + \mathsf{cfp}(c, u') &= t(c) + \mathsf{cfp}(c, a(f)) \\ &\geq t(c) + \mathsf{cfp}(c, []) \\ &= t(c) + \mathsf{cfp}(c, u) \\ &> |b|/2 \end{aligned}$$

Furthermore, we can prove that $t(f) = |a(f)|$, so we can show

$$\begin{aligned} \mathsf{ct}(t, h' \mathbin{;} e) + |u'| &= \mathsf{ct}(t, h \mathbin{;} e) + |u| \\ &\leq |b| \end{aligned}$$

So the invariant holds. Now suppose c is the candidate having their votes transferred. Then we can derive a statement that contradicts the initial assumption $\mathsf{cfp}(c, b) > |b|/2$.

For the rules that declare winners, the invariant automatically holds. □

We now use the invariant to prove the main theorem. Proving the majority criterion holds requires placing a reasonable condition on the quota. While the implementation of STV is fully general, the most commonly used quota in STV elections is the *Droop quota*.

Definition 3.12. *The Droop quota q_d is given by*

$$q_d = \frac{|b|}{s+1} + 1$$

The condition we place on the quota for the main theorem is

$$q \geq |b|/2s.$$

This is a somewhat weaker condition, in the sense that it is implied by the droop quota. Therefore, the theorem is still sufficiently general. We restate the theorem to include this condition.

Theorem 3.13. *Suppose $q \geq |b|/2s$. Then if $c \in C$ satisfies the inequality $\mathsf{cfp}(c, b) > |b|/2$, c is a winner.*

Proof. Assume $2sq \geq |b|$, $\mathsf{cfp}(c, b) > |b|$ and that $(b, q, s) \vdash \mathsf{winners}(w)$ is provable. We want to show that any such c is contained in w. There are two cases to consider, corresponding to the rules that allow us to declare a winner.

In the first case, where the sum of the elected candidates and the hopeful candidates is no more than the number of seats, and so together they are declared the winners, the theorem follows easily from the invariant. Suppose there exists a provable state of the count $(b, q, s) \vdash \mathsf{state}(u, a, t, h, e)$ such that $(b, q, s) \vdash \mathsf{winners}(w)$ is obtained by applying the 'hopeful win' rule, as follows.

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

- $|e| + |h| \leq s$
- $w = e \mathbin{;} h$

Then the invariant tells us

$$c \in e \ \vee \ \big(c \in h \ \wedge \ (t(c) + \mathsf{cfp}(c, u) > |b|/2) \ \wedge (\mathsf{ct}(t, h \mathbin{;} e) + |u| \leq |b|)\big).$$

If the first disjunct of the invariant holds, then $c \in e$ and so $c \in w$, since $w = e \mathbin{;} h$. If the second disjunct holds, then $c \in h$ and so $c \in w$, since $w = e \mathbin{;} h$. Therefore c is winner, as required.

In the second case, where there are as many elected candidates as seats and so the elected candidates are declared the winners, proving the theorem is more difficult. Suppose there exists a provable state of the count $(b, q, s) \vdash \mathsf{state}(u, a, t, h, e)$ such that $(b, q, s) \vdash \mathsf{winners}(w)$ is obtained by applied the 'elected win' rule, as follows.

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

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

Then the invariant tells us

$$c \in e \ \vee \ \big(c \in h \ \wedge \ (t(c) + \mathsf{cfp}(c, u) > |b|/2) \ \wedge (\mathsf{ct}(t, h \,;\, e) + |u| \leq |b|)\big).$$

If the first disjunct of the invariant holds, then $c \in e$ and so $c \in w$, since $w = e$. If the second disjunct holds, we must identify a contradiction, since c satisfies the hypothesis of the majority criterion by assumption, and yet is not being elected.

The contradiction comes from the third conjunct of the second disjunct of the invariant. Intuitively, we think of this as saying that it is not the case that too few votes have been counted to elect c, so c must not satisfy the hypothesis of the majority criterion and thus isn't elected. To prove this contradiction involves manipulating inequalities, which is very well-suited to completing in *Coq* but not very enlightening, and so is omitted here. As usual, the complete formal proof may be found in the code. □

3.2.2 Formalisation in *Coq*

We now outline the implementation of the mathematical formalisation in *Coq*. This begins with the auxiliary functions to count first preferences and sum the current tallies.

Implementation 3.14. *The two functions are readily encoded as* recursive functions*, where a function f is recursive if it maps x to an expression in which f occurs. This is done using the* `Fixpoint` *keyword and pattern matching.*

```
Fixpoint count_fp (c: cand) (l: list ballot) : nat :=
  match l with
   [] => 0
   | x::xs => match x with
               [] => count_fp c xs
              | y::ys => if (cand_eq_dec c y) then (count_fp c xs + 1)%nat else count_fp c xs
              end
  end.

Fixpoint cnt_tly (t: cand -> nat) (l: list cand) : nat :=
  match l with
  | [] => 0
  | c::cs => (t c + cnt_tly t cs)%nat
  end.
```

Along with the definitions, we prove some basic lemmas that were found to be necessary in the proof of the invariant. Firstly, we prove that both functions have a homomorphism property.

```
Lemma count_fp_hom (c: cand) (l1 l2: list ballot) :
   count_fp c (l1 ++ l2) = (count_fp c l1 + count_fp c l2)%nat.

Lemma cnt_tly_hom (l1 l2: list cand) (t: cand -> nat) :
   cnt_tly t  (l1 ++ l2) = cnt_tly t  l1 + cnt_tly t l2.
```

We also prove an upper bound for the first preferences for a candidate.

```
Lemma count_fp_bnd (c: cand) (l: list ballot) : count_fp c l <= length l.
```

It is worth remarking that we have used equivalent ways of giving lemmas and theorems,

```
Lemma count_fp_bnd : forall c l, count_fp c l <= length l.
```

Neither form is preferred over the other as they correspond to the same type. We compare the first form to saying 'Let c be a candidate...' and the second to the expression 'For all candidates c...' and use a mix of the two, as is often found in an informal proof.

