
THE UNIVERSITY *of* EDINBURGH

# A rigorous treatment of Meek's method for Single Transferable Vote with formal proofs of key properties

*Jake Evan Palmer*


Doctor of Philosophy

Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh

2022

# Abstract

This thesis presents a mechanised formalisation of key concepts and properties of Meek's method of Single Transferable Vote (STV). This method is currently in use in a number of local elections in New Zealand, the Royal Statistical Society, and even the Stack Exchange network. Using a formal approach, we show that the iterative solution to the surplus transfer round of Meek's method converges to a unique and valid solution, and connect a functional implementation of its key components to a more abstract and generalised proof.

Along the way, we consider and address issues present in existing pen-and-paper proofs, and discuss a general representation of strict ballots suitable for the proof patterns encountered in our formal development and for the implementation of Meek's method.

We believe that this work pushes the boundaries of interactive theorem proving for the formal verification of voting algorithms, and offers multiple promising avenues for further work on formally verifying the correctness and termination of STV methods in Isabelle/HOL.

# Acknowledgements

I am deeply grateful to my supervisor, Jacques Fleuriot, for accepting me to work on this project and for encouraging me throughout. It was an honour to get to work with such an impressive researcher, lecturer, and mentor. I will always appreciate the friendly atmosphere created in and around the research groups AIAI and AIML which he helps to foster.

I also want to express my appreciation for all the other friendly faces in AIAI and AIML, of which there are too many to mention, and to those closest to me for keeping me company and keeping me sane in what is quite a long, lonely, and difficult process. I wouldn't have been able to do it otherwise (really!).

The research reported in this thesis was supported by funding from the Engineering and Physical Sciences Research Council (EPSRC), which I gratefully acknowledge.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Jake Evan Palmer)

# Table of Contents

# Chapter 1

# Introduction

The 17th of December 2019 marked the 200-year anniversary of Single Transferable Vote (STV), when Thomas Wright Hill, grandfather of STV and member of the Birmingham Society for Literary and Scientific Improvement, had his system used in one of their internal elections. He had this to say of it:

> *"to secure (as nearly as possible) an accurate representation of the whole body ... because experience proves that, owing to imperfect methods of choosing those who are to direct the affairs of a society, the whole sway sometimes gets into the hands of a small party; and is exercised, perhaps unconsciously, in a way that renders many persons indifferent, and alienates others, until all becomes listlessness, decay, and dissolution."*
>
> – Thomas Wright Hill (1819)

A little over 148 years later, on the 21st of February 1968, Brian Lawrence Meek proposed what he regarded, and many others still regard, to be a principled and optimal method for STV. David Hill, the great-great-great grandson of T. W. Hill, wrote the open-source code[1] to be used for New Zealand's implementation of Meek's method of STV, the first implementation of the method we are aware of, and he was a collaborator on the paper which contained the proof of its correctness by Douglas Robert Woodall [41].

Nowadays, the numerous variants of STV, and their various hand- and computer-counting procedures, are used across the world e.g. for the elections in countries such as Scotland, Australia, Ireland and Malta. Meek's method in particular sees use in New Zealand, the Royal Statistical Society, and even the Stack Exchange Network. While the purpose of this thesis is not to argue in favour of this or that system, the timeliness of formal verification research in this area is clear. We hope that this work, along with a few recent ones, will encourage the theorem proving community to contribute to the task, and by its 250th anniversary, perhaps the whole STV family in all their aspects will have been raised to a solid, verified, general foundations and a suite of canonical implementations.

The whole process of elections is, on a sufficiently large scale, a critical system with relatively serious consequences for error. If ballots are lost, misinterpreted, or miscounted on anything beyond a very small scale, the repercussions may be quite dire for the trust of the electorate, for the time and man-power wasted and then needed to rectify it, for the finances and reputation of the election authority, and for the reputation of the elected body and the organisation under its governance (e.g. the national

[1] Available to read on Amazon Kindle since 27th April 2019: "Meek method STV code of Dr David Hill: (New Zealand rules)".

government). Clearly, high confidence in correctness and security should be pursued to the greatest extent possible.

In the current work, the main tool for doing so will be *interactive theorem proving*, a discipline which combines the power and convenience of automated theorem proving, whereby the validity of a statement in a logical language is determined automatically, with the creativity and general applicability of human reasoning.

We present a framework for verifying the fundamental properties of Meek's method for STV, focusing largely on the most complicated component: the surplus transfer round. In doing so, through the substantial development of novel representations and lemmas, we make a significant contribution to the formalisation of this class of voting algorithms. We also provide an implementation of the method and a simple verification of the elimination round. Thus far, existing work on the verification of STV methods has not ventured beyond the verification of the traditional hand-counting methods or the more difficult to hand-count – hence often computer-counted – methods used in Australia and Scotland. The latter do not carry the difficulty that comes with Meek's method's so-called surplus transfer round and its process of iterative convergence (see Section 1.1.1 for an overview). In this work, we formally prove that this process converges to a unique and valid solution, and connect a functional implementation of Meek's method to these results.

## 1.1 An overview of Single Transferable Vote

All STV methods emphasise proportionality and elect to multiple seats. Meek's method (see Section 1.1.1) additionally emphasises the equal treatment of ballots, minimising wasted votes (explained shortly), and does not include any modifications to help hand-counting. STV methods use ranked-ballot voting and are employed for elections with more than one available seat.[2] A voter fills out a ballot by writing 1 next to their most-preferred candidate, 2 to their second-most preferred candidate, and so on. We refer to *a ballot* (formalised in Section 4.2) when we want to emphasise the *ranking* (formalised in Section 4.8), and refer to *a vote* or *votes* or *ballot mass* when we want to emphasise it as an *amount* which is given to a candidate.

A ballot becomes non-transferable (or "exhausted") if it is currently assigned to the lowest-ranked candidate and is selected for potential transfer, e.g. a ballot listing *abc*

[2] STV reduces to Instant Runoff Voting aka Alternative Vote when there is only one seat, which while for good reason is usually considered conceptually distinct from STV is technically subsumed by STV.

that has already been transferred from $a$ to $b$ to $c$.[3] Such votes are no longer assigned to any candidate. The total number of exhausted votes is called the *excess*.

STV proceeds in rounds, with the first one – round 0 – being the initial allocation of votes according to first preferences. Candidates are declared *elected* if their total number of votes reaches (i.e. meets or exceeds) the *quota*. We will write $V_c(st^{(i)})$ to denote the total number of votes candidate $c$ has, given some state $st^{(i)}$ at round $i$. The state may just directly store the current mass of votes each candidate has, or, as we will soon see, may be something more basic like a vector of real numbers associated with each candidate from which the votes they each receive can be derived. We will write $Q$ to denote the quota, which is a specific number usually calculated at the start of the method based on the number of seats $S$ and total number of votes $T$.

In most methods the quota is *static* i.e. it remains the same throughout taking no account of the excess, e.g. $Q = T/(S+1)$. Because of this, if at any round some votes are exhausted, one may not have enough votes "in circulation" – meaning the sum of all votes currently assigned to candidates excluding the excess – for $S$ candidates to meet the quota. This often happens, and when it does one eliminates candidates until $S$ candidates are left and those left are taken as being elected whether they reach the quota or not. Candidates neither eliminated nor elected are called *hopeful*.

When the quota is not static, it is instead given by e.g. $Q(st^{(i)}) = \frac{T-E(st^{(i)})}{S+1}$, where $E(st^{(i)})$ is the excess at round $i$. Here $T - E(st^{(i)})$ is the "votes in circulation" at some given round $i$. In contrast with quotas that do not change, we call this a *dynamic* quota, and hence write $Q(st^{(i)})$ or simply $Q(st)$ for some arbitrary state. As far as we know Meek's method[4] is one of only a couple of STV methods that use a dynamic quota. We will not remark on the relative merits of different quotas here, and simply note that Meek's method uses the dynamic quota with the most commonly used denominator $S+1$.

A generic presentation of STV methods[5] is presented in Algorithm 1, with the representation of the state, the surplus transfer round (Line 6), and the elimination round

[3] Note that lowest-ranked in the context of voting means the candidate given the highest numerical value on the ballot. So if I mark *abc* with ranks 1, 2, and 3, $c$ is lowest-ranked. In other words, one should view ranks as ordinals rather than cardinals.

[4] Warren's method [78], which modifies the state-update method used in Meek's method, also uses a dynamic quota, though we will not deal with it in this thesis.

[5] As far as we know, apart from complicated and very divergent variants like Schulze-STV [70] and CPO-STV [74], the only exception to this general formulation are methods which choose to not elect candidates until they *exceed* the quota, though this has historically been done to get around problems with choosing the quota to be exactly $T/(S+1)$ as this can allow $S+1$ candidates to reach the quota [50]. We are not aware of this being used much or at all in practice, and there are other ways to avoid the problem.

(Line 8) being the major areas of difference between various STV methods. After the transfer round with the state updated $st \mapsto st'$, each candidate $c$ which had surplus votes $V_c(st) > Q(st)$ should have $V_c(st') = Q(st')$, reducing the surplus as desired to zero. If no candidate with surplus exists, the candidate $argmin_{c \in C} V_c(st)$ is eliminated and their votes transferred. If there is a tie it is either broken by looking at previous rounds, later preferences, by evaluating candidates using another method, randomly, or by some less common method. This whole process continues until $S$ (or, as the case may be, $S+1$) candidates are elected.

**Algorithm 1** Generic STV: repeat rounds until $|elected| \geq S$.

**Require:** $S \geq 1 \wedge |C| \geq S$

1: set initial round state $st$ to allocate first preferences
2: $elected \Leftarrow \{c \in C.\ V_c(st) \geq Q(st)\}$
3: $surplus \Leftarrow \Sigma_{c \in elected}(V_c(st) - Q(st))$
4: **while** $|elected| < S$ **do**
5: **if** $surplus > 0$ **then**
6: surplus transfer round [updates $st$]
7: **else**
8: elimination round [updates $st$]
9: **end if**
10: $elected \Leftarrow \{c \in C.\ V_c(st) \geq Q(st)\}$
11: $surplus \Leftarrow \Sigma_{c \in elected}(V_c(st) - Q(st))$
12: **end while**

### 1.1.1 Meek's method

In this introduction we focus on the surplus transfer round of Meek's method, as it is where the formalisation effort of this thesis is largely focused; we do not verify the whole method in this thesis. However, in the short Chapter 2 we will zoom out to give a description of the full method so as to provide the reader with a bigger picture, before zooming back in on the surplus transfer in Chapter 3 for the majority of the remainder of the thesis.

In most variants of STV in use in real elections, the quota is rounded to an integer and whole votes/ballots are transferred. See Figure 1.1 for an illustration of this. In a handful of methods, Meek's method included, *fractional* parts of *all* votes assigned to a

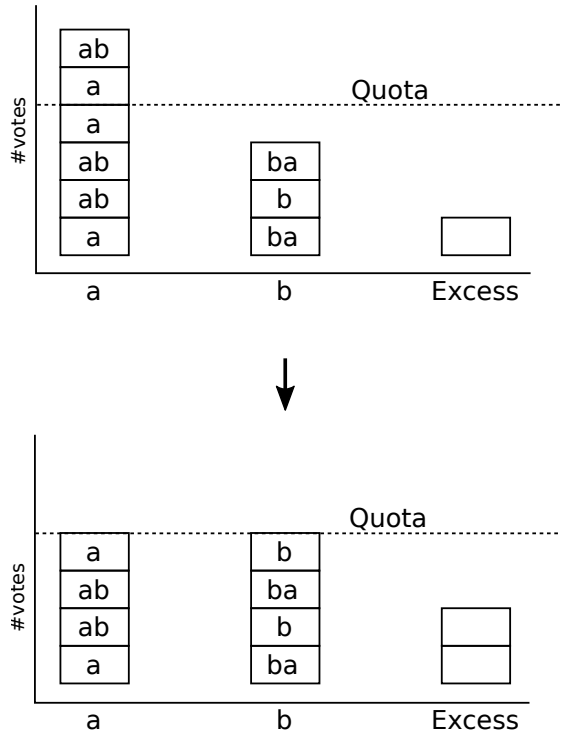



Figure 1.1: Illustration of one surplus transfer for a whole-ballot transfer variant of STV where one transfers the most-recently received (as opposed to e.g. random) ballots. A ballot listing $ab$ becomes $b$ after transfer from $a$, and $a$ becomes $(empty)$.

candidate are transferred according to a candidate's *keep-value* aka their *weight*. This is done for fairness reasons, to avoid arbitrary selection of votes to transfer during surplus transfer rounds. For a standard and minimal implementation of Meek's method,[6] the state is simply[7] the vector of weights, and no other state needs to be stored to determine if candidates meet the quota, are eliminated, and so on. For this reason we can just write $Q(w)$, $V_c(w)$, $E(w)$ for some state $w$. Of course, in order to calculate the actual values of these quantities given some weights one needs the set of ballots, which is left implicit in these expressions.

For weight vectors over the course of an election in Meek's method we could write $w^{(i,j)}$ meaning step $j$ of round $i$, as it almost always takes more than one step to complete one surplus transfer round in this method. However, from here on out we will simply write $w^{(j)}$ as some step $j$ of an implicit round $i$ as done by Hill et al. [41], since it is only the steps of surplus transfer we care about in this thesis.

Meek's method additionally allows transfer to already-elected candidates, which only it and its sibling, Warren's method [78], allow. This is also for fairness reasons, to avoid "skipping over the opinions" of ballots assigned to already-elected candidates. These design decisions also positively impact strategic voting [69] and the number of "wasted votes", which we will not go into here[8] except to say that for our purposes "wasted" is a synonym for "surplus or excess" i.e. wasted either through votes surplus to requirement or by being unassigned to any candidate. See Figure 1.2 for an illustration of one step of the surplus transfer round in Meek's method.

One can calculate the number of votes each candidate has and the amount of excess there is, and hence what the quota is, by dealing with each voter's ballot in turn. If a ballot lists preferences $xyz$ then candidate $x$ will get $w_x$ of that ballot, $y$ will get $w_y(1-w_x)$, and $z$ will get $w_z(1-w_y)(1-w_x)$. This leaves $(1-w_z)(1-w_y)(1-w_x)$ which is exhausted. When the amount that is exhausted – and hence the amount which it contributes to the excess – is not 1, i.e. the ballot is not totally exhausted, we say it is partially exhausted. With this it should be clear why weights are sometimes called keep-values in the literature; when $w_x = 1$ the candidate $x$ keeps all of what they re-

[6] Standard meaning "non-enlisting"', described shortly, and no other optimisations or changes which change the fundamental logical flow. Minimal essentially meaning basic tie-breaking and ordinary elimination.

[7] Almost the only state. Not quite true for the variant we are concerned with, as we will describe shortly when we introduce the necessary context.

[8] "Wasted votes" is a complicated, normative term which does not seem to be dealt with well in the literature, and we are still not sure precisely what it would mean to show that this method minimises wasted votes.

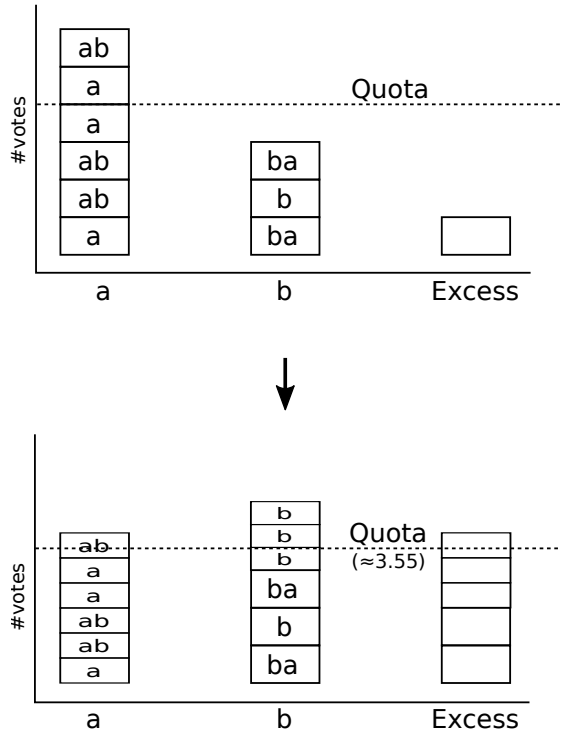



Figure 1.2: Illustration of one surplus transfer step for a Meek's method of STV. Note that the quota decreases due to the increase in excess.

ceive, when $w_x = 0$ they keep none of what they receive, when $w_x = \frac{1}{5}$ they keep one-fifth of what they receive, and so on.

A weight $w_x$ is said to be *valid* if $w_x \in [0,1]$, and a weight vector $w$ is valid if $\forall c \in C.\ w_c \in [0,1]$. We can express[9] the votes that candidate $c$ has given ballots $B$, weights $w$, and a function *frac-of* which returns how much a candidate $c$ gets of some ballot $b$ given some weights $w$ like so:

$$V_c(w) = w_c \Sigma_{b \in B} \textit{frac-of}(b,c,w)$$

where *frac-of* can be written

$$\textit{frac-of}(b,c,w) = \begin{cases} \Pi_{k \in \mathcal{G}(b,c)}(1 - w_k) & \text{if } c \in \mathcal{L}(b) \\ 0 & \text{otherwise} \end{cases}$$

using the function $\mathcal{G}$ which returns the candidates listed greater than a given candidate $c$ on a ballot $b$. See Section 4.2 and the sections following that for further discussion

[9] Similar expressions can be written for non-strict ballots. If a ballot lists three candidates as their most preferred, they each receive a third of the vote. This fair extension is only possible in fractional-vote STV methods. However, we do not cover non-strict ballots in this thesis.

of $\mathcal{L}, \mathcal{G}$. Using the function $\mathcal{L}$, which takes a ballot and returns the set of candidates listed on it, we can write a simple formula for the excess:

$$E(w) = \Sigma_{b \in B} \Pi_{c \in \mathcal{L}(b)} (1 - w_c).$$

One can immediately see from this formula that if every weight is 0, $E(w) = |B| = T$, otherwise (as long as each $w_c \in [0,1]$) we have $E(w) < T$. Neither in Meek nor Hill et al. nor anywhere else we can see are these explicit, closed-form expressions for $V_c(w)$ and $E(w)$ presented, probably because they were not necessary to present and previous authors had a less functional paradigm in mind, but we will use them as our direct implementation as detailed in Section 5.2. In implementations, the excess is typically stored as mutable state and one adds extra exhausted votes to it at each step.

The "equations to be solved at each step" referred to but left unexpanded in Meek and Hill et al. [41] are for each elected candidate $c$, $V_c(w) = Q(w)$. Meek acknowledged [51] that solving this with a dynamic quota, fractional votes, and allowing transfer to the already-elected poses a problem in terms of how to actually do this, citing as an example a non-solution which involves an infinite loop whereby candidates pass votes back and forth between themselves.