The mathematical formalisation of the lemma and the theorem must be specialised to our implementation by including reference to states of the count. For the lemma:

```
Lemma stv_inv : forall b q s c, s > 0 ->
  2 * count_fp c b > length b ->
  forall n, Pf b q s n ->
  forall  u a t h e, n = (state (u, a, t, h, e)) ->
  (In c e) \/
    ((In c h) /\
     (2 * (t c + count_fp c u) > length b) /\
     (cnt_tly t (h ++ e) + length u <= length b)).
```

The first hypothesis excludes a degenerate case by requiring a nonzero number of seats, and the second corresponds to the hypothesis of the majority criterion. The third and fourth hypotheses specialise the lemma to our implementation of Simple STV by supposing a correct state of the count. It is tempting to write these two statements in the single hypothesis:

```
  forall u a t h e, Pf b q s (state (u, a, t, h, e))
```

however, this term is not sufficiently general to allow us to perform induction on the type of proofs. Induction must occur over a completely general instance, that is, where all arguments are unconstrained variables. It suffices to pull out the argument:

```
  forall n, Pf b q s n ->
  forall  u a t h e, n = (state (u, a, t, h, e))
```

The remainder of the statement of the lemma is a clear translation of the disjunction in the mathematical formalisation, using the functions defined in Implementation 3.13.

Proof of the invariant proceeds by induction on the type of proofs, which corresponds to implementing the proof outlined in section 3.2.1. In *Coq*, when the keyword `Inductive` is used to define a type T, as in the case of `Pf`, an induction principle called `T_ind`, a recursion principle called `T_rec` are automatically defined. The `induction` tactic generates a subgoal for each possible form of term. In this case, eight subgoals are generated and proof by induction corresponds to stepping through these subgoals, with induction hypotheses added to the local context for each rule. The `induction` tactic generates a subgoal for each possible form of term. In this case, eight subgoals are generated and proof by induction corresponds to stepping through these subgoals, with induction hypotheses added to the local context for each rule.

We now implement the main theorem in *Coq*.

Implementation 3.15. *The main theorem is given as:*

```
Theorem maj: forall b q s w c,
  s > 0 ->
  2 * (s * q) >= length b ->
  2 * count_fp c b > (length b) ->
  Pf b q s (winners w) ->
  In c w.
```

As in the lemma, we assume a nonzero number of seats. The second hypothesis is the condition placed on the quota and the third is the hypothesis of the majority criterion. Again, we specialise to include reference to the type of proofs, requiring evidence that `winners w` *is a provable state of the count.*

We prove this theorem using the `inversion` tactic. Where the induction tactic applies the induction principle for the type in question to generate a subgoal for each constructor, the inversion tactic takes into account the form of hypothesis to which it is applied, generating a subgoal only for the constructors that could have been used to prove something of this form.[6] In this case, inversion on `p : Pf b q s (winners w)` yields two subgoals corresponding to the rules `hw` and `ew`, so the proof may proceed as outlined in section 3.2.1. The complete formal proof is found in the code at line 1631.

Chapter 4

Generic Termination

The vote counting rules as proof rules formulation has many desirable features. The formal specification of the rules is in close correspondence to the natural language specification; a provably correct vote counting program can be automatically extracted from the specification; the extracted program generates an independently-verifiable certificate in the form of a proof of the outcome; and it is done in a natural environment for proving voting system criteria.

The approach may also be applied to different vote counting protocols. This is evidenced in [16], in which both FPTP and Simple STV are formalised as proof rules. In theory, other protocols may be given the same treatment, such as other variants of STV. Since different electoral authorities across Australia use different versions of STV, it is advantageous for an approach to provably correct, independently verifiable electronic vote counting to be readily adaptable.

However, the current method means that each time a new protocol is considered, the formalisation is started essentially from scratch. This was the case with FPTP and Simple STV. While Simple STV and other STV variants have many commonalities, just altering one rule in Simple STV would necessitate updating the type of proofs, updating proofs of any voting system criteria, and fixing the relevant part of the proof of the existence of winners, along with many of the lemmas on which this depends.

The extent of the revisions required to adapt an existing formalisation to a very similar protocol is great, while at the same time, the features in common between the formalisations of two very different protocols – FPTP and Simple STV – are many. Thus, it makes sense to abstract the features in common to all protocols under the vote counting as mathematical proof approach to see if we can establish a generic framework. Results may then be proved of the framework,

which can then by instantiated with specific protocols. Under this approach, we can change the protocol without having to completely re-do a proof. Rather, it is only necessary to re-establish for one or two relevant rules.

In this chapter, we construct a general framework referred to as *generic termination*, where a terminating proof sequence is one ending in a final judgement declaring the winners. We begin by defining this framework, which involves separating out the rules, rather than giving them as a single type, and then defining a termination condition local to the rules.

After establishing the generic termination framework, we demonstrate its adaptability by first applying it to FPTP and then Simple STV. Finally, we extend this proof of concept to a real world vote counting protocol, the version of STV used by The Australian National University Union Incorporated (the Union), given in Appendix A. This protocol uses a common feature of STV that is not used in Simple STV, namely the assignment of *fractional transfer values* to ballots.

4.1 Method

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

Working from this observation, the idea is to make explicit the understanding that certain judgements are intermediate and certain judgements are final, and then define a function that assigns to a non-final judgement the value of the data in the judgement that is always decreasing. We call this function the *measure* of a non-final judgement. Intuitively, if the measure decreases at every rule application, and there is always a rule that can be applied to a non-final judgement, then we can prove termination – that a final judgement is always reached.

The property of the measure that allows us to prove termination is that it takes values in a type with a well-founded order. A well-founded order is defined classically as follows.

Definition 4.1. *Let X be a set with a strict ordering $\prec$ and inverse ordering given by $\succ$. Then $\prec$ is a well-founded order if there is no infinite descending chain $x_1 \succ x_2 \succ \cdots \succ x_n \succ \ldots$.*

An example of a well-founded order is the natural numbers with the 'less than' ordering. Since a well-founded order has no infinite descending chains, a decreasing measure on the domain of a well-founded order must terminate. In a constructive setting, this definition is valid but not useful as it is given in terms of the non-existence of a property. There is a constructive alternative, defined in terms of the positive property of *accessibility*.

Definition 4.2. *Let A be the accessibility predicate, then accessibility is defined inductively by the following rule:*

$$\frac{\forall y \prec x. A(y)}{A(x)}$$

This says that if every $y \prec x$ is accessible, then x is accessible. From this definition, a well-founded order is defined constructively as follows.

Definition 4.3. *Let X be a set with a strict ordering $\prec$. Then $\prec$ is a well-founded order if and only if every element in its domain is accessible.*

The constructive definition implies the classical definition, but not the converse. Since the accessibility relation is defined inductively, there is automatically an induction principle.

Definition 4.4 (Well-founded induction)**.**

$$\frac{\forall x \in X. (\forall y \prec x. P(y)) \Rightarrow P(x)}{\forall x \in X. P(x)}$$

This says that to show a property holds for an arbitrary element $x \in X$, assume that it holds for all y such that $y \prec x$, then show that it holds for x. It may then be concluded that it holds for all $x \in X$. The familiar mathematical induction is just a special case of well-founded induction, where the well-founded order is the relation $<$ on the natural numbers.

For FPTP, the relation $<$ on the natural numbers is a suitable well-founded order. Since it is not a single piece of data that is always decreasing in the case of Simple STV, a more complicated well-founded order is required to compare tuples.

Definition 4.5 (Strict lexicographic order)**.** *Let A, B and C be three sets with strict orderings $\prec_A, \prec_B$ and $\prec_C$. The strict lexicographic order $\prec$ on the cartesian product $A \times B \times C$ is defined by*

$$(a, b, c) \prec (a', b', c')$$

if and only if

$$a \prec_A a' \text{ or } (a = a' \text{ and } b \prec_B b') \text{ or } (a = a' \text{ and } b = b' \text{ and } c \prec_C c').$$

Theorem 4.6. *If $\prec_A, \prec_B$ and $\prec_C$ are well-founded, then the lexicographic order is well-founded.*

Suppose we have a list of rules, R, and a well-defined measure. We formalise two properties which together give a termination condition local to the rules.

Definition 4.7. *Let* dec *be a property of a list of rules R such that* dec(R) *if whenever a rule holds true of two judgements, the value of the measure of the premise is greater than the value of the measure of the conclusion. In other words, whenever a rule is applied the measure decreases.*

Definition 4.8. *Let* app *be a property of a list of rules R such that* app(R) *if for every non-final judgement, there is always a rule that may be applied.*

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

Theorem 4.9. *For any R such that* dec(R) *and* app(R)*, for every judgement j there exists a final judgement and a proof sequence from j to the final judgement via R.*

The proof of this theorem is given in the formalisation in *Coq*, which we now develop.

4.1.1 Formalisation in *Coq*

The formalisation of generic termination is modelled even more closely on a formal deductive system than the first approach to vote counting as mathematical proof. We provide the familiar features of judgements, rules and a notion of proof before formalising the properties and proving the termination theorem.

The general framework is given as a section in *Coq*. It is not given as a separate file; rather, it can be found at the start of each of `FPTP_generic.v`, `STV_generic.v` and `Union_generic.v`

Implementation 4.10. *We begin by defining the type* `Judgement`*, intended to capture a generic notion of a judgement, rather than specifying the forms it may take. Where before there was an implicit notion of a 'final' judgement giving the outcome of the count, we make this explicit by defining it as a property of a judgement. We also specify that this property is decidable – for every judgement there is either a proof that it is final or a proof that it is non-final.*

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

The keywords `Variable` *and* `Hypothesis` *are used so that we may instantiate these types later on. Similarly, we define a generic relation* `wfo` *on a type* `WFO`*, and hypothesise that this relation is well-founded. It uses the well-founded module in the* Coq *standard library.*

```
Variable WFO : Type.
Variable wfo: WFO -> WFO -> Prop.
Hypothesis wfo_wf: well_founded wfo.
```

Next we define the measure function. It maps a non-final judgement to a term of type `WFO`*, on which the well-founded order exists. To specify that the input judgement is non-final, the domain of the function type is a dependent type. The dependent type pairs a judgement with a proof that the judgement is non-final, and is expressed using familiar set-comprehension notation.*

```
Variable m: { j: Judgement | not (final j) } -> WFO.
```

To use this function, we define a type that takes a judgement, along with evidence that it is non-final and packages this into a dependent type. It is called `mk_nfj`*, read as* make non-final judgement.

```
Definition mk_nfj: forall j: Judgement, forall e: not (final j), { j : Judgement | not (final j) }.
```

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

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

Finally we define a type of proofs, this time based on a generic list of rules. As before, this will allow us to produce an independently verifiable certificate in the form of a sequence of rule applications. The type of proofs is given as a dependent inductive type with two constructors, or ways of giving evidence that a judgement has the property of provability. It is parametrised by an initial judgement and a list of rules.

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

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

This establishes the elements of the general framework. We now encode two properties, parametrised by a list of rules, to capture our reasoning from before.

Implementation 4.11. *The properties are encoded as parametrised dependent function types. The first property* `dec`*, read* decrease*, is defined as:*

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

We read this as saying for a rule in the list and a pair of judgements satisfying the rule, with evidence that these judgements are non-final, the well-founded order holds for the measures of the judgements. That is, the measure of the conclusion is accessible from the measure of the premise.