This system of equations can be solved analytically (more on complexity issues shortly), and we will demonstrate this with a very small example that we will reuse in Section 5.4.1 in a formal context. Take the set of ballots $B = \{ab, a, abc, bc, ba\}$ and number of seats $S = 2$ and let us use the initial weight vector of all 1s, $w = \mathbf{1}$. We name the ballots according to their list of candidates here as each ballot is unique and for ease of presentation; in general there may be $bc_1, bc_2, \ldots$ and so on. The quota comes out to be $5/3 \approx 1.66$, thus $a$ and $b$ are both elected with $V_a(w) = 3$ and $V_b(w) = 2$. We show the derivation of $V_a(w)$ in full for illustration purposes, placing ballots in square brackets to distinguish them from candidates:

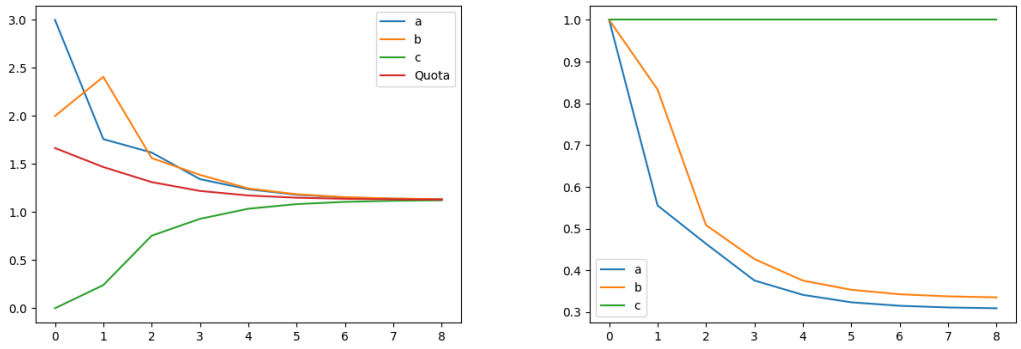
$$
\begin{aligned}
& V_a(w) = w_a \Sigma_{b \in B} \textit{frac-of}(b,a,w) \\
&= \Sigma_{b \in B} \textit{frac-of}(b,a,w) && \text{as } w = \mathbf{1}, \text{ so } w_a = 1 \\
&= \textit{frac-of}([ab],a,w) + \textit{frac-of}([a],a,w) + \\
& \quad \textit{frac-of}([abc],a,w) + \textit{frac-of}([bc],a,w) + \\
& \quad \textit{frac-of}([ba],a,w) && \text{by summing through } B \\
&= \textit{frac-of}([ab],a,w) + \textit{frac-of}([a],a,w) + \\
& \quad \textit{frac-of}([abc],a,w) + \textit{frac-of}([ba],a,w) && \text{as since } a \notin \mathcal{L}([bc]), \textit{frac-of}([bc],a,w) = 0 \\
&= \Pi_{k \in \mathcal{G}([ab],a)}(1-w_k) + \Pi_{k \in \mathcal{G}([a],a)}(1-w_k) + \\
& \quad \Pi_{k \in \mathcal{G}([abc],a)}(1-w_k) + \Pi_{k \in \mathcal{G}([ba],a)}(1-w_k) && \text{by def. of } \textit{frac-of} \\
&= \Pi_{k \in \{\}}(1-w_k) + \Pi_{k \in \{\}}(1-w_k) + \\
& \quad \Pi_{k \in \{\}} + \Pi_{k \in \{b\}}(1-w_k) && \text{expanding } \mathcal{G} \\
&= 1+1+1+(1-w_b) && \text{simplifying products} \\
&= 3 && \text{as } w_b = 1
\end{aligned}
$$

With $w$ as the initial state, we need to solve for some $v$ such that $V_a(v) = Q(v), V_b(v) = Q(v)$, subject to $v_a \in [0,1], v_b \in [0,1], v_c = w_c$, where in this particular case $w_a = w_b = w_c = 1$. If there is no listed hopeful candidate one solution will always be the vector $\mathbf{0}$, though that does not apply here as all three candidates are both listed and hopeful. The vector $\mathbf{0}$ is a "solution" only in a vacuous sense, as the quota and all candidates' votes will be reduced to zero, but this is obviously a problem because that results in all candidates running in the election reaching the quota.

As an aside, proving that the surplus transfer round never converges on $\mathbf{0}$ for any weight vector which would trigger a surplus transfer round is an important result, though it is simply a corollary to proving that the quota always remains positive.

Solving this system reduces to solving $V_a(v) = 4v_a - v_a v_b = Q(v), V_b(v) = 4v_b - 2v_a v_b = Q(v)$ where $Q(v) = (5-(1-v_a)(1-v_b)-(1-v_a)-(1-v_a)(1-v_b)(1-v_c)-(1-v_b)(1-v_c)-(1-v_b)(1-v_a))/3$, which in our simple example with $c$ being hopeful (hence $w_c = 1$) reduces to $Q(v) = (5-2(1-v_a)(2-v_b))/3$. There are two solutions: $v_a = 2, v_b = 4, v_c = 1$ and $v_a = 4/13, v_b = 1/3, v_c = 1$. One can plug both solutions for $v$ into the equations to check that, indeed, $V_a(v) = Q(v)$ and so on for $a, b, c$.

In the first solution we have an invalid weight vector: weights that are greater than




(a) Votes converging from above quota.

(b) Weights converging.

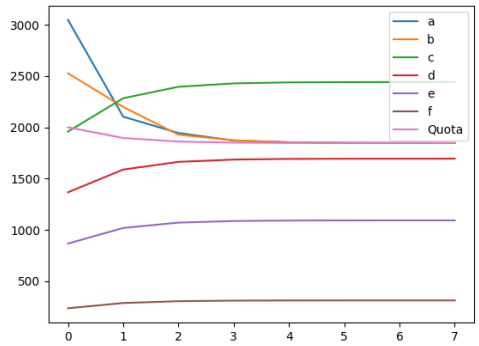
Figure 1.3: $B = \{ab, a, abc, bc, ba\}, S = 2, w = \mathbf{1}$.

1 have candidates take *more* than a whole vote and hence pass on negative vote. This first solution has $V_a(v) = V_b(v) = Q(v) = 0$. The second solution is the one we want, and results in $V_a(v) = V_b(v) = Q(v) = 44/39$. It will turn out that we will always have equal or decreased weight vectors for solutions, here $v_a \leq w_a, v_b \leq w_b$, which is a direct corollary of Hill et al.'s solution to this problem.

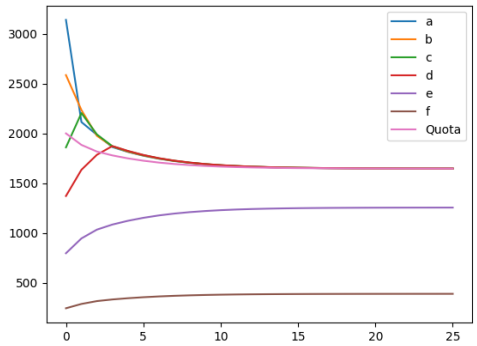
Hill et al.'s computational solution to this problem is elegant, though absolutely did require justification by mathematical proof, provided by Woodall [41]. This iterative solution results in the votes of elected candidates "chasing" the quota.[10] See Figure 1.3 for an illustration of this, which uses the same ballots as in the analytical example. Note in the figure that candidate $c$ happens to receive votes such that it is converging from below; this is not typical outside of small examples like this starting with $w = \mathbf{1}$. Note also that candidate $b$ initially *increases* their vote, and thereafter who has the most votes alternates between $a$ and $b$. See Figure 1.4 for a larger example and an example where candidates can become enlisted into the process as they are newly elected during surplus transfer, i.e. move into the elected set in the middle of the round.

Woodall proved this process converges on a unique and valid solution vector, and as corollaries we can prove the quota remains positive (i.e. is not "chased down to 0") and hence so are the elected candidates' weights and votes, as their votes must remain at-or-above the quota. During the surplus transfer round, we approach such a solution by updating the elected candidates' weights at each step according to the following iteration of the weight vector:

[10] Apart from the rare occasion where all surplus is eventually transferred only to hopeful candidates, whereby the transfer round immediately terminates.




(a) Larger election example.




(b) Larger election w/ enlisting.

Figure 1.4: Randomly generated examples for two larger elections, one without enlisting and one with.

$$w_c^{(i+1)} = w_c^{(i)} \frac{Q(w^{(i)})}{V(w^{(i)})}$$

until the sum of surpluses is less than some constant $\varepsilon$.[11] We discuss our formalisation of these various aspects in Chapter 3. For the duration of the surplus transfer round candidates which were not elected at the start have their weights fixed; for hopeful candidates their weights are fixed at 1, and for eliminated candidates their weights are fixed at 0. In fact, eliminated candidates' weight remain fixed at 0 permanently throughout the method, i.e. remain eliminated.

This is the reason that the weights are not strictly speaking the only state for the "non-enlisting" variant of Meek's method in one specific scenario: at the start of a surplus transfer round we have to store which candidates are initially elected in order to update only their weights. That the weights are the only state is otherwise true, as candidates are eliminated if and only if their weight is zero, and candidates are hopeful if and only if they are not eliminated and do not yet reach the quota (and this implies also they must have weight 1), and candidates are elected if and only if they reach the quota (which implies their weight is in the range $(0, 1]$) *except* in the case where they reach the quota in the middle of the surplus transfer round, in which case their state is updated to elected only at the end of the round.

The computational solution scales better than the analytical approach because there is no general approach to solving the system of equations which would scale to large

[11] In New Zealand, $\varepsilon = 0.0001$. [57]

numbers of candidates and ballots, as we get very large systems of equations of polynomials with inequality constraints on the variables. Both Tarski's method and Collins' cylindrical algebraic decomposition [3] for real closed field problems scale poorly (though are impressive for the problem they are trying to solve), the latter scaling according to a double-exponential.

Apart from scaling better than an analytical approach, as already mentioned the iterative approach also allows the extension whereby one enlists those candidates who reach the quota *during* the surplus transfer round, rather than only addressing them in a new transfer round after the current one has finished, though verifying this other variant is outside the scope of this thesis. The iterative process generally does not reduce the surplus to 0 in finite time, hence the need for the constant $\varepsilon$ to provide some stopping criteria. This also means that in our presentation of the generic algorithm for STV in Algorithm 1, to include Meek's method one would have to replace $surplus > 0$ with $surplus > \varepsilon$.

## 1.2 A brief history of Single Transferable Vote methods

Now that we have some context for the functioning of STV on a technical level, we will briefly review its history and use in practice. This will help situate our work within its historical context and highlight relevant recent developments around the use of STV.

First, we return to Thomas Wright Hill, and his description of his method:

> *"every one who has five votes shall be declared a member of the committee; if there are more than five votes given to any one person, the surplus votes (to be selected by lot) shall be returned to the electors whose names they bear, for the purpose of their making other nominations, and this process shall be repeated till no surplus votes remain, when all the inefficient votes shall be returned to the respective electors, and the same routine shall be gone through a second time, and also a third time if necessary..."*
>
> – Thomas Wright Hill (1819)

One should now be able to see that, given our discussion of STV in the previous section, that the crucial innovation lacking in the above is the *ranked ballot*. Rather than submit ballots ranking candidates from their favourite (placing a 1 next to their name), to their next most preferred candidate (placing a 2), and so on, Hill's system consisted of listing *only* one's most preferred candidate, and then if there were surplus votes[12]

[12] Presumably those most recently given to the candidates were considered the surplus.

these were returned to the voters who cast these ballots, and they were asked to list someone else. It was Carl Andræ who introduced ranking thirty six years later, in 1855, in a proposal for using the method in the Danish Rigsdagen (the name of the national parliament of Denmark at the time).

The second crucial innovation, the elimination of candidates who have no chance of being elected, was only made part of STV when English barrister Sir Thomas Hare published his independently discovered (or invented) version of the system in 1857, in *Machinery of Representation* [36]. Hare is considered by many to be the father of STV. Several other innovations were to follow, such as the modification of the Hare quota of $T/S$ to the Droop quota [19] of $T/(S+1)$.

In 1896 Andrew Inglis Clarke introduced the method to Tasmania, Australia, where, due to various particularities, it has become known as Hare-Clarke STV. The Local Government (Ireland) Act 1919 extended STV to be used in all Irish elections. Between 1915 and 1960 the USA had experiments with STV, though since 1941 the city of Cambridge, Massachusetts is the only place still using it. New Zealand experimented with it locally between 1917 and 1933. All elections in Malta have used it since 1921. In 1922 the Constitution of the Irish Free State mandated proportional representation, and in 1958 and 1968 Ireland rejected two referendums to change to plurality voting (First Past The Post, FPTP) by 51.79% and 60.84% respectively. Canada, similarly to the USA, had various local and regional trials with the system between roughly 1926 and 1971. Australia introduced STV federally in 1948, and more Australian states introduced it internally from 1973 to 1993. India uses STV for the Rajya Sabha upper house, where the electorate are members of each state's legislative assembly. The most recent country to introduce a form of STV is Scotland for local elections in 2007.

STV was almost adopted in 1917 in the United Kingdom (UK), in 211 of the 569 constituencies of the UK, but the proposal was defeated in a clash between the House of Commons and House of Lords. In 2011, a referendum was held about whether to use Alternative Vote (AV, aka Instant Runoff Voting, IRV) for UK elections, but the proposal was also defeated. IRV is effectively a combination of STV with FPTP, with ballots submitted and transferred as in STV but used to elect to single seats (e.g. single-member constituencies), and is thus not a system for proportional representation. It is possible that in the next decade the question will surface again, with the adoption of support for proportional representation by the UK's two largest unions by membership size (at the Unite Policy Conference in Liverpool, 22$^{\text{nd}}$ October 2021, and

at UNISON's National Delegate Conference in 17th June 2022).

Key to our story, however, is the fact that since 2001 in New Zealand all local election authorities have had the option to run their elections using Meek's method for vote counting, and many do so.[13] With some additional development, we hope that it might be possible for the implementation used in New Zealand to be derived from verified code extracted from our formal work. See Section 5.7, Section 5.8, and Section 6.2.1 for further discussion of code extraction.

## 1.3 Contribution

Given this background, we can now summarise the key contributions of this thesis. We present a formal verification of the correctness of the surplus transfer round of Meek's method for STV (Chapters (3, 5)), and in so doing we believe we have contributed a significant step towards formally verifying the whole method. Moreover, given that Meek's method is the most complicated variant of STV in practical use, our work also makes a notable contribution towards reasoning about the correctness of STV methods in general. Noteworthy aspects of our work include the development of a theory of ballots (Chapter 4) suitable for reasoning about strictness using a set-based representation and with ballot induction, rankings, and their properties. We believe this work provides a general framework that others in the Automated Reasoning and Formal Verification community can build on to tackle related and broader challenges involving the formal verification of voting algorithms.

The core of the thesis, where our results are presented and reflected on, is contained in Chapter 3, Chapter 4, and Chapter 5. For more details on specific contributions, hypotheses and evaluation, motivations, and novel ideas and key concepts, see the relevant introductory sections of these chapters: Sections 3.1.1–3.1.5, Sections 4.1.1–4.1.5, and Sections 5.1.1–5.1.4. Discussions of related work can be found in the relevant concluding sections: Section 3.10, Section 4.10, and Section 5.7.

## 1.4 Conclusion

In this chapter we provided essential background for understanding the historical and technical context of the research results we will lay out in the rest of this thesis.

[13] Since 2004, all district health boards in New Zealand also use the method.

We covered the timeliness of formal verification efforts in this area, specifically noting the recent 200$^{\text{th}}$ anniversary of STV and the recent 50$^{\text{th}}$ anniversary of Meek's method. We also discussed the general history of STV and gave a broad technical overview of its functioning in general as well as for Meek's method specifically.

## 1.5 Organisation of the thesis

Having introduced some of the important concepts needed for our work, we now briefly describe how the rest of this thesis is organised. Further information about the structure of the thesis is provided in Section 2.2.2.

In Chapter 2 we provide a whole-method overview of Meek's method. In Chapter 3 we present our abstract verification of the surplus transfer round and elimination round in Meek's method for STV, and then in Chapter 4 we cover a representation of strict ballots alongside a development of sufficient results enough to facilitate connecting an implementation of the component parts of Meek's method in Chapter 5. We conclude with a discussion of our contribution and some avenues for future work in Chapter 6.

# Chapter 2

# Introduction to Meek's method as a whole

In this chapter we give an overview of Meek's method as a whole to help ground the discussion of the surplus transfer round in a broader context. While there is interest in its historical development and subsequent deployment, especially in New Zealand, this will be a purely technical overview along with some helpful intuition. We also provide an introduction to understanding and reading Isabelle/HOL along with some broader context and points on methodology.

## 2.1 Meek's method algorithm

In this section we present an overview of an imperative implementation of Meek's method and a mathematical description equivalent to our implementation in Isabelle/HOL, the latter of which is introduced in Section 5.2.5.

### 2.1.1 Imperative implementation

As we mentioned in the introductory chapter, imperative implementations of Meek's method were all that existed publicly before our work. We will briefly cover a pseudo-code imperative implementation in this section before focusing on a more mathematical description in the next section.

The algorithm presented below (Algorithm 2), is the algorithm presented earlier for generic STV (Algorithm 1) modified to represent Meek's method directly. Where the state was previously given the label *st*, we here give it the label *w* for the weights.

On Line 1 we simply the set the weight vector to be $\mathbf{1}$. This is what we call the "initial allocation of first preferences" in STV in general, whereby the first preferences of each ballot are examined and distributed to the various candidates. If we reword this in terms of weights, we begin by setting the weights so that every candidate keeps all of the votes they receive.

On Lines 2–3 we assign the set of elected candidates as those candidates who reach the quota and set the total surplus to be the sum of the surpluses of elected candidates.

Then on Line 4, if enough candidates are elected, the main loop terminates. Note that the number of candidates elected may be $S$ or $S+1$. The latter only occurs when there is an $(S+1)$-way tie, resolved by random choice or otherwise (not shown here).

Of course, if any single candidate has a surplus the parameter $\varepsilon$ then the surplus transfer round occurs (Line 6), the details of which we exclude here. It is more elegant to present the implementation of this round as a recursive function, which we will do

**Algorithm 2** Meek STV: repeat rounds until $|elected| \geq S$.

**Require:** $S \geq 1 \wedge |C| \geq S$

1: $w \Leftarrow \mathbf{1}$
2: $elected \Leftarrow \{c \in C.\ V_c(w) \geq Q(w)\}$
3: $surplus \Leftarrow \Sigma_{c \in elected}(V_c(w) - Q(w))$
4: **while** $|elected| < S$ **do**
5: **if** $surplus > \varepsilon$ **then**
6: surplus transfer round [updates $w$]
7: **else**
8: $lowest \Leftarrow$ non-eliminated candidates with lowest votes
9: $e \Leftarrow$ picked from $lowest$
10: $w \Leftarrow w\langle e \mapsto 0\rangle$
11: **end if**
12: $elected \Leftarrow \{c \in C.\ V_c(w) \geq Q(w)\}$
13: $surplus \Leftarrow \Sigma_{c \in elected}(V_c(w) - Q(w))$
14: **end while**

in the next section.

Note that the surplus transfer round itself is stopped according to the parameter $\varepsilon$. It is worth imagining what would happen if we set $\varepsilon$ to extremes. If we set $\varepsilon = 0$, then in the majority of cases, i.e. when the transfer is non-trivial, the surplus transfer round will never terminate. It will only terminate in this case if none of the surplus goes to excess (reducing the quota, and triggering further transfer) and none of it goes to already-elected candidates who exist in a loop which will 'pass back' ballot mass to the original candidate.[1] It will continue to tend toward the weight vector which solves the system of equations, but it will never reach it and thus never terminate.

If we set $\varepsilon$ very large, larger than the total number of votes, e.g. $|B| + 1$, then there will never be a surplus transfer round. The candidate with the lowest votes will be repeatedly eliminated until $S$ candidates remain. This turns Meek's method of STV into a crude extension of instant-runoff voting (aka alternative vote). We hypothesise that there is way of choosing $\varepsilon$ such that the final result would be the same for any smaller value, in other words that there is a way of setting the parameter that is non-distorting of the result. We discuss this in future work, Section 6.2.2.

Finally, in lines 8 through to 10, we expand the elimination round:

[1] This latter point is hard to grok but it is not important to understand to follow the thesis.

8: $lowest \Leftarrow$ non-eliminated candidates with lowest votes
9: $e \Leftarrow$ picked from $lowest$
10: $w \Leftarrow w\langle e \mapsto 0\rangle$

This means simply picking one of the candidates whose votes are the lowest and eliminating them, which in Meek's method means setting their weight to 0, i.e. they keep none of what they receive, and pass it all on, as if they had never stood. The angle-bracket notation is chosen to match notation we will use later in our formal development.

If the candidate eliminated had few or no votes, or not many votes which listed further candidates, no new candidates may be elected as a result. In this case, another elimination will take place. In other STV methods repeated elimination requires checking whether $S$ candidates remain after an elimination, but the dynamic quota in Meek's method means that this is unnecessary, as when $S$ candidates remain they will necessarily all meet the quota. For completeness, we note that the set $lowest$ can be computed like so:

$$\{c \in \{k \in C.\ w_k > 0\}.\ \forall c' \in \{k \in C.\ w_k > 0\}.\ V_c(w) \leq V_{c'}(w)\}.$$

All that remains is to specify how to decide which of the lowest-votes candidates to eliminate, i.e. which to pick. The mathematical development only needs to know that such a function exists, but we will still say a little about how one might do it here.

A function $pick(X)$ which picks which candidate to choose from the non-empty set of candidates $X$ might be implemented as a pre-computed set of tuples associating every possible combination of candidates with a choice from that set. For example, two such tuples may be $(\{x,y,z\},y)$ and $(\{z\},z)$; these say that if candidates $x,y,z$ are tied, choose $y$, and if candidate $z$ is "tied", choose $z$ (the only option). Whether this set is computed by uniform random choice, by the alphabetical order of the names candidates, or by some other means is not relevant for us.

However $pick$ is implemented, it can also be re-used for breaking the tie in the case where $S+1$ candidates meet the quota, though real implementations may prefer to use different strategies for this and the case of elimination.

A fully-general $pick$ function signature would allow the use the whole history of states, here weight vectors, to determine which candidate to pick. So, $pick(X,\mathcal{H})$ where $\mathcal{H}$ is the historical list of weight vectors up to the current one. With this, the function could discriminate between the $S+1$ candidates by which was least-recently

elected. An examination of alternative strategies for tie-breaking is outside the scope of this thesis.