For the second property `app`*, read* application*:*

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

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

Note that although we refer to `dec` *and* `app` *as properties, the codomain is* `Type` *rather than* `Prop`*. This is for the same reason as using the type level existential quantifier and the type level disjunction - if we defined it as* `Prop` *we would lose the evidence and just have knowledge of truth or falsity.*

The main result we want to show is that if these two properties hold for a list of rules, then we have *termination*. In the formalisation, termination corresponds to the existence of a term of the type `Pf a Rules c` where `c` is a final judgement. To this end, we first prove a lemma specifying when a proof sequence may be *extended*. Then we prove an auxiliary result of termination before deducing the desired result as a corollary.

Implementation 4.12. *The lemma suppose the list of rules R satisfies the* `dec` *and* `app` *properties. It says that if there is a proof from a to a non-final judgement b via R, there exists a judgement c such that we can extend to a proof from a to c via R. Furthermore, if c is non-final, then it has measure less than the measure of b. It is readily encoded using the formalisations from before:*

```
Lemma extend:
  forall Rules : list Rule,
    dec Rules ->
    app Rules ->
  forall a b : Judgement, forall eb: not (final b),
    Pf a Rules b ->
  existsT c : Judgement,
    (Pf a Rules c) *
    (forall ec: not (final c), wfo (m (mk_nfj c ec)) (m (mk_nfj b eb))).
```

The proof follows easily from unfolding the definitions of `dec` *and* `app`*. In particular, we instantiate the assumption of* `app` *with the non-final judgement* `b` *to say that there exists a rule holding for* `b` *and* `c`*. This allows us to apply the* `mkp` *constructor to go from a proof of* `b` *from* `a` *to a proof of* `c` *from* `a`*. It remains to show the second conjunct of the conclusion, that if* `c` *is non-final then the measure decreases from judgement* `b` *to* `c`*. This follows easily by specialising the assumption of* `dec` *to the judgements* `b` *and* `c`*, as well as the rule that holds for them.*

We now present the theorem from which we can deduce termination. It says that if a set of rules satisfies the `dec` *and* `app` *properties, then for all judgements*

a and b such that b is non-final and there is a proof of b from a, there exists a judgement c such that c is final and there is a proof of c from a.

```
Theorem termination_aux : forall Rules : list Rule,
    dec Rules ->
    app Rules ->
  forall n: WFO,
  forall a b : Judgement,
  forall eb:  not (final b),
    m (mk_nfj b eb) = n ->
    Pf a Rules b ->
      (existsT c : Judgement, final c * Pf a Rules c).
```

To prove this theorem, we perform well-founded induction *on the measure of* `b`*. The proof is omitted here but provided in the code.*

From this theorem, we define the main result as a corollary. It says that if a set of rules satisfies the dec *and* app *properties, then for every judgement a there exists a final judgement c such that there is a proof from a to c. The corollary is readily encoded:*

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

A proof amounts to first supposing a is final and then supposing a is non-final. If a is final, we apply the constructor `ax`*, and if a is non-final, we apply the theorem* `termination_aux`*. The full proof is included in the code.*

This concludes the general proof termination framework. We will now apply it to three different vote counting protocols. This involves providing a specific notion of judgement as well as a particular well-founded order and set of rules. Then after proving dec and app hold, the final corollary can be applied to these as arguments.

4.2 First past the post

For a first simple instantiation of the vote counting protocol, we consider FPTP as given in Chapter 1. The original *Coq* formalisation of FPTP under vote counting as mathematical proof is given in [16].

The code in `FPTP_generic.v` has three sections:

1. `genericTermination` gives the generic framework.

2. `FPTP` specialises the framework to FPTP.

3. `candidates` defines a list of candidates for an example vote count.

In addition, the code ends by instantiating the function to be extracted, using definitions from the three sections, and with a command for extraction.

Implementation 4.13. *The type of judgements is the same as the type* `Node` *in the original FPTP formalisation.*

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

We specify that a final judgement is of the form `winner w`*, and prove decidability for the property of a judgement being final.*

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

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

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

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

We give the rules as separate types, intended to be as easy to read as standard statements in formal logic.

```
(* Rule 1: if there is an uncounted vote for c, then increment
 c's tally by one and update the list of uncounted votes*)
Definition count (p: FPTP_Judgement) (c: FPTP_Judgement) : Prop :=
  exists u1 t1 u2 t2,
  p = state (u1, t1) /\
     (exists l1 c l2,
     u1 = l1 ++ [c] ++ l2 /\
     u2 = l1 ++ l2 /\
     inc c t1 t2) /\
  c = state (u2, t2).

(* Rule 2: If all votes have been counted and all cands have
 fewer votes than c, then c may be declared the winner *)
Definition declare (p: FPTP_Judgement) (c: FPTP_Judgement) : Prop :=
  exists u t d,
  p = state (u, t) /\
  u = [] /\
  (forall e : cand, t e <= t d) /\
  c = winner d.
```

We then define the list of rules.

```
Definition FPTPR : list FPTP_Rule := [ count; declare ].
```

We observed that under every rule application, either the number of uncounted votes decreases or a final judgement is deduced. Therefore, the relevant well-founded order is the less than relation on the natural numbers. We instantiate the framework accordingly, defining first the type on which the order exists, the order and then proving the order is well-founded.

```
Definition FPTP_WFO : Type := nat.
Definition FPTP_wfo: FPTP_WFO -> FPTP_WFO -> Prop := lt.
Lemma FPTP_wfo_wf: well_founded FPTP_wfo.
```

Accordingly, we define the measure to map a non-final judgement to the length of the list of uncounted votes at that state of the count. Rather than defining it directly, we make an interactive definition by giving the following declaration, and then stepping through the construction of a term of the desired type.

```
Definition FPTP_m : { j: FPTP_Judgement | not (FPTP_final j) } -> nat.
```

With this formalised, it is a matter of proving the two properties.

Implementation 4.14. *We provide the arguments for the* dec *property:*

```
Lemma dec_FPTPR : dec FPTP_Judgement FPTP_final FPTP_WFO FPTP_wfo FPTP_m FPTPR.
```

The proof proceeds considering each rule in turn and is given in the code. We provide the arguments for the app *property:*

```
Lemma app_FPTPR : app FPTP_Judgement FPTP_final FPTPR.
```

The proof proceeds by case analysis and is given in the code.