### 2.1.2 Functional implementation

A mathematical description of the method comes very naturally once one thinks through how to derive closed-form expressions for the individual components *frac-of*, $V$, $Q$, and $E$. In this section we will focus on the expression implementing Meek's method as a recursive function; for the derivation of the form of individual components see Section 5.2 of the implementation chapter.

The following is a near-complete description of the method, with ballot representation, $\mathcal{L}$, $\mathcal{G}$, and *pick* left undefined. The component functions:

$$\textit{frac-of}(b,c,w) = \begin{cases} \Pi_{k \in \mathcal{G}(b,c)}(1-w_k) & \text{if } c \in \mathcal{L}(b) \\ 0 & \text{otherwise} \end{cases}$$

$$V_c(w) = w_c \Sigma_{b \in B} \textit{frac-of}(b,c,w)$$

$$E(w) = \Sigma_{b \in B} \Pi_{c \in \mathcal{L}(b)}(1-w_c)$$

$$Q(w) = \frac{|B| - E(w)}{S+1}$$

Additional functions (comments on notation to follow):

$$\textit{nonelim}(w) = \{c \in C.\ w_c > 0\}$$

$$\textit{elected}(w) = \{c \in C.\ V_c(w) \geq Q(w)\}$$

$$\textit{lowest}(w) = \{c \in \textit{nonelim}(w).\ \forall c' \in \textit{nonelim}(w).\ V_c(w) \leq V_{c'}(w)\}$$

$$\textit{surplus}(w,X) = \Sigma_{c \in X}(V_c(w) - Q(w))$$

$$\textit{update}(w,\mathcal{E}) = w\langle \mathcal{E} \longmapsto \lambda c.\ w_c \frac{Q(w)}{V_c(w)} \rangle$$

The rounds, the main function, and the additional function for mapping the terminal state to the set of elected candidates:

$$eliminate(w) = w\langle pick(lowest(w)) \mapsto 0\rangle$$

$$s\text{-}transfer(w,\mathcal{E}) = \begin{cases} w & \text{if } surplus(w,\mathcal{E}) < \varepsilon \\ s\text{-}transfer(update(w,\mathcal{E})) & otherwise \end{cases}$$

$$meek(w) = \begin{cases} w & \text{if } |elected(w)| \geq S \\ meek(eliminate(w)) & \text{if } surplus(w, elected(w)) < \varepsilon \\ meek(s\text{-}transfer(w, elected(w))) & \text{otherwise} \end{cases}$$

$$winners(w) = \begin{cases} elected(w) & \text{if } |elected(w)| = S \\ elected(eliminate(w)) & \text{otherwise} \end{cases}$$

The cases in the *meek* function should be viewed sequentially, meaning in the second case there is an implicit $\neg(|elected(w)| \geq S)$ i.e. $|elected(w)| < S$, and so on. Here *s-transfer* stands for the surplus transfer round. The syntax $w\langle c \mapsto x\rangle$ sets the value of the (weight) vector $w$ at the index $c$ to $x$. The syntax $w\langle X \longmapsto f\rangle$ is equivalent to applying $w\langle c \mapsto f(c)\rangle$ for each $c \in X$.

Given some fixed set of candidates $C$, ballots $B$, and stopping-parameter $\varepsilon$, applying $winners(meek(\mathbf{1}))$ will compute the outcome of the election, returning the set of elected candidates.

Most of this does not need remarking upon, given the background we have already provided in earlier discussion. We make only a few comments. First, note that the surplus transfer round *captures* the set of elected candidates at the start of the round. This is what makes this version non-enlisting. If instead of passing $\mathcal{E}$ to the function each time it were instead to replace $\mathcal{E}$ with $elected(w)$ we would have the enlisting variant.

Second, notice that in the final round *winners* it is assumed that in the 'otherwise' case that we will only have $S+1$ candidates which we assume will all have equal votes. Likewise, in *meek* we are assuming that this function will terminate, i.e. will eventually fill seats, among a number of other properties (mentioned at various points throughout the thesis, especially in future work) necessary for this implementation to make sense.

Finally, this version does not ensure there is a final surplus transfer round which reduces the $S$ remaining candidates' votes to be equal to the quota, stopping when the surplus is less than $\varepsilon$. Ensuring this would mean simply changing the conditions in the

case split in the definition of *meek*, so that the surplus transfer round runs whenever there is surplus, even if all seats are filled.

## 2.2 Methodology: approaching Meek's method using interactive theorem proving

In this section we lay out in broad term the methodology we use to approach the formalisation, including our choice of interactive theorem prover, and in turn how this affects the structure of the thesis.

### 2.2.1 Interactive theorem proving and Isabelle/HOL

In this section we provide an overview of interactive theorem proving and Isabelle/HOL sufficient to orient the reader for the rest of the thesis.

#### 2.2.1.1 History and context

Interactive theorem proving combines automated reasoning – whereby theorems in given circumscribed domains are able to be proven completely[2] automatically – with human intervention. That is, a human mathematician uses their existing mathematical experience and knowledge of the world in combination with automated reasoning tools to produce rigorous, computer-checked proof.

All non-trivial theorems require some degree of human intervention, as this thesis more than demonstrates. The history of interactive theorem proving [39] stretches back at least 50 to 60 years, from the initial explosion of interest spurred on by natural deduction provers (e.g. Semi-Automated Mathematics [30]) and satisfiability checkers (SAT) [10], to more recent theorem provers ranging from so-called "formulas as types" provers (Agda [14], Coq [9], NuPRL [2], Lean [18]) to provers inspired by simple type theory and the Logic of Computable Functions (LCF) methodology (HOL-Light [37], Isabelle/HOL [61], HOL4 [71]).

Isabelle [81] is a generic proof assistant that supports a number of specific logics, including ZF set theory and higher order logic (HOL), with the latter being by far the most popular. Isabelle/HOL [61] provides a higher-order logic theorem proving environment based on simple type theory [16, 43]. It is an LCF style [29] theorem

[2] "Completely" to a first approximation, at least; various fine-tuning is often required, though in the form of parameters and not in the form of proof guidance.

prover, and it is well-known that such an approach can ensure the soundness of formal proofs. In practice, LCF style provers only use a small, trusted kernel on top of which all other development is built.

The two major areas which are usually cited as being of interest for the application of theorem provers like Isabelle/HOL are large proof developments where a combination of size and technical speciality makes it labour-intensive or even impossible for human reviewers to verify a pen-and-paper development, and *formal verification* of both software (algorithms, protocols) and hardware (circuit design, hardware instruction semantics like x86). In the "big proof" area are projects like *Flyspeck* [35], which is a particularly famous example owing to the fact that it took around 20 years between initial submission in 1998 without computer formalisation and 2017 with computer formalisation to finally be accepted to a journal without controversy. Our thesis lands squarely in the formal verification of algorithms area, though we make use of general mathematical results e.g. regarding limits which Isabelle/HOL's multivariate analysis library provides standard definitions and results for.

#### 2.2.1.2 Software errors and problematic properties in voting

This discussion raises the question of errors in software implementation ('bugs') and how common they are in voting software used in the world today. The main thing that needs to be emphasised here is that in an unfortunately quite large number of cases, electronic voting in the wild uses completely proprietary software and hardware [11]. This not only makes it difficult to assess how often bugs occur, it also makes it impossible to scrutinise and correct issues and, ultimately, undermines public trust. Developing any open-source, verified software is thus a crucial goal as long as electronic voting continues to see use. Software which is verified and where the results are published openly in peer-reviewed academic journals should be the default. It is worth noting that the most up-to-date Pascal implementation of Meek's method in use in New Zealand was only recently released (on Amazon (!)), in 2019.

Another very significant example for us on this topic is the history of the Gregory method. In 1983, in Australia, the Gregory method was modified to be more fair (by taking part of each ballot instead of those recently transferred), resulting in a major issue with the method which went without any formal comments on the issue until Farrell and McAllister in 2003 [23]; the subsequently invented "weighted inclusive Gregory method" finally puts this method on a more solid foundation and has since been treated formally [25].

While not strictly bugs, discovery of highly desirable properties which voting methods do not satisfy is a common occurrence (see for example the "butterfly effect" in STV [53], which Hill shows Meek's method is actually resistant to [42]). While Arrow's theorem [4] – and more general theorems like Gibbard's theorem [28] – prove that no voting algorithm exists which can satisfy a specific very minimal set of desirable properties, there nevertheless remain properties which considered more important than others, such as proportional representation.[3] Though one must note that STV methods do not strictly rank outcomes: an outcome of STV is a set of elected candidates, whereas ballots list not sets of elected candidates in order of preference but candidates themselves. It is this feature together with over-early elimination which is commonly cited the source of STV's non-monotonicity (a social choice theory concept) [83, 65, 54].[4]

More amusingly but no less serious, it is widely speculated that what triggered the power-of-two issue (4,096 extra votes) in a Belgian election in 2003 was a bit-flip potentially triggered by solar radiation, as it was not explainable by software error [73]. More subtle issues caused by things like this would need individual verifiability (a voter can check that their ballot was counted) and universal verifiability (anyone can check that the result corresponds to the published ballots, clearly requiring also open-source code). See Kremer et al. [48] for further, formal discussion of the concepts.[5]

#### 2.2.1.3 Practical considerations

For a large part of its early history Isabelle/HOL mainly supported a proof method that uses simple natural-deduction style reasoning, which, while suitable for breaking down proofs in a top-down, mechanical fashion, is not particularly human-understandable. Isabelle/HOL today supports Isar (Intelligible semi-automated reasoning) [79], which allows one to combine the type of forward reasoning typical to pen-and-paper mathematics (with natural-sounding keywords like `have x by <proof> moreover have y by <proof> ultimately have z by <proof>`) with automatic tools

[3] Arrow's theorem and other computationally-unaware theorems are still highly relevant, but somewhat less relevant now the field of computational social choice theory [22] is now asking more precise questions, like whether it is computationally feasible to discover a ballot which allows one to vote strategically and not merely whether it is theoretically possible.

[4] I believe that Meek's method provides a potential avenue on how to get around this, by not simply setting candidates' weights to zero but reducing them, though for now this is speculation. I also think that Meek's principles show that there is no principled reason for immediately eliminating candidates, and doing so is merely seen as necessary for moving-along the algorithm.

[5] The authors also note eligibility verifiability: "anyone can check that each vote in the election outcome was cast by a registered voter and there is at most one vote per voter."

and, where appropriate, backward reasoning when proving mathematical statements.

Of particular importance to the practicality of using an interactive theorem prover is *automated proof discovery*, which in Isabelle is principally achieved via invocations of the tool *sledgehammer* [13]. This can pull in hundreds of facts in-scope and invoke external first-order theorem provers like SMT solvers [12] to attempt to automatically prove the goal, and finally reconstruct the proof in Isabelle/HOL. Sometimes, a proof that may look intricate using pen-and-paper can be mechanised by simply breaking it into simpler subgoals and then invoking *sledgehammer* to discharge each goal in turn.

A key feature of Isabelle is the facilitation of scoped context via locales [45, 44, 7]. For an example, see the following section (Section 2.2.1.4). Locales provide a way of organising a collection of fixed constants, or parameters, and assumptions involving these (as well as any other constants in scope) such that if there is any inconsistency in the assumptions then this is contained within the locale and cannot affect Isabelle at the theory-level, thereby preserving soundness. One can also use this feature to investigate algebraic structures such as groups, by fixing the group operator, assuming the group axioms, and then deriving group-theoretical results in that context [47]. Isabelle also has a mechanism that enables an *interpretation* (or model) to be given for a locale by providing concrete values for the parameters of the locale [6]. For example, one could interpret the group theory locale with the operator being integer addition and the identity element being 0, and then proving each of the group axioms given this choice. This support for interpretation will be important to the current work (see Section 5.4).

The suitability of Isabelle/HOL for our task, namely verifying the most complicated and significant part of Meek's method, is demonstrated by our successful application of the tool in this thesis, in particular for catching errors and subtle oversights made by both ourselves and our source text(s), both of which occurred during development. Mistaken assumptions do not necessarily make the assumptions inconsistent, they can just limit the possible models of the abstraction more than intended. So it is necessary to provide a generic model of them at some point to make sure that this has not occurred, which Isabelle also provides facilities for through locales.

#### 2.2.1.4 Reading Isabelle/HOL syntax

We will briefly address enough of Isabelle/HOL's syntax for the reader to be able to comfortably follow the rest of the thesis. Additional comments on the syntax will appear elsewhere in the thesis when necessary. See the Isabelle documentation for further resources [76].

Consider the following lemma statement from our development, ignoring the details for the purposes of this explanation:

```
lemma w_le1_V_ge0:
  assumes c_cand: "c ∈ cands"
      and w_props: "w $ c ≥ 0"
                   "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
  shows "V_for w $ c ≥ 0"
```

Figure 2.1: An example Isabelle/HOL lemma.

Note that ⋀ and ⟹ represent meta-level for-all and meta-level implication respectively. Object-level syntax is the mathematical language one is using Isabelle to speak with, in this case HOL, whereas meta-level syntax is more fundamental to Isabelle as a framework. The object-level `∀x∈X. P x` is in meta-level syntax `⋀x. x ∈ X ⟹ P x`. The object-level repeated implication `P ⟶ (Q ⟶ R)` can be written `(P ∧ Q) ⟶ R`, likewise we can write `P ⟹ (Q ⟹ R)` as `⟦P; Q⟧ ⟹ R`.

The final technical point worth noting along these lines is the implicit meta-level quantification of candidate `c` introduced in the first assumption `c_cand` and referenced in the conclusion, and the fact that the second assumption `w_props` contains a meta-level quantifier which shadows[6] the original candidate `c`.

The keyword **`lemma`** (equivalent to **`theorem`** or **`corollary`**) is (almost always) followed by a name, here `w_le1_V_ge0`, a colon, and then the lemma statement. As long as the lemma remains unproven it is not possible to invoke it elsewhere. One can circumvent this to enable an assume-first-prove-later methodology while one's theories are still in development, but one cannot create an Isabelle *session* [80] or submit any results that are not fully and legitimately proven to the official repository of third-party Isabelle development, the Archive of Formal Proofs [46].

Once proven it can be utilised later on in the same theory it resides in or in any other importing theory. Each assumption of the lemma may, or may not, be given a name. In the lemma above, `c_cand` names a single fact, whereas `w_props` names a pair of facts, which can be accessed by `w_props(2)` for the second fact and `w_props` or `w_props(1,2)` or `w_props(1-2)` for both facts. Lemmas can be stated at the top-level of a theory file, inside a locale context, or inside a typeclass context (described below). See Figure 2.2 for an example of a locale.

[6] A variable in a larger scope is said to be shadowed if a new, narrower scope is introduced which uses the same name; here scopes are determined by binders.

```
locale election_context =
  fixes cands :: "'c set"
    and ballots :: "'b set"
    and ℒ :: "'b ⇒ 'c set"
    and 𝒢 :: "'b ⇒ 'c ⇒ 'c set"
    and listed :: "'c set"
  defines listed_def: "listed ≡ ⋃{ℒ b |b. b ∈ ballots}"
  assumes all_valid: "b ∈ ballots ⟹
                      valid_strict_ballot (ℒ b) (𝒢 b)"
      and all_nonempty: "b ∈ ballots ⟹ ℒ b ≠ {}"
      and ballots_nonempty: "ballots ≠ {}"
      and finite_ballots: "finite ballots"
      and finite_cands: "finite cands"
      and listed_cands: "listed ⊆ cands"
begin

lemma gt_compr_𝒢:
  assumes "b ∈ ballots"
      and "k ∈ ℒ b"
  shows "{c. c >_b k} = 𝒢 b k"
<proof>

end
```

Figure 2.2: An example Isabelle/HOL locale.

The keyword **locale** plus a name introduces a locale context. Here, we are defining an election context, but we will ignore the details until Section 4.9. After the equality, one can "add in" other already-existing locales. Consider how a locale capturing a group may start by adding in a locale defining monoids: **locale** `group = monoid + ...`. After this, one may or may not fix a number of locale parameters, which are then characterised by the assumptions. These parameters can later be instantiated during an interpretation with more concrete values (see Section 5.4 where we use this mechanism for Meek's method).

Assumptions can also be named and are available throughout the locale context and in any extending locales. Here, we use a special keyword **defines** in order to introduce a concept that can be referenced across the assumptions.

A typeclass [31] is like a locale but which is where the constants and functions it fixes must operate on a particular type variable. The typeclass `finite` does not provide any additional constants but requires that any type which instantiate it is able to prove that the set of all values belonging to the type is finite. The typeclass `ord`, e.g. in the expression **fix** `x :: "'a :: ord"`, requires that the type represented by the type variable `'a` provides definitions for the functions `less_eq` ($\leq$) and `less` ($<$),

but makes no requirements on properties they must satisfy. It is used as a foundation for typeclasses like `preorder`, which requires among other things that `less_eq` is transitive.

After a lemma statement Isabelle enters *proof* mode, and it expects a proof to be given of the statement. Here one can invoke a single natural deduction proof step, or a command which in turn invokes an automatic theorem prover, such as the classical first-order theorem-prover `blast`, optionally with additional arguments such as introduction, elimination, or simplification rules one thinks will be needed to prove the result. Alternatively, one can begin an Isabelle/Isar proof (here for the lemma `gt_compr_`$\mathcal{G}$):

```
proof
  show "{c. c >_b k} ⊆ 𝒢 b k"
    using ballot_gt_def by auto
next
  show "𝒢 b k ⊆ {c. c >_b k}"
  proof
    fix x
    assume "x ∈ 𝒢 b k"
    then have "x ∈ ℒ b"
      using assms in_𝒢_listed_in_ℒ by blast
    then have "x >_b k"
      by (simp add: ⟨x ∈ 𝒢 b k⟩ assms(2) ballot_gt_def)
    thus "x ∈ {c. c >_b k}"
      by simp
  qed
qed
```

Figure 2.3: An example Isabelle/Isar proof.

Isar proofs all start with the keyword **`proof`**. If one does not follow **`proof`** with a hyphen then Isabelle will attempt to apply a default rule, and if no such rule applies will fail with an error. There is a default rule here (`equalityI`) which breaks the goal down into two subgoals, proving a subset relationship from two directions. In place of the hyphen one can also pass proof methods like **`proof`** `(rule equalityI)` – which applies the default rule in the case here – or **`proof`** `(rule equalityI, auto)` which would apply the default rule and then attempt to solve all goals using the classical reasoner `auto`.

```
next
  show "𝒢 b k ⊆ {c. c >_b k}"
  proof
    fix x
    assume "x ∈ 𝒢 b k"
    (* ... *)
    thus "x ∈ {c. c >_b k}"
      by simp
  qed
qed
```

Figure 2.4: An partially suppressed example Isabelle/Isar proof.

The keyword **have** introduces a fact and again enters Isabelle into the *proof* mode, where Isabelle expects a proof of the stated fact to be worked out. This can be done by starting a new nested proof block using **proof** or by collecting zero or more facts with **using** and then dispatching the goal using one of the provided proof methods. In this example we use the classical reasoner `blast`, the ordinary single-subgoal simplifier `simp`, and the multi-subgoal classical reasoner `auto`. The keyword **show** (or **thus**, which is shorthand for **then have**) is the same as **have** except for when the fact one is aiming to prove is the current sub-goal. A successful **proof** block is closed with the keyword **qed**. We use *(* ... *)* to indicate suppressed Isabelle proof script (for readability) and <**proof**> to indicate a suppressed Isabelle proof command.

Finally, note that on very rare occasions we omit small fragments of the Isabelle statement of a locale, lemma, or definition. We do this either because it fixes types which are obvious from context, and which including only adds noise, or because they are syntax elements which are not necessary to introduce and which add nothing conceptually to understanding. Isabelle/HOL source is provided in Appendix A, and the full development is publicly available on GitHub [66].

### 2.2.2 Structure of the thesis

Before we begin the first results chapter of the thesis it is necessary to say a few words about about the overall structure of the thesis, now with the necessary background on Meek's method and locales in Isabelle/HOL.

The formalisation begins in Chapter 3 with an abstract representation of the component functions of Meek's method, followed by the transfer round, and eventually the election round. Then in Chapter 4 we cover representation and reasoning about strict ballots, before finally ending in Chapter 5 with a concrete implementation which

models the locales laid out in Chapter 3.