We can now give an example set of candidates and extract a program that will count votes for them, as we showed before for Simple STV.

Example 4.15. *We provide candidates as for our example of Simple STV.*

```
Inductive cand := Alice | Bob | Claire | Darren.
Definition cand_all := [Alice; Bob; Claire; Darren].
```

and then define the termination function.

```
Definition FPTP_termination :=
  termination (FPTP_Judgement cand) (FPTP_final cand) (final_dec cand)
  FPTP_WFO FPTP_wfo FPTP_wfo_wf (FPTP_m cand) (FPTPR cand) (dec_FPTPR cand)
  (app_FPTPR cand cand_all cand_finite cand_eq_dec cand_inh).
```

We can then extract this function to obtain Haskell code, see `FPTPCode.hs`, with a wrapper for the sake of visualisation, see `FPTPCount.hs`.

4.3 Simple STV

As a second example, Simple STV is implemented in the generic termination framework. The code has three sections:

1. `genericTermination` gives the generic framework.
2. `STV` specialises the framework to Simple STV.
3. `candidates` defines an example list of

The specialisation of the framework to the protocol is explained below.

Implementation 4.16. *The definition of a judgement is the same as the previous formalisation of Simple STV, except with two adjustments – the types corresponding to the tally and elected candidates are now given as dependent types, to hard-code a property of each. A tally is a function paired with evidence that the value of the function is never greater than the quota, and the elected candidates are a list paired with evidence that the length of the list is never greater than the number of seats.*

```
Inductive STV_Judgement :=
   state:                          (** intermediate states **)
       list ballot                 (* uncounted votes *)
     * (cand -> list ballot        (* assignment of counted votes to first pref candidate *)
     * { tally : (cand -> nat) | forall c, tally c <= qu }     (* tally *)
     * (list cand)                 (* hopeful cands still in the running *)
     * { elected: list cand | length elected <= s}    (* elected cands no longer in the running *)
     -> STV_Judgement
  | winners:                       (** final state **)
    list cand -> STV_Judgement.    (* election winners *)
```

In the previous formalisation, these properties where proved as lemmas. Since we are now working in a generic framework, the instantiation itself need not be generic.

A final judgement is defined to be a judgement of the second form,

```
Definition STV_final (a : STV_Judgement) : Prop :=
   exists w, a = winners w.
```

and showing this property is decidable is a routine proof of the following lemma.

```
Lemma final_dec: forall j : STV_Judgement, (STV_final j) + (not (STV_final j)).
```

The generic definition of a rule as a property of a pair of judgements is specialised to Simple STV judgements.

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

The rules may then be defined, with an individual type for each rule. They are expressed differently but correspond to the same rules as before, except for minor adjustments due to the dependent types in the judgement type. We also dispense of the rule corresponding to the start of the count, since the type of proofs allows us to specify a starting state.

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

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

To determine an appropriate well-founded order and measure function, we analyse the rule applications to identify quantities in the judgement type that decrease. This is more complicated than for FPTP because although there is always some quantity in the judgement that decreases, it is not always the same quantity. For example, under the 'count one' rule, the number of uncounted votes decreases and the number of hopeful candidates remains the same, while under the 'transfer least' rule, the number of hopeful candidates decreases and the number of uncounted votes increases.

A strict lexicographic order on triples of natural numbers is used as the well-founded ordering. We define the measure as a function on non-final judgements to $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ by:

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

that is, a triple of the length of the list of hopeful candidates, the length of the list of uncounted votes and the sum of the lengths of the uncounted votes. In two of the five rules concerning non-final judgements, the number of hopeful candidates decreases. Of the remaining rules, the 'count one' and 'empty votes' rules both preserve the number of hopeful candidates but decrease the number of uncounted votes, while the 'transfer votes' rule preserves the number of hopeful candidates

and the number of uncounted votes but decreases the sum of the lengths of the uncounted votes. Therefore, this satisfies the definition of a lexicographic order.

We now implement these definitions in *Coq*.

Implementation 4.17. *The definition of a lexicographic order on a triple of natural numbers, as well as a proof of well-foundedness, comes from [4] and we omit the details here. Using this, we define a product type and lexicographic order on the product type and prove that it is well-founded.*

```
Definition STV_WFO := nat * (nat * nat).

Definition STV_wfo : STV_WFO -> STV_WFO -> Prop := (fun x y : nat * (nat * nat) =>
 lt_npq (mk3 x) (mk3 y)).

Lemma STV_wfo_wf : well_founded STV_wfo.
  unfold STV_wfo.
 apply wf_inverse_image.
 apply wf_lexprod.
Qed.
```

Since the lexicographic order in [4] is defined for ease of proving well-foundedness using pre-existing objects from the Coq *libraires, rather than in terms of its operational behaviour as defined earlier, we prove the following lemma to show the two are equivalent.*

```
Lemma wfo_aux:  forall a b c a' b' c' : nat,
  (lt_npq (mk3 (a, (b, c))) (mk3 (a', (b', c'))) <->
    (a < a' \/
    (a = a' /\ b < b' \/
    (a = a' /\ b = b' /\ c < c')))).
```

Now we may define the measure function as discussed earlier. We must first define a function that takes a list of lists and returns the sum of the lengths of all the lists in the list. We also prove some basic properties about this function for later proofs – namely that it is a homomorphism and that there is an order relation. We include the definition below and omit the lemmas.

```
Fixpoint sum_len {A: Type} (l: list (list A)) : nat := match l with
| [] => 0
| x::xs => (length x + sum_len xs)%nat
end.
```

Then we can provide the measure by constructing a proof term of the following type.

```
Definition STV_m: { j: STV_Judgement | not (STV_final j) } -> STV_WFO.
```

With the generic framework instantiated, it is now a matter of proving the dec and app properties hold of the list of Simple STV rules. The formal proofs may be found in the code; here we provide informal proof sketches.