We could have approached this by first implementing Meek's method concretely with top-level definitions, then proved successively more high-level theorems about it, before reaching something like the level of abstraction we tackle the method with in Chapter 3. There are several issues with this latter approach for us, however. First of all, the only existing public implementations of Meek's method before this work, as far as we are aware, was the version laid out in Hill et al. [41] and the version implemented in OpenSTV [64]. Both of these were imperative implementations. As such, we did not have a reference implementation to work with, but we did have a reference proof which implemented abstract, high-level concepts. So this was the natural place to start, during which we naturally "discovered" the elegant recursive formulation of Meek's method we presented in Section 2.1.2.

Secondly, STV is a family of methods, so it was important to us throughout all of the formalisation to ensure that wherever possible we left open the possibility of future generalisation. For example, to cover Warren's method [78], or more traditional hand-counting forms of STV. We thus see our locale hierarchy as a foundation with which future work can build on, in order to produce a hierarchy of locales representing broader families of STV. Even within individual STV methods, Meek included, there are a number of possible implementations. There are a number of choices for tie-breaking in STV generally and a wide range of different quotas in-use. Meek's method has two key variants: one where during the surplus transfer round candidates are "enlisted" into weight updates as soon as they reach the quota, and the standard version that we formalise which fixes the set of elected candidates whose weights will be updated during the round. Thus, by representing the component functions of Meek's method and the rounds at an abstract level first we provide the opportunity of modelling those assumptions using multiple different implementations of Meek's method.

# Chapter 3

# Abstract verification of transfer round convergence in Meek's method

This chapter develops an abstract representation of the surplus transfer round of Meek's method sufficient to prove that it converges on a unique and valid solution vector. We also prove some additional theorems, including basic correctness properties of the elimination round. In the course of discussing this, we will touch on various representational decisions and difficulties along the way.

## 3.1 Overview

In this section we provide an overview of the chapter covering our aims, motivations, hypotheses, and key ideas. We conclude with a summary of the structure of the rest of this chapter.

### 3.1.1 General aims

In order to verify the correctness and termination of Meek's method as a whole, one needs to first deal with the following operations in STV: initial allocation, elimination, the final read-off (transformation of final state into the set of elected candidates), tie-breaking, and surplus transfer. The modularity of STV permits dealing with each of these on its own terms. Initial allocation, elimination, the final read-off, and tie-breaking all terminate trivially, as they all involve a single operation. Surplus transfer, however, is a uniquely involved process and needs to be dealt with using techniques from analysis. This will be the focus of this chapter. We will end with a brief demonstration of applying the results developed for the surplus transfer round to the elimination round.

#### 3.1.1.1 Correctness

A transfer round in STV is correct, that is to say it implements the operation properly, if:

1. None of those elected at the start of the round, i.e. those who meet the quota, have any of their votes transferred.

2. At the end of the transfer round, those elected at the start have no surplus.

3. Those who reached the quota at the start of the round still reach the quota by the end of the round.

Translating these general requirements into the form they take in Meek's method, where the only state that needs to be stored is the weight vector, we get the following, where $C$ is the set of candidates, $w$ is the vector at the start of the round, and $w'$ is the vector at the end of the round:

1. $\forall c \in C.\ V_c(w) < Q(w) \longrightarrow w_c = w'_c$
2. $\forall c \in C.\ V_c(w) \geq Q(w) \longrightarrow V_c(w') \leq Q(w')$
3. $\forall c \in C.\ V_c(w) \geq Q(w) \longrightarrow V_c(w') \geq Q(w')$

Requirements (2) and (3) of course imply $V_c(w') = Q(w')$ for elected candidates. One of Meek's method's unique aspects, however, is that this does not happen, which is why we separate the two here. In general, to account for Meek's method and other methods with only approximate transfer rounds, the requirement (2) becomes instead: at the end of the transfer round, those elected at the start have a *non-distorting* amount of surplus, which can be stated like so:

$$\forall c \in C.\ V_c(w) \geq Q(w) \longrightarrow V_c(w') \leq Q(w') + \varepsilon$$

By non-distorting we mean that if the surplus keeps being reduced, i.e. if $\varepsilon$ is chosen to be smaller, this will not change the *final* outcome at the end of all rounds. Proving non-distortion, though it is an important property, is outside the scope of this thesis. It would require proving that for any election size, there is some $\varepsilon$ which can never distort the outcome by terminating transfer rounds too early. We discuss this avenue for future work in more detail at the end of this chapter.

Requirement (1) is also violated in optimised implementations of Meek's method, but this does not mean the requirement is wrong or overly strict. This common optimisation rolls up successive transfer rounds into a single longer-running round which includes any candidates who newly reach the quota in the transfer process. We call this the 'enlisting' variant, and it can almost certainly be proven to be mathematically equivalent to running successive transfer rounds until a non-transfer round must be done, as noted by Hill [40] in what seems to be the only source which acknowledges this. Proving this equivalence also lies outside the scope of this thesis, though we sketch an argument for equivalence at the end of this chapter as an avenue for future work.

Note, however, that in a floating-point implementation, rounding errors may cause differences between the enlisting and non-enlisting variants. This is as true here as it

is of any two algorithms which even apply numerical operations in a different order despite representing the same equation, and in general we have not found good reason to investigate floating-point considerations. STV voting procedures are not the kind of algorithm which are run many hundreds, thousands, or millions of times a day, and use of precise rational arithmetic should be preferred in implementations, removing any good reason to study floating-point error in these contexts.

It is possible that there are issues with a rational-arithmetic implementation of Meek's method. While we have not seen this in our own experimental implementation even for non-trivial election sizes, it is worth investigating systematically, as it is the only possible source of significant slowdown or scaling issues we can see. We leave it as future work to investigate this, described briefly in Section 6.2.7.

### 3.1.2 Specific objectives

In summary, in this chapter our aims are to verify the correctness of the most complex part of Meek's method: the surplus transfer round. Concretely, this means developing representations in Isabelle/HOL for the relevant data structures, components of the overall algorithm, and abstract representations of the processes those components implement.

There is an existing proof due to Woodall in a paper by Hill et al. [41] that the iterative process involved in the transfer round of Meek's method converges as the number of steps of the process approaches infinity. Thus, the initial work in setting up the locales and the skeleton of the lemmas and theorems in Isabelle/HOL builds on this.

Leveraging this existing work also draws in the kinds of aims that come with any formalisation effort building on existing pen-and-paper mathematics: establishing the sufficiency of the original presentation by ordinary pen-and-paper mathematical standards, discovering any outright errors, and evaluating its appropriateness for translation into a formal system. Concretely evaluating whether such a formalisation is possible, deriving insights, and evaluating the original work comes naturally out of the process of attempting a formalisation, and we will provide commentary on this where relevant throughout this chapter.

### 3.1.3 Motivation

The fundamental motivation we have already discussed, which is that one needs to verify each component of Meek's method in order to verify it as a whole. Thus, all the motivations for verifying voting in general apply here, which we summarised in the opening remarks of Chapter 1.

Beyond this, Woodall's existing proof is a pen-and-paper one that is detached from the associated implementation (in Pascal) and has some insufficiencies regarding gaps in the proof, and it intentionally side-steps potential complications in the two different levels of abstraction underlying one particular part of the argument.

By gaps, we mean the usual gaps which are just the difference between pen-and-paper and formal mathematics, but also more significant gaps that reflect problems in the original presentation. The former kind of 'gap' is occasionally, but not often, interesting to a typical mathematician – that is, a mathematician not interested in automated or interactive theorem proving – though are much more often interesting to those concerned with the philosophy and sociology of mathematics, especially as it relates to the extent to which everyday mathematics is built on pure deductive reason, versus other modes of reason and their intersection: inductive, subconscious-intuitive, abductive, visual-intuitive, and sociological (e.g. appeal to trusted authority).

Replacing these other kinds of arguments with deductive ones can itself be a very insightful process; consider an appeal to intuition when moving from a specific example to a claim that the approach works in the general case, and the insights that can be derived from uncovering the very delicate and subtle problems associated with this being only partially or conditionally the case.

The more problematic kind of gap, reflecting anything from misleading or incomplete presentation to outright error (by omission or otherwise), of which we present a couple of significant ones in this chapter, are clearly important regardless of one's position on formalisation of this sort. For further discussion of the relationship of automated reasoning and interactive theorem proving to mathematical endeavour see Hales on the Kepler Conjecture [34], or even our own work on formalising an axiomatic system for Minkowski spacetime where misplaced geometric intuition is a common issue [68].

### 3.1.4 Hypothesis and evaluation

We claim that the aims laid out in Section 3.1.1 are feasible, and demonstrate this in this chapter by presenting our specific formulation of the theorems necessary to achieve these aims. That we have proven what we claim can be seen by:

- Confirming by inspection that the key definitions, e.g. the definitions for feasible and solution vectors and the weight-update function, are properly implemented.
- Inspecting the statements of the theorems presented in this chapter, to confirm that they indeed prove the correctness and proper termination of the round. We will provide further evidence of this in Chapter 5: that the assumptions can be fulfilled is shown through a concrete example, for which we also provide an analytical solution with an application of our mechanised theorems from this chapter to show that this is the unique solution.
- Further development presented in Chapter 5 showing that the assumptions of the locale are consistent and are applicable to a concrete implementation of the method.

In addition, given that the existing proof due to Woodall is very short, we expected there to be much to say about the relationship between the pen-and-paper and formal proofs, whether this was simply that there are many intuitive ideas which are hard to formalise, or perhaps that there are even gaps and errors in the presentation. Indeed, this turned out to be the case, and we will discuss this in the sections to follow.

We also claim that the motivations laid out in Section 3.1.3 convincingly argue the case for the worthwhile-ness of the aims laid out in Section 3.1.1.

### 3.1.5 Novel concepts and ideas

In carrying out the formalisation of the transfer round, we identified several gaps in the original presentation by Woodall. The representation decisions necessary for this part of the work, which is what we started with, deeply informed the representation and proof development that followed and which we discuss in Chapter 4 regarding ballots and will discuss later in Chapter 5 on the implementation.

Through developing a minimal set of assumptions in order to prove the correctness of this round, and thus transfer rounds generally, it forced us to consider how to characterise what precisely a method has to satisfy in order to be considered an STV. This

also led to our identification of the problem of 'distortion', which we conjecture should be solvable by exhibiting a function for producing a non-distorting stopping-parameter ε for any given election size (see Section 6.2.2).

We have generalised the original proof of Woodall to any fractional quota, $\frac{T-E(w)}{S+c_1}+c_2$, where $c_1 \in \mathbb{N}_+, c_2 \in \mathbb{R}$. See section 1.1 for a discussion introducing these constants and terms. We do not see any way to extend this to integral quotas where e.g. there is a floor function enclosing the expression.

The main reasons for this generalisation are that there are a wide variety of quotas in use across the spectrum of different STV methods in use today, and because we believe existing justification for the use of this quota in Meek's method was not very strong. Proving Meek's method's correctness for as wide a range of quotas as possible then gives us the opportunity to say that this form and range of quotas is necessary for Meek's method to function as intended, and it is clearly desirable to choose the smallest (where $c_1 = 1, c_2 = 0$) because any larger quota could leave candidates unelected.[1] Also, it is interesting to see precisely which results constrain the allowable quotas; we note in particular that the Hare quota where $c_1 = 0$ – at least for the proof approach used by ourselves and Hill et al. [41] – has to be abandoned from the allowable quotas in order to prove the key convergence result.

All of this together provides increased confidence in the correctness of Meek's method as a whole, which has not seen any formalisation effort before, except insofar as others have formalised certain classes of STV, and while all STV methods share a common basic structure these existing efforts nevertheless do not extend to Meek's method.

### 3.1.6 Structure of this chapter

In Section 3.2 we provide a description of our approach to representing the transfer round. In Section 3.3 we present an overview of the constants and assumptions of the locale representing the components of the transfer round. In Section 3.4 we introduce definitions for working with weight vectors in the context of the transfer round and characterise a context for a non-trivial transfer round. Section 3.5 presents a high-level view of the proof that all vectors remain 'feasible' (defined in Section 3.4) for all steps of the round along with a proof that the process converges. Section 3.6 discusses the proof of convergence to a unique solution vector, thus proving the correctness of the

[1] To see this, consider the case where $S+1$ candidates reach the quota exactly, in the case where $c_1 = 1, c_2 = 0$. Clearly, any larger quota would leave them all unelected.

round. Finally, in Section 3.9, Section 3.10, and Section 3.11 we respectively discuss the size of the formalisation, related work, and make some closing remarks.

## 3.2 General approach

The development is split up into two locales: one representing *the components* of Meek's method generally (see Section 3.3), and one representing an abstract surplus transfer round (see Section 3.4). The first locale simply fixes what one might call the necessary "election parameters", such as the number of ballots, number of seats, parameters of the quota, and the set of candidates. It does not make any assumptions about whether we are in an elimination round, a terminal state, or a transfer round. We will avoid dwelling on this first locale and quickly move to the second locale on surplus transfer which extends the first; references back to results proven within this first locale will be made as needed. Further discussion of representation decisions are reserved for when they appear.

Throughout both locales the set of ballots is not directly represented, as we have no need to do so in order to reason about the kinds of properties relevant to the key theorems. This follows Woodall in Hill et al. [41] where $V_c(w)$ is written, even though if one were to pass all relevant quantities to functions we would need the ballots to calculate the votes as in $V_c(B,w)$. Nor is any individual ballot ever referenced formally, though there is one informal reference in Hill et al. on "inspecting each ballot" towards the end of the argument in Woodall's proof justifying a particular step, which we will come back to later in the chapter.

## 3.3 Meek's method locale and additional set-up

In this section we present first an overview of the locale that characterises the component functions of Meek's method generally and abstractly, and then cover some additional definitions and their relationship to the transfer round that are essential for proving convergence in general and which largely follow the concepts introduced by Woodall in Hill et al. [41]. We must first take a brief detour to introduce among other things the harpoon notation, `w ⌊ c, r`.

### 3.3.1 Some helpful notation for vectors

In order to ease working with vectors, three additional functions are helpful. Most of the results needed to work with these definitions are simple consequences of concepts and lemmas from Isabelle's standard library and its multivariate analysis session.

Our definitions not only make reading and writing proof scripts in Isabelle/HOL easier and more compact, but also help with the suggestion of new lemmas and facts during proof. This is due to the reduced visual noise and clearer argument order, and also because they enable symmetries to be more easily noticed as a result of their intuitively logical and compact, symbolic style.

```
definition dec1 ::
  "(real, 'a) vec ⇒ 'a::finite ⇒ real ⇒ (real, 'a) vec" where
  "v ⌊ x,r ≡ χy. if x = y then v $ x - r else v $ y"
```

Figure 3.1: Definition for reducing an element of a weight vector with convenient notation.

This function decreases the `x` element of a vector `v` by `r`, or increases it if `r < 0`. This eases the parsing of facts and goals with several such decreases in one expression, occasionally repeated on the same weight vector.

The vector type `(real, 'a::finite) vec` uses a finite index type (here the type variable `'a`) to access real numbers; note the unintuitive order where index type is in the second position. The dollar in `v $ x` is simply the syntax used for accessing element `x` of vector `v` in Isabelle/HOL, and `(χy. f y)` is a vector with value `f y` at index `y`.

```
definition vec_upd ::
  "('val, 'arg::finite) vec ⇒ 'arg ⇒ 'val ⇒ ('val, 'arg) vec"
   where
  "v<x ↦ val> ≡ vec_lambda ((vec_nth v)(x := val))"
```

Figure 3.2: Notation for updating the value of one element of a weight vector.

This is just a vector update function. It maps the value of vector `v` at index `x` to `val`. The additional notation:

```
definition repl_all ::
  "('b, 'a) vec ⇒ 'a::finite set ⇒ ('b, 'a) vec ⇒ ('b, 'a) vec"
   where
  "v<X↦ v'> ≡ χx. if x ∈ X then v' $ x else v $ x"
```

Figure 3.3: Definition for updating a set of elements of a weight vector to the associated values of a new vector.

maps the values of a vector $v$ to the corresponding values of $v'$ at all the elements in $X$. For all three of these we prove various lemmas, such as:

```
lemma replace_effectively_UNIV [simp]:
  assumes notin_eq: "⋀x. x ∉ X⟹ v' $ x = v $ x"
  shows "v<X↦ v'> = v'"

lemma replace_insert_dec:
  fixes v :: "(real, 'a::finite) vec"
  assumes "x ∉ X"
  shows "v<insert x X↦ v'> = v<X↦ v'>⌊x,(v $ x - v' $ x)"
```

Figure 3.4: Example lemmas proving properties of the vector update functions.

many of which can be handed over to the simplifier for automatic application.

### 3.3.2 Component functions of Meek's method locale

The first locale will be presented in parts, beginning with its parameters:

```
locale abstract_meek_carrier =
  fixes V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec"
    and Q_for :: "(real, 'c) vec ⇒ real"
    and c1 :: nat
    and c2 :: real
    and E_for :: "(real, 'c) vec ⇒ real"
    and num_ballots :: "nat"
    and seats :: "nat"
    and cands :: "'c set"
```

Figure 3.5: The locale characterising Meek's method generally for a specific election (ballots left implicit).

The naming convention `*_for` indicates that a function takes any weight vector (perhaps with additional arguments) and produces some result. The already-introduced mathematical notation $V$, $Q$, $E$, $T$, and $S$ (see Section 1.1.1 and Chapter 2) become

`V_for`, `Q_for`, `E_for`, `num_ballots`, `seats` in Isabelle. In general, we believe that our adopted syntax for the formalisation aids proof readability.

Though we do not represent ballots explicitly, it is important to note that we are still fixing constants representing the votes given some weights $V(w)$, the quota given some weights $Q(w)$, and the excess given some weights $E(w)$. See Section 5.4.2.8 for implementation details; at this abstract level, it is sufficient to reason from axiomatic characterisations of these functions.

As `V_for w $ c` returns a definite value for the votes of candidate $c$ given weights $w$, the set of ballots is in a sense an implicit additional argument of each function. Even if we did represent the ballots explicitly as an argument to the locales, we would still only be able to speak in terms of *bounds* because we do not have a concrete set of ballots such as $\{abc, cab, ac, b, cba\}$.

The function `V_for` takes a vector of real numbers indexed by values of a finite type `'c` and returns a vector of the same type, denoted by `(real, 'c) vec`. In particular, the input is a vector of weights and the output is a vector of votes. The polymorphic type `'c` is that of candidates, and it must be finite due to the constraints of Isabelle's `vec` type constructor, which underneath is merely a wrapper for functions over finite types. This is convenient to work with as it ties into Isabelle's multivariate analysis library, providing most of what we will need regarding limits and the like.

The parameters `c1` and `c2` will be used to constrain the valid forms of the quota `Q_for` and thus aid our generalisation of Woodall's proof. The quota also takes a weight vector and returns the quota for that vector as a real number. Similarly for the excess `E_for`, which is the sum total of all partially or wholly exhausted votes (see Section 1.1.1 and Chapter 2 for a reminder about these functions).

Finally, the three election parameters – `num_ballots`, `seats`, and `cands` – are necessary to characterise the form of the quota and state the assumptions needed about each of the three `*_for` functions. There is an important decision here regarding representation, where we have chosen to include the set of candidates running in the election, `cands`, as a parameter of the locale. Hence the `*_carrier` suffix of the locale naming. We had originally not done this, and instead used the whole type as the set of candidates, the set of which is accessible in Isabelle/HOL using the overloaded constant for the universal set `UNIV :: 'c set`. We will have more to say about this in Section 5.4.2 on interpretation in the next chapter, but suffice to say this initial approach was problematic, even if it did make the statement of assumptions, lemmas, and facts less cluttered with assumptions and case splits about `cands`.

#### 3.3.2.1 `V_for` assumptions

Following this are three assumptions sufficient to characterise `V_for` in terms of both strict and non-strict bounds for all except specific kinds of sums (for which we will introduce an additional assumption later):

```
assumes
    V_change: "⟦c ∈ cands; w $ c ≠ 0⟧ ⟹
                V_for (w ⌊ c,r) $ c =
                  V_for w $ c * (1 - r / w $ c)"
    and V_winc: "⟦c ∈ cands; c' ∈ cands; c' ≠ c;
                  w $ c' ≥ 0; r ≥ 0;
                  ⋀k. k ∈ cands ⟹ w $ k ≤ 1⟧ ⟹
                 V_for (w ⌊ c,r) $ c' ≥ V_for w $ c'"
    and self_winc: "⟦c ∈ cands; w $ c ≥ 0;
                     ⋀c. c ∈ cands ⟹ w $ c ≤ 1;
                     r ≤ 0⟧ ⟹
                    V_for w $ c ≤ V_for (w ⌊ c,r) $ c
```

Figure 3.6: Locale assumptions characterising `V_for`.

These characterise

- what happens to candidate with non-zero weight when their weight is changed,[2]
- what happens to a candidate's vote when *another, distinct* candidate's weight is changed (subject to partial validity requirements on the weights), and
- what happens to a candidate's vote when a candidate's weight is weakly increased,[3] even if their weight is 0 i.e. they are eliminated (also subject to partial validity).

The last assumption is not present in Woodall. While these assumptions repeatedly refer to `w ⌊ c,r`, as they say nothing about the value of `r` this means nothing more than `c`'s weight changes. It is important to continue to notice when `r` is not constrained. The first two assumptions `V_change` and `V_winc` characterise the vast majority of what one will want to say about `V_for`, but `V_change` does not cover the case of the candidate being eliminated, which `self_winc` exists to allow.

[2]The reader may notice that a candidate with non-zero weight is just an eliminated candidate. This is true, but as we are characterising the functions and not yet locating ourselves within a concrete transfer round, it is confusing to refer to candidate status like "eliminated" or "elected". We do so when it is convenient and hopefully unlikely to confuse, but it is good to keep this in mind. The method's elegant encoding of status in weights does make presenting properties of functions on weights confusing outside of the context of actual rounds.

[3]Of course, this does not happen in real rounds.

We had initially tried to generalise cases such as "if the weight is weakly decreased" to "if the weight is changed and this results in the votes weakly decreasing", but this turned out to be a problematic attempt at generalisation which caused numerous issues on the interpretation side, so what we present here ends up being close to the original presentation of the assumptions by Woodall.

The assumption `V_change` gives the formula for a weight change as in Woodall's first assumption "$V_c(w)$ decreases *in exact proportion* [emphasis ours] to the decrease in $w_c$", the explicit formula which they give later in their paper without derivation. Unlike Woodall, we do provide a derivation, which we describe later when discussing the interpretation of the locale (see Section 5.4.2).

Finally, `self_winc` allows one to conclude that if `c` has weight 0, their votes must weakly increase (from 0) upon increasing their weight (we name this lemma `elim_winc`). Eliminated candidates in the actual algorithm never have their weights further decreased, so this is sufficiently general to cover all the cases of changes to weights that affect `V_for`. Although we name this assumption `elim_winc`, note that we are simply saying what happens to a function when a component of a vector equal to 0 changes in a particular way, and that this represents "an eliminated candidate" will only truly make sense later in the context of a concrete round.

It is clear when thinking about the ballots why `self_winc` is true: there are either some ballots which the candidate is listed on which are not currently all assigned to other candidates, or there are not; in the former case they will receive some votes upon the weight increase, and in the latter case they will remain at 0. Less obviously, one way in which we can use `self_winc` via `elim_winc` is by using a trick where we decrease a candidate's weight to 0 and then increase it again (note non-candidates' weights are covered in Section 3.3.3):

```
1  lemma w_le1_V_ge0:
2    assumes c_cand: "c ∈ cands"
3        and w0: "w $ c ≥ 0"
4        and w1: "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
5    shows "V_for w $ c ≥ 0"
6  proof -
7    have "V_for (w<c ↦ 0>) $ c = 0"
8      by (simp add: c_cand eliminated_no_votes)
9    then have "0 ≤ V_for (w<c ↦ 0> ⌊c,(- w $ c)) $ c"
10     using elim_winc [of c "w<c ↦ 0>" "- w $ c"] by (simp add:
       c_cand vec_upd_def w0 w1)
11   also have "... = V_for w $ c"
12     <proof>
13   finally show ?thesis .
14 qed
```

Figure 3.7: Lemma for the non-negativity of candidate votes, whose proofs demonstrates the weight change trick.

This proof states that, given eliminated candidates have no votes (line 7) and increasing eliminated candidates' weights weakly increases their votes (lines 9–11), it must be the case that candidates that are not eliminated have non-negative votes (line 13).

In the premises of the locale assumptions we consistently make minimal assumptions about the weight vector necessary for the conclusion to follow. We do this largely because it is genuinely necessary at some points during the proofs. This is particularly the case because we do not know that weights remain valid for all steps of the surplus transfer round until the first major theorem is proven. As we sometimes have to make locale assumptions more general by weakening assumptions about weights, we decided to make locale assumptions maximally general, for both consistency and future-proofing. That is, as opposed to steadily generalising locale assumptions as needed; there is no repeated fiddling to be done if we maximise generality straight away.

This approach does make it marginally more difficult to prove some results on the implementation side in order to prove it models these assumptions, but the difficulty increase is not significant. Intuition is only sacrificed until the conclusion of the first major theorem, after which we can return to thinking only in terms of valid weights.

#### 3.3.2.2 `E_for` assumptions

We now introduce the three assumptions regarding the excess specifically:

```
    and E_winc: "⟦c ∈ cands; r ≥ 0;
                  ⋀k. k ∈ cands ⟹ w $ k ≤ 1⟧ ⟹
                 E_for (w ⌊c,r⌋) ≥ E_for w"
    and E_lower: "⟦⋀c. c ∈ cands ⟹ w $ c ≥ 0;
                   ⋀c. c ∈ cands ⟹ w $ c ≤ 1⟧ ⟹
                  E_for w ≥ num_ballots * (∏c∈cands. 1 - w $ c)"
    and E_upper: "⟦c ∈ cands; ⋀c. c ∈ cands ⟹ w $ c ≥ 0;
                   ⋀c. c ∈ cands ⟹ w $ c ≤ 1;
                   ⋀k. k ∈ cands ⟹ w $ c ≤ w $ k⟧ ⟹
                  E_for w ≤ num_ballots * (1 - w $ c)"
```

Figure 3.8: Locale assumptions characterising `E_for`.

The first assumption simply says that when one weakly increases a candidate's weight, the excess weakly increases. This is subject to some partial validity requirements as usual, which we will not continue to remark upon. This is provable on the implementation side even if some weights are negative; in general, we make minimally strong assumptions to ease proof. Additionally, as long as one starts with valid weights, we will later eventually prove that weights do only ever (weakly) decrease, and never increase, so the constraint on `r` is still sufficiently general for our purposes; the same applies to the other assumptions.

The second and third assumptions require a little more justification. We know little about the codomain of the excess `E_for` from just `E_winc` (and the invariant, described below). We thus additionally assume lower and upper bounds for the codomain of `E_for` under certain conditions. This will later help us put bounds on both the quota and the votes in circulation. Neither of these assumptions appear in Woodall's proof.

The lowest possible value, characterised in `E_lower`, occurs when every candidate is listed on every ballot, in which case the amount of excess each ballot contributes is the same, and the total excess is found simply by multiplying by `num_ballots`. Note that this is 0 only when all candidates are hopeful i.e. all weights $= 1$.

The maximum possible excess, characterised in `E_upper`, is found by finding the candidate with the (equal-)lowest weight, `c`, and supposing that every ballot only lists `c`, and in which case every ballot contributes `1 - w $ c`. Note this is again only 0 when all candidates are hopeful, i.e. when `w $ c = 1`, and is potentially as high as `num_ballots` if `w $ c = 0`, i.e. if `c` is eliminated, though this only actually occurs if every listed candidate is eliminated. Crucially, this allows us to show that the excess never *exceeds* `num_ballots`.

#### 3.3.2.3 The votes invariant

We will now deal with a single assumption that requires a non-trivial amount of effort to prove on the interpretation side (see Section 5.5.3) but which is necessary to assume for both generality and elegance in this locale:

```
    and votes_invariant:
      "num_ballots = (Σk∈cands. V_for w $ k) + E_for w"
```

Figure 3.9: Locale assumption stating the votes invariant.

Woodall provides five self-evident assumptions about the properties of the functions `V_for`, `E_for`, and `Q_for` when one *weakly decreases* i.e. decreases or leaves the same the weight of a single candidate:

> We shall make extensive use of the following facts which are obvious [...], and in which we use the term 'increases' and 'decreases' in the weak sense (that is, both terms correctly describe a number that does not change): if one component $w_j$ of $w$ is decreased whilst all the other components remain unchanged, then:
>
> 1. $V_j(w)$ *decreases*, in exact proportion to the decrease in $w_j$;
> 2. each $V_k(w)(k \neq j)$ *increases*;
> 3. the sum of the votes for all the 'elected' candidates *decreases* by an amount $v \geq 0$ (since the contribution from each ballot paper decreases);
> 4. the excess vote *increases*, by at most $v$;
> 5. the quota *decreases*, by at most $v/(s+1)$.

We only assume properties (1), (2) and (4) (stated as `V_change`, `V_winc`, and `E_winc` in our locale – see Figure 3.6 and Figure 3.8) and derive the rest by introducing our votes invariant.

Note our version of (1) requires weights be non-zero. This may appear to be less general, as Hill et al. do not seem to make that assumption, but that is because they are stating properties which hold for elected candidates, who they already assume have valid, positive weights. So in fact our assumption is stronger, as it covers all candidates and also applies for a broader range of weights. In general, handling weights and weight validity leading up to the first theorem is much more subtle than it appears in Hill et al.

The invariant itself is perhaps a little surprising: it says that the number of ballots is equal to the sum of votes in circulation plus the excess, given *any* weight vector.

That is, even invalid ones. Unlike with the other assumptions, we do not do this out of necessity for later proof or consistency of presentation, but because it simply falls very easily out of the design of Meek's method that one needs to make no assumption about weights.

The fact that the invariant can be easily stated while making no mention of a particular step of the surplus transfer round will be useful when we come to interpret the locale "within itself" after introducing the second theorem in Section 3.6.1.

One could rephrase the assumption to say that the sum of votes in circulation plus the excess is equal for any two weight vectors, but then one has the problem of proving that this constant is equal to `num_ballots`, and there seems to be nothing to gain from this additional complication.

#### 3.3.2.4 Remaining assumptions

We conclude the presentation of the locale by summarising the purpose of the remaining assumptions:

```
    and quota_form:
      "Q_for =
       (λw. (num_ballots - E_for w) / (seats + c1) + c2) ∧
       c2 ≥ 0 ∨
       Q_for =
       (λw. real_of_int ⌊(num_ballots - E_for w) /
            (seats + c1)⌋ + c2) ∧
       c2 > 0"
    and num_ballots_gt0: "num_ballots > 0"
    and seats_gt0: "seats > 0"
    and noncand_no_V_change: "⟦c ∈ cands; c' ∉ cands⟧ ⟹
                              V_for (w ⌊ c',r) $ c =
                                V_for w $ c"
    and noncand_no_Q_change: "c ∉ cands ⟹
                              Q_for (w ⌊ c,r) = Q_for w"
```

Figure 3.10: Remaining locale assumptions about the quota, number of ballots, seats, and non-candidates.

The first fixes the form of the general quota to either fractional or integral, with any additive constants we like, provided that in the former case $c_2 \geq 0$ and in the latter case $c_2 > 0$. These constraints are necessary to ensure the quota always remains positive, because if it does not then most of the proofs cannot get off the ground. Although Meek's method in practice always uses the fractional variant with $c_1 = 1 \wedge c_2 = 0$, it is an easy generalisation to make. We prove as much as we can without restricting

the form of the quota, but eventually do have to settle on the fractional quota with a constraint on $c_1$.

The remaining assumptions characterise a non-trivial election – `num_ballots_gt0` and `seats_gt0` – and require changes to non-candidates' weights to not affect candidates' votes nor the quota.

### 3.3.3 Assumptions about non-candidates

We generalise `noncand_no_V_change` and `noncand_no_Q_change` to cover arbitrary changes to weight vectors. These kind of "non-candidate" assumptions are necessary due to our use of a carrier set, `cands`, as opposed to using the whole type for the set of candidates. Lemmas like the following which leverage these assumptions are important as it allows us to conclude $V_c(w) = V_c(w')$ if all candidate weights are equal:

```
lemma eq_cand_weights_V_eq:
  assumes "c ∈ cands"
      and cands_eq: "⋀c. c ∈ cands ⟹ w $ c = w' $ c"
  shows "V_for w $ c = V_for w' $ c"
proof -
  have "V_for w $ c = V_for (w<X⟼ w'>) $ c"
    if "X ⊆ UNIV - cands" for X
    using finite that
  proof (induct X rule: finite_induct)
    case empty
    then show ?case
      by simp
  next
    case IH: (insert k X)
    (* ... *)
    finally show ?case
      by simp
  qed
  (* ... *)
  finally show ?thesis .
qed
```

Figure 3.11: A generalisation of the fact that non-candidates' weights do not affect any candidates' votes.

We have shown the cut-down version of the above proof to demonstrate a very common induction technique in our development using an easy-to-understand lemma. We induct on a (necessarily finite) subset $X$ of candidates whose weights are updated from an initial vector $w$ to a final vector $w'$. Usually we take $X \subseteq cands$; the induction pattern is exactly the same, only the set is different. The typical way of finishing off lemmas

of these sort is to take a fact one has like $P(V_c(w < cands \longmapsto w' >))$ and show this is equivalent to $P(V_c(w'))$ as long as $c \in cands$, using lemmas derived from the likes of `eq_cand_weights_V_eq`.

## 3.4 Transfer round locale

In this section, we characterise a non-trivial transfer round according to Meek's method and prove correctness and proper termination

### 3.4.1 Feasible and solution vectors

There are two final concepts to introduce before we get to the surplus transfer round locale. Following Woodall, we introduce the notion of `feasible` vectors and `solution` vectors:[4]

```
definition feasible :: "(real, 'c::finite) vec ⇒ (real, 'c) vec ⇒
    real ⇒ 'c set ⇒ 'c set ⇒ bool" where
  "feasible w V Q cands elected ≡
       ∀c∈cands. w $ c ≥ 0 ∧ w $ c ≤ 1 ∧ (c ∈ elected ⟶ V $ c ≥ Q)"

definition solution :: "(real, 'c::finite) vec ⇒ (real, 'c) vec ⇒
    real ⇒ 'c set ⇒ 'c set ⇒ bool" where
  "solution w V Q cands elected ≡
       ∀c∈cands. w $ c ≥ 0 ∧ w $ c ≤ 1 ∧ (c ∈ elected ⟶ V $ c = Q)"
```

Figure 3.12: Definitions for feasible and solution vectors.

In other words, a vector is feasible if all of its weights are valid, and the candidates which are supposed to be elected reach the quota. We presume the term "feasible" is used by Hill et al. because a wide class of such weight vectors may occur in the course of a transfer round, depending on the specific set of ballots involved. Similarly for solution vectors, where the elected candidates' votes are equal to the quota. Note that we quantify over candidates, which is necessary with the `cands` carrier set approach. The following additional definitions are also useful:

[4]These definitions occur outside the locale in order to be reused in later theory files at the top-level, which is why they take the sets `cands` and `elected` as arguments.

```
definition feasible_given :: "(real, 'c::finite) vec ⇒ (real, '
    c::finite) vec ⇒ (real, 'c) vec ⇒ real ⇒ 'c set ⇒ 'c set ⇒
    bool" where
  "feasible_given w0 w V Q cands elected ≡
       feasible w V Q cands elected ∧
       (∀c∈cands. c ∉ elected ⟶ w $ c = w0 $ c)"

definition solution_given :: "(real, 'c::finite) vec ⇒ (real, '
    c::finite) vec ⇒ (real, 'c) vec ⇒ real ⇒ 'c set ⇒ 'c set ⇒
    bool" where
  "solution_given w0 w V Q cands elected ≡
       solution w V Q cands elected ∧
       (∀c∈cands. c ∉ elected ⟶ w $ c = w0 $ c)"
```

Figure 3.13: Definitions for feasible and solution vectors given some initial weight vector which must remain unchanged.

These latter two definitions explicitly require that a feasible (respectively solution) vector's weights for non-elected (i.e. hopeful and eliminated) candidates is the same as some other vector `w0`, so-named because this will always be the initial vector at the start of the transfer round in our usage. This is helpful when obtaining an arbitrary solution vector in Theorem 2, for example (see Section 3.6).

These all state a little more than Woodall's definitions as they explicitly require that hopeful candidates' weights are also valid, though Woodall likely implicitly considered that hopeful candidates initially have valid weights and as they are never updated will continue to do so. So we are simply making this more explicit.

### 3.4.2 Transfer round locale definition

We introduce the surplus transfer round in two parts. First, we introduce the locale that fixes the set of elected candidates and the weights at step 0 of the surplus transfer round. Inside this locale context, we introduce numerous definitions and abbreviations, including a definition characterising the weight vector at each subsequent step. Second, we use these definitions and abbreviations to state the various assumptions characterising the surplus transfer round.

#### 3.4.2.1 Transfer round locale definition: first part and additional definitions

The initial head of the transfer round locale, containing just the additional fixed constants and one additional assumption:

```
locale meektransfer_fixes_carrier = abstract_meek_carrier +
  fixes transfer_weights :: "(real, 'c) vec"
    and elected0 :: "'c set"
  assumes elected_cands: "elected0 ⊆ cands"
```

Figure 3.14: Transfer round locale constants, with one assumption on $elected0$.

The constant `elected0` should be read as "the set of candidates elected at step 0". The `transfer_weights` vector is the initial weight vector, which should comply with this set of elected candidates (and indeed we will force this to be the case shortly). We can here introduce one basic assumption that does not require additional notation, which is the simple fact that those elected must be candidates.

One may wonder why we do not *define* `elected0` as the set of candidates who reach the quota given the initial weights. It is, as with several of the decisions regarding representation, to do with Theorem 2. We need to be able to interpret this locale with the same set of elected candidates but a *potentially* different initial weight vector. When we finally get there, these subtle representation decisions will pay off in a comparatively brief discussion of Theorem 2.

We can now leverage our vector update notation to tersely describe what it means to update a weight vector once:

```
abbreviation update_one where
  "update_one w ≡
       w<elected0⟼ (χ c. w $ c * Q_for w / V_for w $ c)>"
```

Figure 3.15: One step of the transfer round defined within the transfer round locale.

Note that non-elected weights, including non-candidates' weights, are made to remain the same implicitly. We could set `undefined` for non-candidates' weights to be maximally agnostic about them, but we do not see a benefit in doing so, as in any case because Isabelle requires functions to be total we have to choose something, and it does make things easier to suppose they remain the same. We now simply define the sequence of weight vectors by repeated applications of single-updates using the power function:

```
definition w_at :: "nat ⇒ (real, 'c) vec" where
  "w_at i ≡ (update_one^^i) transfer_weights"

lemma w_upd:
  assumes "c ∈ elected0"
  shows "w_at (Suc i) $ c = w_at i $ c * Q_at i / V_at i $ c"

lemma w_no_upd:
  assumes "c ∉ elected0"
  shows "w_at (Suc i) $ c = w_at i $ c"

abbreviation V_at :: "nat ⇒ (real, 'c) vec" where
  "V_at i ≡ V_for (w_at i)"

abbreviation feasible_for :: "(real, 'c::finite) vec ⇒ bool" where
  "feasible_for w ≡ feasible w (V_for w) (Q_for w) cands elected0"

abbreviation feasible_at :: "nat ⇒ bool" where
  "feasible_at i ≡ feasible_for (w_at i)"
```

Figure 3.16: Definitions capturing the state of the weight vector and votes at all steps of the transfer round, plus two lemmas to illustrate this is equivalent to the usual presentation.

For reference, the following hold for the function `(^^)`: `f^^0 x = x`, `f^^(Suc i) x = f (f^^i x)`. We also introduce additional abbreviations not shown here such as `Q_at`, `solution_at`, and so on. We abbreviate `feasible w (V_for w) (Q_for w) elected0` as `feasible_for w`. Similarly, we introduce `solution_for`. In prior revisions of these locales, the sequence `w_at` was a locale parameter and was characterised directly by how it should update, but given that the weight vector over all steps is uniquely determined by the initial weight vector, there is not much sense in this. The only thing having `w_at` as a locale parameter allows one to do over a definition is to say nothing about non-candidates.

#### 3.4.2.2 Transfer round locale definition: main part

We can now fully begin our treatment of the transfer round:

```
locale meektransfer_carrier = meektransfer_fixes_carrier _ _ _ _ _
   _ _ V_for
  for V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec" +
  assumes elected_nonempty: "elected0 ≠ {}"
      and seats_not_exceeded: "card elected0 ≤ seats"
      and feasible0: "feasible_at 0"
      and E_cont: "isCont E_for w"
      and V_cont: "isCont V_for w"
      and nonelected_nonnegative:
        "⟦c ∈ cands; c ∉ elected0⟧ ⟹ V_at 0 $ c ≥ 0"
      and vote_sum_ge1
        "⟦⋀c. c ∈ cands ⟹ w $ c ≥ 0;
          ⋀c. c ∈ cands ⟹ w $ c ≤ 1⟧ ⟹
         (Σc∈cands. V_for ((w_at 0)<elected0 ⟼ w>) $ c) ≥ 1"
```

Figure 3.17: The transfer round locale with all remaining assumptions.