To show that a rule can always be applied proceeds by case analysis. For an arbitrary non-final judgement $\mathsf{state}(u, a, t, h, e)$, we are required to prove the existence of a rule and a second judgement such the rule holds true of the judgement pair. To prove existence, we must exhibit a witness, that is, explicitly provide a rule and a judgement and then prove that the required properties hold. For example, u is either the empty list or of the form $u \mathbin{;} us$. We use tactics to break it into these two cases. Suppose u is the empty list, then we make a new case distinction – either $|e| + |h| \leq s$ or $|e| + |h| > s$. In the first case, we may apply the 'hopefuls win' rule, and we have enough information about the terms u, a, t, h and e to construct a second judgement so the the rule holds. The case analysis proceeds.

In showing that the measure always decreases, for the sake of modularity and readability, we split the dec property up by proving a series of lemmas for each rule saying that the measure decreases under that rule. This is routine manipulation of the definitions of the rules so we omit the details.

Having proved these two properties, we may again use the termination result proved in the generic framework to define a function and extract a vote counting program producing an outcome and a proof in the form of a proof sequence. The function is given as follows, using the candidates defined in the candidate section.

```
Definition STV_termination :=
  termination (STV_Judgement cand qu s) (STV_final cand qu s) (final_dec cand qu s) STV_WFO
  STV_wfo STV_wfo_wf (STV_m cand qu s) (STV cand qu s) (dec_STV cand qu s)
  (app_STV cand cand_eq_dec qu s).
```

4.4 The *ANU Union* vote counting protocol

The Australian National University Union Incorporated (the Union) uses a protocol based on a variant of STV using *fractional transfer* values. A fractional transfer value is a rational number less than 1 assigned to a candidate's surplus at the stage of transfer. In our version of simple STV, we did not take this into account. The voting procedure for the Union is included as Appendix A.

With fractional transfer, the tally is the sum of the transfer values on the ballots. The formalisation draws on the method of manual counting in which there is a 'pile' of ballots corresponding to each candidate. Throughout the

count, ballots are moved between the piles as candidates are eliminated and their votes are transferred. This image is used in the formalisation. We also think of an extra pile corresponding to a backlog of candidates requiring their votes to be transferred. This accounts for the order of transfer being important - transfers happen in the order candidates were eliminated.

4.4.1 Mathematical formalisation

Let C be a set of candidates. A single ballot is represented by a pair $B = (v, w)$, where the 'vote' $v \in \mathsf{List}(C)$ is a *permutation* of the set of candidates and $w \in \mathbb{Q}$ is the 'weight' of the ballot, also known as the transfer value.

Definition 4.18. *If $b \in \mathsf{List}(B)$ represents the list of ballots cast and $s \in \mathbb{N}$ represents the number of seats available to be filled, then the quota $q \in \mathbb{Q}$ is given by the Droop quota, that is*

$$q = \frac{|b|}{s+1} + 1$$

Definition 4.19. *If $b \in \mathsf{List}(B)$ represents the list of ballots cast and $s \in \mathbb{N}$ represents the number of seats available to be filled, then a judgement takes one of two forms:*

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

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

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

where $w \in \mathsf{List}(C)$ represents the list of winners of the election.

The first judgement corresponds to an intermediate state of the count, while the second judgement corresponds to the final state of the count. We explicitly define the form of a 'final' judgement.

Definition 4.20. *A judgement is said to be* final *if it is of the second form, $(b, s) \vdash \mathsf{winners}(w)$ for some $w \in \mathsf{List}(C)$.*

We consider the lexicographic order on the set of triples of natural numbers. We prove this is a well-founded order in *Coq*.

Definition 4.21. *Let the measure be a function which takes a judgement of the form* $\mathsf{state}(ba, t, p, bl, e, h)$ *and returns* $(|h|, |bl|, |ba|)$.

The rules of the deductive system are given in the usual way, in the form of a premise and a conclusion with side conditions relating the these judgements, and a label to the right. Bullet indentation corresponds to quantification.

Definition 4.22. *There are seven deduction rules.*

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

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

holding for a list of ballots requiring attention, a list of hopeful candidates, a candidate c and a ballot b when b requires attention, and c is the first hopeful candidate on the ballot. Then define the rule:

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

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

read as:

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

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

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

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

read as:

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

Elect *applies when there are no candidates requiring attention and there are hopeful candidates who have reached the quota to be elected. To specify that the lists of hopeful candidates and elected candidates are updated, let* leqe *be the relation*

$$\mathsf{leqe}(k, l, l') \equiv \forall x, x \in k \Rightarrow \mathsf{eqe}(x, l, l')$$

holding for $k, l, l' \in \mathsf{List}(X)$ *and* $x \in X$ *when* l *and* l' *are equal except that* l' *additionally contains all the elements of the list* k*, where* eqe *is as defined earlier.*

Let ordered *be a function ordering a list according to according to a rational-valued function* f *such that if* $f(x) \geq f(y)$, x *is before* y *in the list. Let* map *define a function by* λ *extraction over something. Then define the rule:*

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

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

read as:

> *"If there are no ballots requiring attention, and there are continuing candidates who have reached the quota (but no more than the number of available seats), order these candidates by surplus and declare them elected by moving them from the list of hopefuls to the list of elected candidates. Update the transfer values in the piles of the newly elected candidates, while leaving the other piles unchanged. Add the list of newly elected candidates to the end of the backlog. "*

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

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

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

read as:

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

Hopeful win *declares the winners of the election in the case where the number of elected plus hopeful candidates is no greater than the number of seats. Define the rule:*

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

- $|e| + |h| \leq s$
- $w = e \mathbin{;} h$

read as:

> *"If the number of candidates that are either hopeful or elected is less than or equal to the number of seats available, then scrutiny ceases and all candidates that are either elected or hopeful are declared winners of the election".*

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

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

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

read as:

> *"If the number of elected candidates equals the number of seats available, scrutiny ceases and the elected candidates are declared the winners of the election".*

4.4.2 Formalisation in *Coq*

As a third demonstration of how the generic framework may be readily applied to different vote counting protocols, the mathematical formalisation of the Union protocol is implemented. The implementation of a protocol in the generic termination should now seem routine. The code `Union_generic.vs` has two sections:

1. `genericTermination` gives the generic framework.

2. `unionCount` specialises the framework to the Union protocol.

To determine the measure, again there is not a single quantity in a non-final judgement that decreases on every rule application. However, there is always a quantity that decrease, and the well-founded order is again a strict lexicographic order on a triple of natural numbers. We define the measure as a function on non-final judgements to $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ by:

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

that is, a triple of the length of the list of hopeful candidates, the length of the list of candidates whose votes are to be transferred and the length of the list of ballots requiring attention. In two of the four rules concerning non-final judgements, the number of hopeful candidates decreases. Of the remaining rules, the 'count one' rule preserves the number of hopeful candidates and preserves the length of the backlog but decreases the number of votes requiring attention, while the 'transfer votes' rule preserves the number of hopeful candidates but decreases the length of the backlog. Therefore it satisfies the definition of a lexicographic order.

We now implement these definitions in *Coq*.

Implementation 4.23. *This time, we specify that a ballot is a permutation of the candidates, along with a transfer value, given as a pair type where the first component is a dependent type:*

```
Definition ballot :=  ({v : list cand | Permutation cand_all v} * Q).
```

A judgement is encoded as follows. We use the prefix 'FT', read as 'fractional transfer', throughout.

```
Inductive FT_Judgement :=
   state:                                         (** intermediate states **)
       list ballot                                (* uncounted votes *)
     * (cand -> Q)                                (* tally *)
     * (cand -> list ballot)                      (* pile of ballots for each candidate*)
     * list cand                                  (* backlog of candidates requiring transfer *)
     * {elected: list cand | length  elected <= st} (* elected cands no longer in the running *)
     * list cand                                  (* hopeful candidates still in the running *)
     -> FT_Judgement
  | winners:                                      (** final state **)
    list cand -> FT_Judgement.                    (* election winners *)
```

We define the property of being a final judgement as taking a particular form, and prove that the property is decidable.

```
Definition FT_final (a : FT_Judgement) : Prop :=
   exists w, a = winners w.

Lemma final_dec: forall j : FT_Judgement, (FT_final j) + (not (FT_final j)).
```

We specialise the definition of a rule to the particular type in question.

```
Definition FT_Rule := FT_Judgement -> FT_Judgement -> Prop.
```

The rules are encoded as individual types. We have to define several functions to formalise the rules, which we omit. An example is the 'count' rule

```
Definition count (prem: FT_Judgement) (conc: FT_Judgement) : Prop :=
  exists ba t nt p np bl h e,      (** count the ballots requiring attention **)
  prem = state (ba, t, p, bl, e, h) /\     (* if we are in an intermediate state of the count *)
  [] <> ba /\                              (* and there are ballots requiring attention *)
  (forall c, exists l,         (* and for each candidate c there is a list of ballots *)
     np(c) = p(c) ++ l /\      (* such that the pile for c is updated by adding l to the top *)
     (forall b, In b l <-> fcc ba h c b) /\ (* and a ballot is added to c's pile iff c is fcc *)
     nt(c) = sum (np(c))) /\       (* and the new tally for c updated *)
  conc = state ([], nt, np, bl, e, h).      (* then we proceed with the updates *)
```

The remainder of the rules are written in the same form and included in the code. The list of all rules is defined.

```
Definition FTR : list FT_Rule := [count; transfer; elect; elim; hwin; ewin].
```

The well-founded order is again a strict lexicographic order on a triple of natural numbers. The implementation is the same as before, and the same lemma of well-founded order operational behaviour is proved.

```
Definition FT_WFO := nat * (nat * nat).
Definition FT_wfo : FT_WFO -> FT_WFO -> Prop := (fun x y : nat * (nat * nat) =>
 lt_npq (mk3 x) (mk3 y)).
Lemma FT_wfo_wf : well_founded FT_wfo.
```

We implement the definition of the measure by providing the term, as discussed before, of the following type.

```
Definition FT_m: { j: FT_Judgement | not (FT_final j) } -> FT_WFO.
```

Having specialised the framework, it remains to prove the two properties dec and app hold. The dec property has been proved using the same approach as for Simple STV. This is included in the code, but is routine and not elaborated on here.

The app property is again proved by case analysis. The proof is still in development, due to the time restraints on the project, however there is confidence that it can be completed with more time.

Concluding remarks

The electronic vote counting as mathematical proof approach is a fruitful application of elegant ideas in structural proof theory and type theory. By proving the majority criterion holds for the implementation of STV, we have demonstrated that the method readily accommodates comparison of voting system criteria. By constructing a generic framework, we have made the application of the approach to different vote counting protocols easier.

Further work begins with finishing the final proof in the generic termination implementation of the Union protocol, which is anticipated to be a straightforward task. It would also be interesting to explore other benefits of the modularity of the generic termination framework, for example, by trying to determine conditions local to the rules that ensure certain voting system criteria are satisfied.

Appendix A

The Union constitution, sections 22-23

We include the relevant sections of the constitution on which the protocol in Chapter 4 is based, so that the reader may verify that the protocol formalised is the same as the protocol specified in the legal document. The complete constitution may be found here [18].

22. Counting of Votes

    (1) Provided that the Returning Officer is satisfied there has been no irregularity in the course and conduct of the election, then, immediately after the close of the poll, the Returning Officer or her/his deputy shall open the ballot box containing the voting papers and count the first preference votes.

    (2) Following the count of first preference votes, the Returning Officer may adjourn the count of votes to such time and place as the Returning Officer thinks fit, and may make such further adjournments as she/he feels necessary.

    (3) No member of the Union or employees of the Union shall be engaged in the counting of votes at an election.

    (4) Each candidate shall be entitled to nominate a scrutineer to represent her/him at the counting of votes. Such nominations must be in writing and signed by the candidate.