We assume the continuity of `V_for` and `E_for`, which will be needed for proving our theorems involving limits. Clearly based on our discussion of Meek's method in Section 1.1 these are simply compositions of continuous functions, and we indeed prove their continuity in Section 5.4.2. We have proven that as long as the quota is fractional, the lemma `Q_cont` stated `isCont Q_for w` is true. All of the assumptions of this locale are about characterising the specific, valid and non-trivial surplus transfer round whose properties we want to prove. It is valid because:

1. The weights are initially valid and the elected candidates at least reach the quota: `feasible0`. This forces compliance between the constants `elected0` and `transfer_weights`.
2. Non-elected (either hopeful or eliminated) candidates have non-negative votes: `nonelected_nonnegative`. We need this so that we conclude *all* candidates start with non-negative votes.
3. The seats are not over-filled: `seats_not_exceeded`.

and we are in a non-trivial surplus transfer round because, aside from the non-triviality assumptions of the prior locale:

1. At least one candidate is elected: `elected_nonempty`.

2. No matter how one changes the elected candidates weights at step 0, as long as they are still valid then the sum over all candidates' votes must still be at least 1: `vote_sum_ge1`.

The last of these is intended to capture Woodall's *non-triviality condition*, where they state "there is at least one ballot paper that contains the name of a 'hopeful' candidate [$h$] in its list of preferences". If this is the case then we have $V_h(w) \geq 1$ regardless of what happens to the elected candidates' weights in $w$ as long as they are valid, and elected candidates' weights are the only candidates whose weights change. This ensures non-triviality because if there is no such hopeful candidate, the votes *would* chase the quota down to 0, and in that case rather than go through this iterative process, one should just immediately choose to elect randomly from the remaining hopeful candidates to fill the seats (an undesirable turn of events in practice, of course).

Without access to ballot *structure*, we cannot make a simple assumption equivalent to "there is at least one hopeful candidate listed on at least one ballot". If we assume something like $\exists c \in \mathit{cands}.\ w_c = 1$, we still do not know if they are listed on any ballot. If we abandon our abstraction away from a set of ballots and assume[5] $\exists c \in \mathit{cands}.\ \exists b \in \mathit{ballots}.\ c \in \mathcal{L}(b) \wedge w_c = 1$, without access to the concrete definition of the votes function and ballot structure we cannot use this to conclude that this candidate will always receive some votes. But one of the two major points of using locales for this in the first place, aside from abstracting from ballots and focusing purely on the bounds, is flexibility of specific implementation.

Note that these assumptions do not actually exclude all trivial cases. There is the case where $|\mathit{elected}0| = |\mathit{cands}|$ and the method is clearly finished. And there is the case where $\forall c \in \mathit{elected}0.\ V_c(w^{(0)}) = Q(w^{(0)})$. In the latter case the surplus transfer round would not occur, similarly if the sum of surpluses were less than the stopping-parameter $\varepsilon$. The reason we do not exclude these situations is that these cases are subsumed within the analysis without special treatment (e.g. no case splits), and the intuitive context for this locale is in any case not quite *"given this is a surplus transfer round..."* but rather *"if one were to carry on as if we were in a surplus transfer round without a stopping-criterion..."*, and in the latter of these two trivial cases the weights still "converge", immediately on the initial weight vector. We can now begin proving

[5] Note here we already have to introduce $\mathcal{L}$, and so we are only the step of adding $\mathcal{G}$ away from being at the lower-level of representation except without the concrete function definitions; this would add a level of complexity and syntactic noise that is quite unnecessary, as all facts pertaining simply to bounds are as far as we can tell statable without this. One slightly convoluted assumption does not undermine this basic fact.

the two major theorems.

## 3.5 Theorem 1: feasible convergence on a solution

**Theorem 1.** *The assumptions characterising the votes, excess, and quota, along with the equation for updating weights, constructs a sequence of feasible vectors that converges to a solution vector.*

Proving this proceeds in two parts. First, one proves:

```
theorem all_feasible: "feasible_at i"
```

Figure 3.18: Theorem: the weight vector is feasible at every step of the transfer round.

Then, one proves that `w_inf`, the limit of the sequence of vectors $(\lambda i.\ w^{(i)})$ exists and is a solution vector:

```
definition w_inf :: "(real, 'c) vec" where
  "w_inf ≡ χc. (THE l. (λi. w_at i $ c)⟶l)"

theorem w_inf_solution:
  assumes gfq: "∀w. Q_for w =
                (num_ballots - E_for w) / (seats + c1) + c2"
  shows "solution_for w_inf"
```

Figure 3.19: The limit vector of the transfer round is a solution.

In the above, the operator `THE` is a non-computational aspect of Isabelle/HOL which selects the unique value that satisfies the enclosed predicate. The `The` operator is the Hilbert choice operator for values which uniquely satisfy the predicate passed and `THE` merely provides the ability to bind a variable for convenience.

### 3.5.1 Theorem 1.1: all steps of the process are feasible

The first part proceeds in three steps:

1. Show the quota is positive when the weight vector is feasible.
2. Prove general statements on how the votes and quota change given decreases to weights.

3. Using these general statements, induct on the step and hence show (in ordinary mathematical notation for brevity) $V_c(w^{(i+1)}) \geq V_c(w') = V_c(w^{(i)})w_c^{(i+1)}/w_c^{(i)} = Q(w^{(i)}) \geq Q(w^{(i+1)})$ as per Woodall, where $w'$ is $w^{(i)}$ with $c$'s weight replaced by $w_c^{(i+1)}$.

Formalising the first step is not trivial, and we had to prove the following weaker lemma:

```
lemma feasible_at_Q_pos:
  assumes feas: "∀j. j ≤ i ⟶ feasible_at j"
  shows "Q_at i > 0"
```

Figure 3.20: Lemma showing that the quota remains positive as long as all prior steps of the transfer round produced feasible vectors.

which says that at any given step $i$ if the weight vector is feasible at this and at *every prior step* $j$ then the quota must be positive. For this we have to first prove, among other things, that:

1. No matter what happens to the elected candidates weights, the sum of all votes is always at least 1, using `votes_sum_ge1`.
2. The quota is nonnegative for feasible vectors, using `E_upper`.
3. For any sequence of weight vectors, if at some step i and all prior steps the weights were feasible and updated in the usual way, then the excess never exceeds `num_ballots - 1`.

It is important here that the excess is always less than or equal to some value less than the number of ballots, as when taking the limit we still need `E_at w_inf < num_ballots`.

The general proofs about decreasing weight vectors are also quite involved, and are also used in several places in Theorem 2. This is partly because for Theorem 1, Woodall has these statements for the special case of $w^{(i)}$, $w^{(i+1)}$, and $w'$, but then in Theorem 2 uses similar vectors with the same relationships to construct a solution vector "by Theorem 1". In formalisation, one can tackle such things by proving each case separately or by proving a general lemma which cover both cases, which is what we have done.

Our two general proofs of a little over 200 lines replace Woodall's "By (2), (1), (6) and (5)". The one which substitutes for $V_c(w^{(i+1)}) \geq V_c(w')$, proven in the prior locale, is:

```
lemma many_wdec_V_winc:
  assumes "c ∈ cands"
      and cands_le: "⋀c. c ∈ cands ⟹ w_le $ c ≤ w $ c"
      and c_no_change: "w $ c = w_le $ c"
      and w_ge0: "⋀c. c ∈ cands ⟹ w_le $ c ≥ 0"
      and V_nonpos_eq: "⋀c. ⟦c ∈ cands; V_for w $ c ≤ 0⟧
                            ⟹ w_le $ c = w $ c"
      and w_props: "⋀k. k ∈ cands-{c} ⟹ w_le $ k ≥ 0"
                   "⋀k. k ∈ cands ⟹ w $ k ≤ 1"
  shows "V_for w $ c ≤ V_for w_le $ c"
```

Figure 3.21: Lemma showing that if all candidates' weights are weakly decreased *except* candidate $c$'s, which remains the same, $c$'s votes must increase.

The assumption `V_nonpos_eq` is capturing the fact that we need every candidates' votes to be positive to be able to say anything about the effects of decreasing their weight (due to e.g. the premises of the locale assumptions), so we require that any candidates whose votes are not positive have their weights left the same. Allowing negative votes but requiring such candidates' weights to not change is a weakening of the assumption that seems unnecessary, but is actually quite important in terms of easing into the applications of these general lemmas in the surplus transfer locale, as we do not assume or know immediately in the context of a surplus transfer round that all candidates' votes are nonnegative, this has to be proven. Similar results are proven for the excess, quota, etc. Also, important for proof automation, after this we are able to prove that candidates with positive votes remain positive after decreasing other candidates' weights. We then prove Theorem 1.1:

```
theorem all_feasible: "feasible_at i"
```

Figure 3.22: Theorem: the weight vector is feasible at every step of the transfer round, proven.

Note that in the proof of `all_feasible` we make use of strong as opposed to weak induction,[6] necessary due to `feasible_at_Q_pos` requiring every prior round to be feasible. Also note that we have no assumption about what form the quota takes, and so this is general over any integer or fractional dynamic quota one might use. The proof is completed in a little under 200 lines.

[6]Strong induction, for natural numbers specifically, has at the induction step that the proposition is true for all previous ($\forall m < n$) steps and one is required to prove it is true for the current ($n$) step. In weak induction the proposition must be proven true for a base case ($n = 0$) and then given it is true at the current step ($n$) one must prove it is true for the next ($n+1$).

With Theorem 1.1 proven, we can then prove stronger versions of lemmas like `feasible_at_Q_pos` with the assumptions removed, along with a number of other corollaries, including the following now undecorated by any assumptions besides the fact that the $c$ under consideration is in each case a candidate (or more specifically for one lemma, an elected candidate):

```
corollary V_sum_pos:
  "(Σc∈cands. V_at i $ c) ≥ 1"

corollary E_lt_max:
  "E_at i ≤ num_ballots - 1"

corollary Q_pos:
  "Q_at i > 0"

corollary elected0_Vi_ge_Qi:
  assumes "c ∈ elected0"
  shows "V_at i $ c ≥ Q_at i"

corollary w_ge0:
  assumes "c ∈ cands"
  shows "w_at i $ c ≥ 0"

corollary w_le1:
  assumes "c ∈ cands"
  shows "w_at i $ c ≤ 1"
```

Figure 3.23: Various important and now easy to state corollaries to the first part of Theorem 1.

#### 3.5.1.1 Theorem 1.1: additional remarks

There are a number of basic facts one needs for this development that one would not think to explicitly prove using pen-and-paper. For example, some proofs to do with nonnegativity or positivity of candidates' votes e.g. after weight updates. They are particularly important for facilitating proof discovery by `sledgehammer`. One such lemma is the fact that if all candidates' weights are valid, then all candidates' votes are nonnegative:

```
lemma valid_all_nonneg:
  assumes "c ∈ cands"
      and le1: "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
      and ge0: "⋀c. c ∈ cands ⟹ w $ c ≥ 0"
  shows "V_for w $ c ≥ 0"
```

Figure 3.24: Lemma stating that all candidates' votes are non-negative as long as their weights are valid.

We prove this using a similar induction trick to that mentioned earlier where we start at the 0-vector and map back to the original weight vector one by one. For this particular lemma, `valid_all_nonneg`, we then prove:

1. The singleton set base case using `elim_winc` and a case split on whether the singleton element is `c`, and
2. the inductive step using `elim_winc`, the lemma assumptions, and the fact that eliminated candidates have no votes.

Things such as these basic facts clarified the delicate structure of some of what is implicit in Woodall's proof in terms of the order in which one is able to build up these lemmas. For example, the positivity of the quota for feasible weights only follows once you have a number of other results about nonnegativity of votes, the fact that the excess never reaches its upper bound thanks to the non-triviality criterion, and some of this in turn requires first proving weaker lemmas as a stepping-stone to what one will be able to finally prove, and so on.

Illustrative of this, we had originally encoded in our assumptions the variant of Meek's method where new candidates can become elected when they reach the quota during a surplus transfer round (i.e. the "enlisting" approach); this completely changes the natural order of proof and makes initial proof development much easier, but introduces other significant complications when it comes to convergence. See Section 6.2.6 on future work to do with this variant.

### 3.5.2 Theorem 1.2: the process converges and the limit is a solution

The second part proceeds in roughly four steps:

1. Prove $(\lambda i.w_c^{(i)})$ is convergent.

2. Show $0 \leq V_c(w^{(i)}) - Q(w^{(i)}) = V_c(w^{(i)}) - V_c(w') \leq T(w^{(i)} - w^{(i+1)})$
3. Show the validity of `w_inf`.
4. Using the above series of inequalities, `V_cont`, and `Q_cont` we show `V_for w_inf = Q_for w_inf`.

Recall that $T$ is the number of ballots, as introduced in Section 1.1. The validity of `w_inf` simply follows from the fact that $(\lambda i.\ w_c^{(i)})$ is bounded below by 0 and above by 1. The convergence of `w_at` follows as the weights are bounded below by 0 and are weakly monotonically decreasing as a consequence of Theorem 1.1, given that non-elected candidates' weights do not change and elected candidates weights are always multiplied by some number between 0 and 1.

#### 3.5.2.1 Theorem 1.2: a non-proof in Woodall

For the series of inequalities however, Woodall claim that

$$V_c(w^{(i)}) - V_c(w') \leq T(w_c^{(i)} - w_c^{(i+1)})$$

is true "since decreasing $w_c$ by $\delta$ cannot decrease $V_c(w)$ by more than $T\delta$". We can prove that the left-hand side can be rewritten so that this reduces to proving the following:

$$V_c(w^{(i)})(w_c^{(i)} - w_c^{(i+1)})/w_c^{(i)} \leq T(w_c^{(i)} - w_c^{(i+1)})$$

This follows immediately if $w_c^{(i)} = w_c^{(i+1)}$. If they are not equal, this reduces to showing $V_c(w^{(i)})/w_c^{(i)} \leq T$, i.e. the total votes received before passing on according to $c$'s weight does not exceed the total number of ballots.

We prove a generic version of this, i.e. a version not dependent on steps of the round, in the locale `meekabstract_carrier`. We were for a time unsure about the provability of this at the level of abstraction where one does not have access to ballot structure, and had included it as an additional assumption to the locale, which felt somewhat unjustified and out of place at that level.

Once we had made it an assumption, we proved it on the interpretation side using a lot of unfolding, simplification, and refolding for the specific implementation we use of the various functions. However, we eventually found a way to prove it using a trick similar to that we showed earlier with `w_le1_V_ge0` to do with mapping weights to some constant and back, and using the typical induction pattern we have also already shown with `eq_cand_weights_V_eq`:

```
theorem fracs_le_num_ballots:
  assumes c_cand: "c ∈ cands"
      and w0: "⋀c. c ∈ cands ⟹ w $ c ≥ 0"
      and w1: "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
  shows "V_for w $ c / w $ c ≤ num_ballots"
```

Figure 3.25: Lemma stating that the total votes received before passing on according to $c$'s weight does not exceed the total number of ballots.

If $w_c = 0$ or $V_c(w) = 0$ the theorem follows trivially. In all other cases we are dealing with a candidate with positive weight and some votes. Dividing the votes by the candidate's weight recovers – as long as $w_c \neq 0$ – is the raw amount of votes they receive from others "before passing anything on". We briefly discussed what we have been calling *frac-of*, the "fractions of votes going to a candidate given some ballot", in Section 1.1. This quantity $V_c(w^{(i)})/w_c^{(i)}$ essentially represents the sum of fractions of each ballot going to $c$ at step $i$. The function *frac-of* will be given its proper Isabelle/HOL definition in Section 5.2. We now give an outline of the proof.

```
proof -
  have "V_for (w<cands⟼ 1><Y⟼ w>) $ c /
          w<cands⟼ 1><Y⟼ w> $ c ≤ num_ballots"
    if "Y ⊆ cands" for Y
  using finite that assms
  proof (induct Y rule: finite_induct)
    case empty
    (* ... *)
    finally show ?case .
```

Figure 3.26: Outline of the proof of `fracs_le_num_ballots`: first fragment.

The trick here is to first map all candidates' weights to 1. The base case, which we cut out here for brevity, reduces to proving $V_c(w < cands \longmapsto 1 >) \leq T$, which follows by a helper lemma (which is what triggered the idea for the proof of this theorem):

```
lemma w1_V_le_num_ballots:
  assumes c_cand: "c ∈ cands"
      and w1: "⋀c. c ∈ cands ⟹ w $ c = 1"
  shows "V_for w $ c ≤ num_ballots"
```

Figure 3.27: Outline of the proof of `fracs_le_num_ballots`: second fragment.

This follows from the invariant plus the fact that there is no excess when all candidate weights are 1. We then proceed with the induction step:

```
  next
    case IH: (insert x X)
    then have IH': "V_for (w<cands↦ 1><X↦ w>) $ c /
                      w<cands↦ 1><X↦ w> $ c≤ num_ballots"
      by blast
  (* ... *)
  qed
  thus ?thesis
    by fastforce
qed
```

Figure 3.28: Outline of the proof of `fracs_le_num_ballots`: third fragment.

We cut out most of the proof here, as neither the structure nor the proof methods are enlightening beyond what we describe in the following. Overall the theorem requires over 100 lines even with several helper lemmas which were created just for this purpose. We first begin by extracting the induction hypothesis, with our goal to prove being `V_for (w<cands ↦ 1><insert x X ↦ w>) $ c / w<cands ↦ 1><insert x X ↦ w> $ c ≤ num_ballots`. We then case split on whether `w $ c = 0`. The $w_c = 0$ part of the case split is trivial, though we need an additional helper lemma to say that the votes a candidate receives after multiplication by the weight is bounded above by $T$. The second part of the case split is a bit more involved, though in the end only requires some important lemmas for pushing through algebra to do with our vector updating functions over candidates, the assumption `V_change`, and several invocations of the helper lemma already mentioned to do with votes kept being bounded above by `num_ballots`.

#### 3.5.2.2 Theorem 1.2: completing the proof

Finally, we can prove `V_for w_inf $ c = Q_for w_inf` for every elected $c$. We have that $V_c(w^{(i)}) - Q(w^{(i)})$ is bounded below by 0 and above by $T(w_c^{(i)} - w_c^{(i+1)})$, and this upper bound also converges to 0 because the weights converge, and hence the difference is "squeezed" down to 0:

```
lemma V_lim_eq_Q_lim:
  assumes "c ∈ elected0"
  shows "(λi. V_at i $ c - Q_at i)⟶0"
```

Figure 3.29: The different between the votes and the quota of each elected candidate is squeezed down to 0.

which finally enables us to prove Theorem 1.2:

```
theorem w_inf_solution:
  assumes gfq: "∀w. Q_for w
          = (num_ballots - E_for w) / (seats + c1) + c2"
  shows "solution_for w_inf"
```

Figure 3.30: As long as the quota is fractional, the limit vector in the transfer round is a solution vector.

where we need a fractional quota because we need the following fact that does not follow using an integral quota (along with an analogous `V_w_inf`):

```
lemma Q_w_inf:
  assumes gfq: "∀w. Q_for w
          = (num_ballots - E_for w) / (seats + c1) + c2"
  shows "Q_at⟶ Q_for w_inf"
```

Figure 3.31: As long as we are using a fractional quota, the quota remains bounded above by the value of the quota in the limit.

Nevertheless, this is still a generalisation of the original theorem.

## 3.6 Theorem 2: uniqueness of the solution

**Theorem 2.** *"The solution vector, whose existence was proved in Theorem 1, is unique."*

In Woodall this amounts to taking two arbitrary solution vectors `w` and `w*` and showing `w_inf = w` and `w = w*`. We produce a modified proof which takes an arbitrary `w` and show `w_inf = w`. We will use Isabelle notation exclusively here, as we will follow the proof relatively closely and need to refer to newly introduced constants. It consists of the following parts:

1. Construct the vector `w_min` which is the element-wise minimum of `w_inf` and `w`.
2. Prove `w_min` is feasible and hence construct the solution vector `w_min_inf`, which exists by Theorem 1.
3. Prove `Q_for w_min_inf = Q_for w` and `∀c∈cands. V_for w_min_inf $ c = V_for w $ c`.

4. We have ∀c∈cands. w_min_inf $ c ≤ w $ c.
5. Let L = {c∈cands. w_min_inf $ c < w $ c }.
6. Show L = {} using (3) and the fact that strictly increased weights implies strictly increased votes, thus ∀c∈cands. w_min_inf $ c = w $ c.
7. Thus ∀c∈cands. w_min_inf $ c = w_inf $ c, and so ∀c∈cands. w $ c = w_inf $ c, as required.