23. Determination of Election

(1) After the recording to first choices towards candidates and rejection of all informal voting papers, the aggregate number of first choices shall be divided by one more than the number of candidates required to be elected, and the quotient increased by one, disregarding any remainder. This shall be the quota required for election.

(2) Any candidate who has, upon the first choices being counted, a number of such votes equal to or greater than the quota shall be declared elected. Where the number of such votes for a successful candidate does not exceed the quota, the voting papers shall be set aside as being finally dealt with.

(3) Where the number of votes (including transferred votes), obtained by any candidate exceeds the quota, the proportion of votes in excess of the quota shall be transferred to the other candidates not yet declared elected, next in the order of the voters' respective preferences as follows:

  (a) the surplus of the elected candidate shall be divided by the total number of votes obtained (including transferred votes) and the resulting fraction shall be the transfer value;

  (b) the ballot papers shall be marked with the transfer value, and in subsequent transfers shall be marked with the product of the current transfer value with previous transfer values;

  (c) the votes of the elected candidates shall be distributed according to the next preferences on the ballot papers weighted in accordance with the transfer value or product, and shall be added to the votes of the not yet elected candidates.

(4) Where, on the counting of first choices or on any transfer, more than one candidate has a surplus, the largest surplus shall be first dealt with. Where a surplus arises only after a transfer of votes, any surpluses which arose before such transfer shall be first dealt with. Where two or more surpluses are equal, the surplus of the candidate highest on the poll at the last count or transfer shall be first dealt with. If otherwise equal, the Returning Officer shall decide by lot which surplus shall be first dealt with.

(5) Where the number of votes obtained by a candidate is raised up to above the quota by a transfer of votes, such candidate shall be declared elected, and the transfer completed. No vote of any other candidate

shall be transferred to the elected candidate. Where the number of such votes for a successful candidate does not exceed the quota, the voting papers shall be set aside as being finally dealt with.

(6) Where, after first choices have been counted and all surpluses have been transferred, fewer candidates than the number of candidates required to be elected, have obtained a quota, the candidate with the fewest votes (including transfers) shall be excluded, and the votes transferred to other candidates not yet elected, according to the next preferences indicated on the ballot papers. The votes transferred from excluded candidates shall not be further discounted.

(7) Where any surplus arises it shall be dealt with before any other candidate is excluded.

(8) The same process of excluding the candidate with the fewest votes and transferring them to other candidates shall be repeated until all the candidates except the number remaining to be elected, have been excluded. All unexcluded candidates shall then be declared elected.

(9) Where at any time it becomes necessary to exclude a candidate, and two or more candidates have the same number of votes (including transfers), then the Returning Officer shall determine by lot the candidate to be excluded.

(10) (In determining which candidate is next in the order of the voter's preference, any candidates who have been declared elected or have been excluded shall not be considered, and the order of the voter's preference shall be determined as if the names of such candidates had not been on the ballot paper. Where the ballot paper fails to indicate sufficient preferences so as to transfer the vote, it shall be set aside as exhausted.

(11) The Returning Officer may, of her/his own motion, or on the request of any candidate setting out the reasons for the request, recount the voting papers received in connection with any election.

(12) When the Returning Officer has ascertained the result of the election, and after any necessary recount has been completed, the Returning Officer shall declare the poll for the election by announcing, in order of their election, the names of the successful candidates.

Figure A.1: Sections 22 and 23 of the ANU Union Constitution.

Bibliography

[1] Australian Electoral Commission. Media advisory – Western Australian Senate recount. http://www.aec.gov.au/Elections/federal_elections/2013/wa-senate.htm, 2 November 2013.

[2] Australian Electoral Commission. Counting the votes for the senate. http://www.aec.gov.au/voting/counting/senate_count.htm, 6 May 2016.

[3] B. Beckert, R. Gore, C. Schurmann, T. Bormer, and J. Wang. Verifying voting schemes. *Journal of Information Security and Applications*, 19:115–129, 2014.

[4] P. Castéron. Discussion of the *Coq* Theorem Prover. http://permalink.gmane.org/gmane.science.mathematics.logic.coq.club/2555.

[5] P. Castéron and Y. Bertot. *Coq'Art*. Springer, 2004.

[6] A. Chlipala. Coq tactics quick reference. http://adam.chlipala.net/itp/tactic-reference.html.

[7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2015.

[8] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[9] D. Cochran and J. Kiniry. Vótáil: A formally specified and verified ballot counting system for irish pr-stv elections. *Pre-proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software*, 2010.

[10] H. DeYoung and C. Schürmann. *E-Voting and Identity: Third International Conference, VoteID 2011, Tallinn, Estonia, September 28-30, 2011, Revised Selected Papers*, chapter Linear Logical Voting Protocols, pages 53–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] J.-Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[12] R. Goré and T. Meumann. Proving the monotonicity criterion for a plurality vote-counting program as a step towards verified vote-counting. *International Conference on Electronic Voting*, 2014.

[13] R. Goré, D. Pattinson, and V. Teague. Formally verified voter-verifiable Australian Elections. Australian Research Council proposal for funding, 2016.

[14] R. Goré and V. Teague. Submission to parliament of Australia JSCEM inquiry into the 2013 federal election. Submission 114, March 2014.

[15] P. Martin-Löf. An intuitionistic theory of types: Predicative part. *Proceedings of the logic colloquium Bristol, July 1973*, pages 73–118, 1975.

[16] D. Pattinson. Vote counting as mathematical proof. 2015.

[17] J. Slaney. Propositional natural deduction. `http://users.cecs.anu.edu.au/~jks/LogicNotes/Chapter2.html`, 2014.

[18] The ANU Union. Constitution and board minutes. `http://www.anuunion.com.au/the-anu-union/constitution/`.

[19] The Coq Development Team. *Reference Manual.* Number Version 8.5. 2015.

[20] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. `https://homotopytypetheory.org/book`, 2013.

[21] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.