Steps (3–6) must be repeated, replacing w by w_inf. As is commonly the case in prose, this repetition can be dispatched by careful use of "without loss of generality" or "by analogy to the previous proof". Unfortunately, aside from a few specific attempts [38], this is often difficult to achieve formally.

### 3.6.1 Theorem 2: statement

We will present our proof interspersed with associated discussions, starting with the statement of Theorem 2 in Isabelle:

```
theorem unique_solution:
  assumes gfq: "Q_for = (λw. (num_ballots - E_for w) /
                         (seats + c1) + c2)"
      and "c1 > 0"
  shows "∃w. solution_given_for (w_at i) w ∧
         (∀w'. solution_given_for (w_at i) w' ⟶
                (∀c∈cands. w' $ c = w $ c))"
```

Figure 3.32: The solution to the transfer round is unique.

The proof is the same whether we write w_at 0 or w_at i in the statement, so we make the easy generalisation. We are not able to use the binder ∃!w meaning "there exists a unique w" because we need to restrict our statement to the set cands. For this purpose we introduce a helper lemma which breaks the proof down into first showing existence, which follows trivially from the fact that we have one solution already in w_inf, then showing uniqueness. The additional constraint on c1, that it must be positive and hence at least 1 (given it is a natural number), comes from our generalised proof of the fact that Q_for w_min_inf = Q_for w:

```
lemma Q_eq_via_other_le:
  assumes gfq: "∀w. Q_for w
          = (num_ballots - E_for w) / (seats + c1) + c2"
      and "c1 > 0"
      and "card X < seats + c1"
      and "Q_for w_le ≤ Q_for w"
      and "∀c ∈ X. w_le $ c ≤ w $ c
               ∧ Q_for w_le = V_for w_le $ c
               ∧ Q_for w = V_for w $ c"
      and "∀c. c ∉ X ⟶ w_le $ c = w $ c"
      and "∀c. w_le $ c ≥ 0"
      and "∀c. V_for w $ c ≤ 0
               ⟶ w_le $ c = w $ c"
  shows "Q_for w_le = Q_for w"
```

Figure 3.33: The fractional quota is equal for weight vectors related by

The reason we need $c_1 > 0$ in this lemma is (at least) for a step in a proof by contradiction where we derive the fact (where S is `seats`) that $S + c_1 < |X|$, where this contradicts $|X| < S + c_1$ only if $c_1 > 0$. For $X = elected0$, $|elected0| < S + c_1$ follows from our locale assumption `seats_not_exceeded` which says that $|elected0| \leq S$, again as long as $c_1 > 0$.

### 3.6.2 Theorem 2: construction of a feasible vector and self-interpretation

One of the key parts of Theorem 2 is constructing the solution vector `w_min_inf`. In Woodall this is done by claiming the five assumptions about the properties of the functions (see Section 3.5.2) plus Theorem 1 produce a solution vector, and that all of this applies equally to `w_min` given that it is feasible. Formally, this is a lot more work. We need to *interpret* the locale `meektransfer_carrier` within itself using `w_min`, and thus cannot assume anything in that locale not provable at this level of abstraction at this point in the proof.

An interpretation of a locale in Isabelle/HOL is an instantiation of the arguments with particular constants and functions with proofs that these satisfy all of the assumptions of the locale. Here we are instantiating the arguments (line 13) using the same constants and functions we have access to within the locale *except* we instantiate `transfer_weights` with `w_min` (defined on line 4).

```
next
  fix w
  assume w_sol: "solution_given_for (w_at i) w"
  define w_min where "w_min ≡ χ c. min (w $ c) (w_inf $ c)"

  (* ... *)

  have "feasible_for w_min" unfolding feasible_def
  <proof>

  (* ... *)

  interpret w_min_loc: meektransfer_carrier Q_for c1 c2 E_for
    num_ballots seats cands w_min elected0 V_for
  <proof>
  let ?w_min_inf = w_min_loc.w_inf
  have w_mininf_sol: "solution_given_for w_min ?w_min_inf"
  <proof>

  (* ... *)
```

We then proved each assumption of the locale for `w_min` in turn, where only the non-negativity of votes and the non-triviality criterion required a non-trivial proof step. After proving a number of inequalities involving the various weight vectors, we continue:

```
1  have inf_le: "Q_for ?w_min_inf ≤ Q_for w"
2  <proof>
3  have Q_eq: "Q_for ?w_min_inf = Q_for w"
4  <proof>
5  then have V_eq: "⋀c. c ∈ elected0 ⟹
6                       V_for w $ c = V_for ?w_min_inf $ c"
7  <proof>
```

The first fact (line 1) follows from a lemma `many_wdec_Q_winc`, analogous to `many_wdec_V_wdec` (shown earlier) but for the quota, proven in the locale `meekabstract_carrier` locale and reused a few times in the proof of this theorem. The second fact (line 3) follows from `Q_eq_via_other_le` (see Section 3.6.1). The final fact (line 5) follows simply from unfolding and the simplifier.

### 3.6.3 Theorem 2: considering each ballot paper

This brings us to the final step (6) in the proof outline we enumerated earlier, where we must conclude `L = {}`.

Woodall claims that the fact that the votes strictly increase with strictly increasing weights is not difficult to see, as one can see this is the case "by considering each ballot paper". This is what we meant in Section 3.1.1 when we referred to two different levels of abstraction in Woodall's argument. After much trial and error, we concluded that this is best (perhaps only) provable with access to ballot structure (see Section 5.4), as Woodall's comment implies, and so we introduce a new, straightforward assumption, which may appear complicated because of the required validity restrictions on the weights. This leads to the following locale extension:

```
locale meektransfer_strictinc_carrier =
  meektransfer_carrier +
  assumes many_dec_V_sum_dec:
    "⋀w w_le. ⟦
       X ⊆ cands; X ≠ {};
       ⋀c. c ∈ X ⟹ w_le $ c < w $ c;
       ⋀c. ⟦c ∈ cands; c ∉ X⟧ ⟹ w_le $ c = w $ c;
       ⋀c. c ∈ cands ⟹ w_le $ c ≥ 0;
       ⋀c. ⟦c ∈ cands; V_for w $ c ≤ 0⟧ ⟹ w_le $ c = w $ c⟧
    ⟹ (Σc∈X. V_for w_le $ c) < (Σc∈X. V_for w $ c)"
```

Figure 3.34: An additional assumption for a *strictly* decreasing sum over a strictly decreased weight vector.

Theorem 2 lives inside this final locale context. With all of this, including much of the setup for Theorem 1 described in Section 3.3, we can prove this final step:

```
1   let ?L = "{c ∈ elected0. ?w_min_inf $ c < w $ c}"
2
3   (* ... *)
4
5   have "?L = {}"
6   proof (rule ccontr)
7   (* ... *)
8   then have w_w_min: "w $ c = w_min $ c"
9       if "c ∈ cands" for c
10  <proof>
11
12  (* ... repeat 3--5 *)
13  then have w_min_w_inf: "w_min $ c = w_inf $ c"
14      if "c ∈ cands" for c
15  <proof>
16
17  show "∀c∈cands. w $ c = w_inf $ c"
18    by (simp add: w_min_w_inf w_w_min)
19 qed
```

The set $L$ is defined (line 1) and proven to be empty by contradiction (line 6), as

described previously. We then show that the weight vector constructed at the start of the proof, `w_min`, is equal to the arbitrary weight vector we are trying to show is unique. We then in turn show `w_min` is equal to the limit vector `w_inf` and hence prove that the arbitrary weight vector must, in fact, be the limit vector itself (line 17), concluding the proof of uniqueness.

With all these steps formally demonstrated, the Isabelle formalisation of the two theorems of Woodall is complete.

## 3.7 Additional theorems

The first two theorems, built-on and generalised from Woodall, show that the weight vector converges on a solution. What is not shown is that this convergence process is monotonic; just based on proven results it could still be the case that the total surplus increases in some steps and decreases in others. In this section we prove that after a particular condition has been met, all steps from that point strictly monotonically decrease the total surplus.

### 3.7.1 Abbreviation for surplus and some results

We introduce the following abbreviation, which makes writing and reading lemma statements and proofs involving surplus easier to follow:

```
abbreviation surplus_at :: "'c ⇒ nat ⇒ real" where
  "surplus_at c n ≡ V_at n $ c - Q_at n"
```

Figure 3.35: Abbreviation for the surplus of a given candidate at a given step.

Before moving on to the proof of monotonically decreasing surplus, we will first prove two simple theorems. One which states that for any ε and any candidate $c \in elected0$, we can show that there is some step after which the surplus always remains below ε. The other trickier theorem states that the sum of surpluses of any subset of the elected candidates eventually always remains below ε. This does not get us strict monotonicity, but they are helpful theorems to have in any case.

```
theorem surplus_arbitrarily_small:
  assumes "c ∈ elected0"
      and "ε > 0"
shows "∃n. ∀m ≥ n. surplus_at c m < ε"
proof -
  have "surplus_at c ⟶ 0"
    <proof>
  thus ?thesis
    <proof>
qed

theorem surplus_sum_arbitrarily_small:
  assumes "X ⊆ elected0"
      and "ε > 0"
  shows "∃n. ∀m ≥ n. (Σc∈X. surplus_at c m) < ε"
```

Figure 3.36: Theorems stating there exists a step after which the total and individual surplus remain below ε.

The first of these two theorems demonstrates which we pick the first argument position for the candidate: passing in a candidate gives a sequence of surpluses for that candidate. It follows trivially from the existing development (both of the suppressed proofs are one-liners, suppressed to reduce visual noise).

The second theorem is only a touch trickier. The case where $X$ is empty is trivial. When $X$ is non-empty, we leverage the first theorem to show for all $c \in \mathit{elected0}$:

$$\exists n.\ \forall m \geq n.\ \mathit{surplus\text{-}at}(c,m) < \frac{\varepsilon}{|X|+1}$$

We then take the largest $n$, call it $n_{max}$, at which any of the elected candidates' surplus drops below this value. It thus satisfies:

$$\forall c \in \mathit{elected0}.\ \forall m \geq n_{max}.\ \mathit{surplus\text{-}at}(c,m) < \frac{\varepsilon}{|X|+1}$$

Fix $m \geq n_{max}$. We can then use the above to prove:

$$\begin{aligned}\sum_{c\in X} \mathit{surplus\text{-}at}(c,m) &< \sum_{c\in X} \frac{\varepsilon}{|X|+1} \\ &= \varepsilon\frac{|X|}{|X|+1} \\ &< \varepsilon\end{aligned}$$

which completes the proof.

### 3.7.2 Termination is not over-eager

Here we prove that after the surplus is reduced below ε, there is no point at which the surplus comes back above ε. This is an important result for showing that "stopping too early" is not a concern. To prove this one needs to show that the surplus is monotonically decreasing. This is not immediately obvious in the initial phase of the round given considerations about elected candidates, who may have a weight of 1, and the fact that the quota decreases. We prove the following general theorem about surplus:

```
theorem surplus_sum_dec_general:
  assumes gfq: "Q_for =
                  (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
      and w_props: "⋀c. c ∈ cands-elected0 ⟹ w_le $ c = w $ c"
                   "⋀c. c ∈ elected0 ⟹ w_le $ c < w $ c"
                   "⋀c. c ∈ cands ⟹ w_le $ c ≥ 0"
                   "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
      and elected_V_gt0: "⋀c. c ∈ elected0 ⟹ V_for w $ c > 0"
      and c1_gt0: "c1 > 0"
shows "(Σc∈elected0. V_for w_le $ c - Q_for w_le) <
       (Σc∈elected0. V_for w $ c - Q_for w)"
```

Figure 3.37: General theorem for decreasing surplus.

This states that given a fractional quota and a pair of valid weight vectors, one of which is strictly less than the other element-wise across elected candidates, the sum of surpluses over the lesser weight vector is smaller than the sum of surpluses over the other weight vector. This follows by case-splitting on whether the excess changes plus largely leveraging results from previous sections.

At first glance this seems to be enough to prove that surplus strictly monotonically decreases, but it is not, because we have never actually proved that once candidates' weights all strictly decrease on one step, they all strictly decrease on all subsequent steps. We will not dwell on the details here, but the key lemma we build up to is the following:

```
lemma Q_dec_w_always_dec:
  assumes gfq: "Q_for =
                  (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
      and Q_dec: "Q_at (i + 1) < Q_at i"
      and i_lt_j: "i + 1 < j"
      and j_lt_k: "j < k"
      and c_elec: "c ∈ elected0"
  shows "w_at k $ c < w_at j $ c"
```

Figure 3.38: If the quota decreases once, all elected candidates' weights always decrease.

This states that as long as the quota decreases once, and it is a fractional quota, the weights of every elected candidate at any later steps $j$ and $k$ where $i+1 < j < k$ satisfy $w_c^{(k)} < w_c^{(j)}$. The requirement for the quota to decrease once is important: our formalisation of the surplus transfer round includes cases where the so-called infinite iterative convergence is over in a few steps.

Consider a case where two candidates $a$ and $b$ are elected with $w_a = w_b = 1$. If $a$ has surplus votes and $b$ does not, $a$ could transfer all of their surplus to $b$, and then if nobody who listed $b$ first listed $a$ anywhere below $b$ on their ballot, $b$ transfers all of their votes somewhere other than back towards $a$. If $b$ transfers all their votes to some third candidate $c$ who still does not reach the quota, the surplus transfer round terminates after just a few steps, with strictly zero remaining surplus. With that said, the desired theorem is now provable:

```
theorem surplus_sum_always_dec:
  assumes gfq: "Q_for =
                  (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
      and Q_dec: "Q_at (i + 1) < Q_at i"
      and i_lt_j: "i + 1 < j"
      and j_lt_k: "j < k"
      and c1_gt0: "c1 > 0"
shows "(Σc∈elected0. surplus_at c k) <
       (Σc∈elected0. surplus_at c j)"

corollary surplus_sum_strict_mono:
assumes gfq: "Q_for =
                (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
    and Q_dec: "Q_at (i + 1) < Q_at i"
    and c1_gt0: "c1 > 0"
shows "strict_mono (λj. - (Σc∈elected0. surplus_at c (i + 2 + j)))"
```

Figure 3.39: As long as the quota decreases once, the surplus sum strictly monotonically decreases.

This initial series of steps where nothing may happen to decrease the quota may be arbitrarily long in theory, though this will likely be unheard of in practice even for one or two steps. The corollary simply leverages an existing Isabelle/HOL standard library constant for expressing strict monotonic decreasing sequences.

A consequence of this result is that we know that the iterative procedure does not ever send weights erratically through the space of possible solution vectors. For example, before proving this it is conceivable that some intermediate weight vector could drastically reduce the total surplus, so that it falls below ε, such that the round terminates. Then, on the next step, the surplus is increased back above ε, and only then much later is the true solution vector approached. In this sense, we now know that there is no such thing as "over-eager" termination.

## 3.8 Elimination round

In this section we briefly cover applying the results proven for the surplus transfer round to the elimination round.

### 3.8.1 Elimination round locale

Like we characterised the surplus transfer round, we have to characterise the elimination round:

```
locale meekelimination_carrier = abstract_meek_carrier +
  fixes weights :: "(real, 'c) vec"
    and pick :: "'c set ⇒ 'c"
  assumes eliminatable: "∃c∈cands. weights $ c > 0"
      and w_g0: "c ∈ cands ⟹ weights $ c ≥ 0"
      and w_le1: "c ∈ cands ⟹ weights $ c ≤ 1"
      and valid_pick: "⟦X ⊆ cands; X ≠ {}⟧ ⟹ pick X ∈ X"
```

Figure 3.40: Elimination round locale.

Like the surplus transfer round locale, we extend the locale which provides the very high-level abstract characterisation of the component functions. Also like the surplus transfer round, we have to provide some assumptions which characterise when the elimination round *can* apply. Here, this is just that there is some candidate available to eliminate. We also fix some function `pick` which can decide between equally-good candidates for elimination, whose only property must be that it picks from the set we give to it.

We assume weights are valid; this is no different from the surplus transfer round, where we assumed that the weights were valid *on the first step*, and had to prove they remained so. With the elimination round there is only the one step, so there is no additional work to do. We introduce the following definitions:

```
definition noneliminated :: "'c set" where
  "noneliminated ≡ {c∈cands. weights $ c > 0}"

definition lowest_cands :: "'c set" where
  "lowest_cands ≡ {c∈noneliminated. ∀c'∈noneliminated. V_for
   weights $ c ≤ V_for weights $ c'}"

definition eliminated :: 'c where
  "eliminated ≡ pick lowest_cands"

definition elim_weights :: "(real, 'c) vec" where
  "elim_weights ≡ weights<eliminated ↦ 0>"
```

It is then possible to trivially leverage all of the work we did proving general results in preparation for the surplus transfer round, such that all of the following are provable automatically (albeit occasionally by using `smt`):

```
lemma eliminated_V0:
  "V_for elim_weights $ eliminated = 0"

lemma Q_wdec:
  "Q_for elim_weights ≤ Q_for weights"

lemma elim_V_wdec:
  "V_for elim_weights $ eliminated ≤ V_for weights $ eliminated"

lemma all_V_winc:
  assumes "c ∈ cands"
      and "c ≠ eliminated"
  shows "V_for elim_weights $ c ≥ V_for weights $ c"

lemma E_winc:
  "E_for elim_weights ≥ E_for weights"

lemma elim_diff:
  "V_for weights $ eliminated - V_for elim_weights $ eliminated =
   E_for elim_weights - E_for weights +
   (Σc∈cands-{eliminated}. V_for elim_weights $ c) -
   (Σc∈cands-{eliminated}. V_for weights $ c)"
```

## 3.9 Size of formalisation (rough de Bruijn factor)

It is difficult to put a number on the difficulty of a formalisation compared to what it would take in a rigorous pen-and-paper effort, but a large part of the development in this chapter is devoted to a formal reconstruction of Woodall's pen-and-paper proof. The original proof takes up around 1.5 pages, whereas the relevant theory files (excluding additional theorems) come in around 4,700 lines, thousands of lines of Isabelle/HOL per page of proof.

Compared to our development of a formal theory of Minkowski spacetime in separate work [68], where 22 pages convert into roughly double the number of lines (around 9,000), this development seems like quite a lot. Even if one said the spacetime development were really more like 11 pages taking into account font size, this development on Meek's method is still notably larger. I do not think this is surprising, given that the spacetime development is based on a source text *already* very rigorous, and our source text was more of a proof sketch in parts as opposed to more rigorous definition-lemma-theorem style development. Furthermore, we have generalised enough of the results that as we will see later allow us to fairly trivially prove properties of the elimination round as well.

There have been a handful of iterations refining this, which represents its own hidden complexity. Lines-of-code is not a terribly good measure of complexity however, and one could add many lines to this by additional comments, a different approach to Isabelle style, and a more rounded development of lemmas including many more not directly connected to our end-goal, and indeed remove many lines by using a very compressed style. The whole development around specifically the surplus transfer round is an order of magnitude greater, over 10,000 lines.

## 3.10 Related work

As far as we know there is no existing fully formal treatment of any aspect of Meek's method. There is Woodall's proof of solution uniqueness [41], of course, and we have covered almost every aspect of it in this chapter and more beyond.

Wichmann's batch of tests [82] for Meek's method focuses on differences in two implementations of Meek's method, the effect of adding a small epsilon to the quota, and how the final outcome changes depending on one's approach to tie-breaking. They discover multiple errors in the implementations. However, their results are only tan-

gential to our focus on the transfer round.

There are a number of formal verification efforts involving other forms of STV, which we mention briefly. Ghale et al. [25] formalise the specifications of a number of STV methods in Coq using descriptions of the methods as sequences of rules. One of their results is in enabling one to check more easily whether legal and formal descriptions really match, as well as being able to apply the approach to a number of different STV methods. This is an improvement on Dawson et al. [17] – who provide a nice overview of current work in this area – principally as using Coq allows one to extract the specification to runnable Haskell, and so produce verified software without much additional effort, and because Ghale et al's approach scales to larger elections. They do not apply their approach to Meek's method. The work of Ghale et al. [25, 26, 24] is the closest thing there currently is to a framework for formally verified STV in general.

One could potentially subsume Meek's method into one of these representations, by representing the surplus transfer round non-computationally; for example, approaches which specify what the various rounds' pre- and post-conditions are could simply state that the weight vector is updated such that the votes of all the elected candidates are reduced to the quota and non-elected candidates' weights are left the same. Of course, one would first have to implement a more general form of the votes and excess functions which is compatible with Meek's-method style weights. However, any specification which relies on there being no surplus left after the surplus transfer round will fall down when applied to Meek's method as it is practically deployed, as its surplus transfer round terminates when $V_c(w) < Q(w) + \varepsilon$. Each of the approaches also fail to subsume Meek's method if they assume a static quota, which, as Meek's method pioneered the use of a dynamic quota and is as such not very common, they invariably do assume a static quota at present.

## 3.11 Conclusion

We have formalised and expanded upon a proof that the equations $V_c(w) = Q(w)$ to be solved simultaneously for each elected candidate $c$ during the surplus transfer round of Meek's method of STV always has a unique and valid solution. One necessarily converges to a unique solution vector by iteratively updating the elected candidates' weights according to $w_c^{(i+1)} = w_c^{(i)} Q(w^{(i)})/V_c(w^{(i)})$ as long as (in addition to transfer round validity and non-triviality conditions) the quota is of the form $Q(w) = (T - E(w))/(S + c_1) + c_2$ where $c_1 \in \mathbb{N}_+, c_2 \in \mathbb{R}$.

We identified several gaps in the original presentation by Woodall, and filled them in the course of the formal development. This includes the appeal to intuition in "by inspecting each ballot", the error in the proof whereby an inequality was "proved" by stating the inequality in English, by removing the appeal to intuition present in applying the specific weight-update results in the first theorem to a quite different context in the second theorem, and by drawing out what is necessary to assume about the components in the context of a simple-to-state invariant. The rest is rigorously filling in all of the necessary details and intermediate steps, hopefully providing a greater appreciation for the subtlety of the two key theorems as a whole, and providing insight into the functioning of the method. Other gaps related to connecting the implementation more rigorously both with the assumed and proven properties, are covered in Chapter 5.

The locale assumptions we have settled on were developed through several rounds of refinements (see also Section 5.4.2.1 on refinement), and should serve as a good basis for further analysis of the method at this level of abstraction if desired. Within the two locales developed we prove two main theorems.

The first theorem shows that the transfer round weight vector remains feasible throughout and that, as a result, it converges on a solution vector. We build up to proving this by proving several general results about the behaviour of the functions `V_for`, `E_for`, and `Q_for` given changes to a single candidate's weight followed by changes to the weights of any number of candidates.

The second theorem states that the solution vector is unique. That is, for any other vector which is a solution to the transfer round, it must be equal to the solution one gets by taking the limit of the sequence of feasible vectors, defined by starting with the initial weight vector at the start of the transfer round and iteratively updating according to the weights update rule.

It is possible that our commitment to not representing ballots on any level except in their total number in the two locales is unnecessarily limiting. While it is true that there is little point in representing full ballot structure at this level of abstraction, and we have not needed more than what we currently have for deriving the results in Chapter 5, we could have looked more into meeting abstract ballot representation somewhere in the middle. For example, a sub-locale characterising the set of *listed* candidates would enable us to introduce assumptions distinguishing the effects of changes to the various functions between listed and unlisted candidates. Potentially, something along the lines of this could allow us to prove lemmas like `many_dec_V_dec` (see Section 3.6.3) at the more abstract level of the locales.

We will return to these locales in Chapter 5 to show our implementation is a model of the abstraction developed in this chapter. In the following chapter we will first develop a theory for ballots suitable for aiding this interpretation.

# Chapter 4

# Representing ballots and elections in Isabelle/HOL

This chapter develops a theory for ballots sufficient for connecting our implementation in the following chapter, Chapter 5, to our abstraction in Chapter 3.

## 4.1 Overview

In this section we provide an overview of the chapter covering our aims, motivations, hypotheses, and key ideas. We conclude with a summary of the structure of the rest of this chapter.

### 4.1.1 General aims

This chapter is exclusively concerned with developing a theory of ballots which can serve two main aims. One such aim is for it to serve as a basis for reasoning about the implementation in Chapter 5 and hence as the necessary glue between the abstracted development of Chapter 3 and the implementation.

The other is for it to be isolated from the rest of the development to do with Meek's method and thus provide the option for it to be used independently as a starting point for other work which needs to cover preference orderings or ballots (the two are equivalent for our purposes). This theory development for ballots has to satisfy a few more specific properties:

- It should be suitable for combination with our implementation in a relatively frictionless way; that is, representational decisions should not be made in isolation from the implementation.
- In order to serve as the glue between the implementation and the abstraction, it needs to allow us to dispatch each of the locale assumptions for our specific implementation.
- The theory should help minimise boilerplate in proofs by providing lemmas useful for automated proof search, case split rules, and induction rules which do not require too much initial heavy lifting in the induction step.
- Finally, where possible, it should not bake-in representations for strict ballots too deeply, allowing the theory to be extended for non-strict ballots.

We also need to provide a way of representing *election context* that consists of at least a set of valid ballots and candidates ready to be counted. This context allows for further notational convenience and provides a context with which we can provide an

interpretation of the locales. That is, we will interpret the locales from Chapter 3 within a locale representing election context.

### 4.1.2 Specific objectives

Our specific objectives are determined by what will be needed for our formal development (of Chapter 3 and Chapter 5) as it relates to Meek's method. The focus of our work is not on building a general framework for ballots and elections, but on verifying the surplus transfer round of Meek's method of STV. However, we have kept in mind potential generalisation throughout and discuss these where appropriate. We thus set our concrete objectives as follows (with more details to follow in the chapter):

- Define strict ballots in terms of the ballot's carrier set $\mathcal{L}$ and the greater-than function $\mathcal{G}$ in line with our implementation (see Section 5.2), suitable for reasoning at the level of ballots-in-general and the implementation of the components of STV specifically.
- Prove a rule for ballot induction, needed for more complex proofs.
- Define constants necessary for capturing the ballot's ranking of candidates in terms of natural numbers, as with real elections where candidates are ranked by assigning them to the numbers 1, 2, and so on in order of preference.
- Prove that the function which identifies the candidates on the ballot with their rank is bijective on the set of listed candidates, can be fully identified with a function that returns a candidate given a rank, and this latter function can in turn be used to state a few key properties such as the fact that those listed greater than the candidate at rank $n$ have ranks 0 through to $n-1$.

### 4.1.3 Motivation

The motivation for this chapter is relatively simple. This development is necessary in order to reason about the implementation in the context of a set of valid, strict ballots, in order to connect it with the results in Chapter 3.

However, it is not just this. It is worthwhile highlighting and isolating those aspects of the development which can be plucked out and applied in other contexts. There are several things in both of the other core chapters that can be taken as general lessons, approaches, or starting points for generalisation, but everything in this chapter exists entirely on its own. We hope that it could serve as a good basis for generalisation beyond strict ballots and beyond a mere election *context*.

### 4.1.4 Hypothesis and evaluation

We claim that using a carrier set and greater-than function on candidates is a suitable and even desirable representation for ballots or preference orderings, and that it scales well for more complex proofs involving ballot induction. We demonstrate this through the use of a ballot induction rule to prove various uniqueness results for ballots in general, and we will use it multiple times again in Chapter 5.

We also claim that parts of the theory (ballots, weights) can be linked into existing ordering theorems developed for carrier sets in the Isabelle/HOL standard library, by interpreting the ordering locales provided by HOL-Algebra specifically, and that this can then be leveraged for automation. Providing this link is essentially equivalent to showing that our functional approach can be translated into the more common binary-relation-first approach. We have indeed proven that we can link into HOL-Algebra in this way, however our hope that this could be leveraged for automation did not come to fruition, and so we have to answer this claim negatively until developments in existing Isabelle libraries are improved, e.g. as part of the ongoing overhaul of HOL-Algebra by the maintainers, to the point where we can come back and leverage them for this formalisation. As the automation was not able to be leveraged it did not make any impact on the formalisation, and so we will not cover our development surrounding this in the thesis.

The ballots theory is intended to allow us to interpret the locales from Chapter 3, and indeed we use it to do so in Section 5.4. Our representation originates from the development discussed in Section 5.2, though as mentioned our ballot formalisation is independent of Meek's method and could in theory be used to study elections in general.

### 4.1.5 Novel concepts and ideas

The representation of ordering, whether for preferences or ballots, is one of the main novelties presented in this chapter. We begin from a function $\mathcal{G}$, as opposed to a binary relation $\succ$ over pairs of candidates, or a relation represented by a set of pairs of candidates. This is novel, at least within the topic of voting and social choice in Isabelle/HOL. The carrier set $\mathcal{L}$ would have been needed in any approach, as each ballot has its own set of listed candidates. Of course, each possible representation is in the end equivalent, and one can prove transfer rules for going from one to the other.

As far as we are aware, nothing else currently exists in Isabelle/HOL along the lines

of our use of ballot induction for verifying aspects of voting algorithms. Additionally, there is nothing analogous to the definitions for *directly above* and *directly below* candidates or to our approach that connects the basic representation to natural-number rankings.

### 4.1.6 Structure of this chapter

In Section 4.2 we present a formal representation of strict ballots and some initial lemmas. Next, in Section 4.3 we present a representation of subballots along with a few important lemmas, which will help facilitate our ballot induction rule in Section 4.4. In the sections that follow this, Sections (4.5, 4.6, 4.7), we respectively cover two predicates for referring to the unique candidate(s) directly above and directly below a candidate on a ballot, touch on some uniqueness results employing ballot induction, and make some remarks on ballots and automation. Finally, in Section 4.8 we cover definitions and results for connecting ballots to natural-number rankings, before presenting a formal representation of election context sufficient for our purposes in this thesis in Section 4.9. We conclude in Section 4.10 and Section 4.11 by discussing related work and future work respectively.

## 4.2 Strict ballots

In order to reason about voting generally, and Meek's method specifically, we need a way of representing and reasoning about ballots. For us this only needs to be an abstract, minimal representation. It should allow us to state what a valid strict ballot is, and allow us to obtain a candidate or set of candidates satisfying a property on one or more ballots. As mentioned previously, it should also be amenable to an elegant induction rule, and fit well with the implementation.

### 4.2.1 A predicate for strictly ranked ballots

A ballot is simply a total order over a subset of candidates. It may also be necessary in some contexts for it to be adorned with metadata like a voter ID, but at base it is simply a total order. An order which is not over an entire type, as is the case with ballots, carries what is called a *carrier set*. For an ordering over (e.g. the ranking of) a ballot $b$, this is $\mathcal{L}(b)$.

It is known that all one needs to define a total ordering is a less-than (or greater-than) relation and an equality relation both of which are total on the type (or carrier set). What we have is a greater-than relation with the function $\mathcal{G}$ and an equality relation which is just ordinary equality `(=)` over the type `'c`. With the addition of the carrier set provided by $\mathcal{L}$, we clearly have all we need to define a total ordering for ballots. The definition below does just that.

**Definition 1** (Strict ballots)**.** *A ballot $b$, given $\mathcal{L}, \mathcal{G}$, is a strict ballot iff*

- *$\mathcal{L}(b)$ is finite.*
- *The greater-than function maps to a subset of the listed candidates:*

$$\forall c \in \mathcal{L}(b).\ \mathcal{G}(b,c) \subseteq \mathcal{L}(b)$$

- *Candidates are not listed greater than themselves:*

$$\forall c \in \mathcal{L}(b).\ c \notin \mathcal{G}(b,c)$$

- *The greater-than function is relationally transitive:*

$$\forall c\ k\ l \in \mathcal{L}(b).\ c \in \mathcal{G}(b,k) \wedge k \in \mathcal{G}(b,l) \longrightarrow c \in \mathcal{G}(b,l)$$

- *For any two listed candidates, one is listed greater than the other or they are equal:*

$$\forall c\ k \in \mathcal{L}(b).\ c \in \mathcal{G}(b,k) \vee k \in \mathcal{G}(b,c) \vee c = k$$

It is important to note that we cannot talk about a value $b$ being a ballot or not on its own. It is a ballot (or not) *given* some $\mathcal{L}, \mathcal{G}$. This also means that the same $b$ can represent an entirely different ballot given different accessor functions. Because of this, we introduce a shorthand for referring to ballots, namely a tuple $(\mathcal{L}(b), \mathcal{G}(b))$ (*the ballot*), representing the carrier set with type `'c set` and the greater-than function with type `'c ⇒ 'c set`.

We embed this shorthand throughout the ballot theory because only this tuple is relevant when considering a single individual ballot, and not some general $\mathcal{L}$ and $\mathcal{G}$ which are being applied to numerous ballots. We also remove the need to explicitly refer to $b$ and talk simply of the ballot $(\mathcal{L}, \mathcal{G})$ with $b$ left implicit.

So for the remainder of this section, we should think about $\mathcal{L}$ as having the type `'c set` and $\mathcal{G}$ the type `'c ⇒ 'c set`. In the section on elections, Section 4.9, it will become necessary to introduce a fixed $\mathcal{L}$ and fixed $\mathcal{G}$ along with a set of ballots $B$, and

they again become accessor functions in the proper sense, taking the types `'b ⇒ 'c set` and `'b ⇒ 'c ⇒ 'c set`. We overload these two symbols in both the informal and formal notation, though it will be clear when we are using one sense or the other. Thus, the Isabelle/HOL definition below.[1]

```
definition valid_strict_ballot :: "'c set ⇒ ('c ⇒ 'c set) ⇒
   bool" where
  "valid_strict_ballot ℒ 𝒢 ≡ finite ℒ ∧
     (∀x∈ℒ. 𝒢 x ⊆ ℒ ∧ x ∉ 𝒢 x ∧
            (∀y∈ℒ. (∀z∈ℒ. y ∈ 𝒢 x ∧ z ∈ 𝒢 y ⟶ z ∈ 𝒢 x) ∧
                   (x ∈ 𝒢 y ∨ y ∈ 𝒢 x ∨ x = y)))"
```

Figure 4.1: The definition of strict ballots in our theory development.

#### 4.2.1.1 Additional commentary on the definition and its use in Isabelle/HOL

The ballot is represented by the pair $(\mathcal{L}, \mathcal{G})$, which we nevertheless always pass around individually to curried functions. The definition is followed by a number of derived introduction and elimination rules in the usual way as well as obvious corollaries, as is the case for all our definitions. Most of the other basic results about valid (strict) ballots follow immediately from the definition and basic set-theoretical results. For example:

```
lemma in_G_in_L:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "c ∈ ℒ"
      and "k ∈ 𝒢 c"
  shows "k ∈ ℒ"
by (meson 𝒢_sub_ℒ assms psubsetD)
```

Figure 4.2: A simple ballot lemma, directly provable using set-theoretical results.

Isabelle/HOL's ability to work with sets in this way makes the initial development using this representation quite trivial. To further build some intuition for how these functions encode ballot information, consider the strict ballot *abc*. Its representation in Isabelle/HOL could be:

[1] Note that while we require the carrier set to be finite, we do not require the type to be finite, unlike in the rest of the development. This is because this ballots theory is modularly separate from the rest of the formalisation, and so we have one eye on not over-fitting to the rest of the formalisation so that it is more easily pulled out and used by others.

```
ℒ = {a,b,c}
𝒢 = λx.
    if x = a then
        {}
    else if x = b then
        {a}
    else if x = c then
        {a,b}
    else
        undefined
```

Figure 4.3: One possible representation of the ballot *abc* in Isabelle/HOL notation.

Neither the pen-and-paper definition nor the Isabelle one, `valid_strict_ballot`, say anything about what $\mathcal{G}$ should return for candidates not listed on that particular ballot. This is standard in contexts where one has no reason to discuss elements not in the carrier set, which is typically the case. Relatedly, if we know that some $k \in \mathcal{G}(c)$, we also need to know that $c \in \mathcal{L}$ for this to be informative of $k$ (e.g. see Figure 4.2).

Technically speaking, this also means Figure 4.3 is only one of many possible representations of the ballot *abc*, as while $\mathcal{L}$ has to be $\{a,b,c\}$ here, the function $\mathcal{G}$ only needs to return the appropriate sets of $\{\}$ for $a$, $\{a\}$ for $b$, and $\{a,b\}$ for $c$. That is to say, we can just as well implement it using a case-expression instead of an if-expression (though in Isabelle/HOL functions are equal iff they are equal on all inputs, known formally as the *principle of extensionality*), and return whatever arbitrary set we like for values other than these three, such as $\{a, undefined\}$ as opposed to $undefined$. Owing to this, we cannot check equality of the ballots $(\mathcal{L},\mathcal{G}),(\mathcal{L}',\mathcal{G}')$ by simply checking $(\mathcal{L},\mathcal{G}) = (\mathcal{L}',\mathcal{G}')$, but instead we have to check $\mathcal{L} = \mathcal{L}' \wedge (\forall c \in \mathcal{L}.\ \mathcal{G}(c) = \mathcal{G}'(c))$.

Clearly, assuming the ballot is finite is quite reasonable, assuming there is nothing very interesting that can be done in our context with "infinite ballots". This also trivially means there are finitely many candidates listed greater than any other.

We do not preclude the possibility that the ballot may be empty in the definition. One reason for this is that permitting emptiness can ease case splits, which are generated by needing to check whether one is removing the *last* candidate from a ballot, e.g. removing $a$ from $[a]$; in other words, removing candidates could make ballots become invalid. This would also make our induction rule (Section 4.4) and subballots (Section 4.3) more inelegant.

One may wonder why we have restricted ourselves to strict ballots, after all STV allows one to list two or more (even all) candidates as equally preferable. Focusing on these allowed us to make significant progress promptly without having to worry

about non-strictness, which is not too theoretically interesting or essential for practical purposes, given that almost all real-world STV elections disallow voters from listing two or more candidates/parties as equally preferable. We further argue this point in Section 5.2.

#### 4.2.1.2 Basic representation

Before introducing a predicate like `valid_strict_ballot` one has to decide what it will be defined on. That is, what will the types of its arguments be, and what form do the values of that type have to take in order to be considered a "valid ballot".

If we start by assuming that we are using a carrier set $\mathcal{L}$ and a greater-than function $\mathcal{G}$, as we did in the parent section, the definition we gave for valid strict ballots is the obvious one. However, there are multiple other choices for basic representation that are worth considering and comparing to that we have chosen. In particular, of the following would have worked as an alternative basis.

- A list: `b :: 'c list`. In this case we do away with an additional greater-than function, an explicit carrier set, and a predicate for testing whether a ballot is valid. For that reason we can directly give ballots a concrete type with this approach.
- A predicate relation (with a necessary carrier set): $\succ$ `:: 'c` $\Rightarrow$ `'c` $\Rightarrow$ `bool` with $\mathcal{L}$ `:: 'c set`, representing "is preferred to", with the usual notion of equality.
- A set-based relation: `R :: ('c * 'c) set`, where if $(c,k) \in R$ we say "$c$ is preferred to $k$".

The linked-list-based ballot is the most elegant in terms of basic representation, as the list datatype itself builds-in both a notion of a carrier set (the candidates in the list) and the ordering. The only way in which a list can not be a valid ballot is if it lists the same candidate more than once. Nevertheless, there are a few things to say against starting with a list-based representation in Isabelle/HOL, besides our desire for frictionless integration with the implementation of Meek's method (which does not use lists). The main one being that this is often not the most natural for doing traditional mathematical proof, especially not proof development that does not rely much on typical functional implementation styles like folding and mapping. An initial experiment with lists showed this to be true.

The latter two approaches (relation and set-based relation) are the closest to our actual representation and variant forms of both are quite common in general treatments of social choice and voting [5, 15]. One can easily see how one might map between $\mathcal{G}$ and $\succ$ and vice-versa.

However, we consider four largely practical (i.e. not deeply theoretical) aspects that make our actual representation preferable to a more typical relational one. First, ours allows a seamless integration with (our understanding of) a natural implementation of the component functions of Meek's method. Second, choosing a novel base representation is more likely to throw up novel and interesting research questions and directions. Third, when defining an example election, our approach leads to smaller statements. More specifically, when starting from a binary relation in either the predicate or set approach, unless one takes the transitive closure when considering whether a ballot is valid or not (complicating the basic setup), one has to provide pair-wise comparisons for every candidate. Fourth, it makes defining the rank of a candidate in terms of the basic representation trivial (see Section 4.8).

Note that all of these approaches can be made as general as our own approach when it comes to a full election context simply by adding as a first argument to each an additional ballot argument of an unrestricted type variable `'b`. The approach of using type variables for ballots (`'b`) and candidates (`'c`) instead of more basic, concrete types is simple but powerful, as one could start working from our representation and decide that concretely one does not want to *literally* define ballots in terms of functions, and instead only use that representation for analysis. For one's actual ballot representation if one instead, say, wanted them to be defined using the set-based relation approach, you could given some `R :: ('c * 'c) set` transfer it to our $(\mathcal{L}, \mathcal{G})$ like so:

```
  ℒ = fst ` R ∪ snd ` R
𝒢 = (λc. {k ∈ ℒ. (k, c) ∈ R})
```

One would then have to make sure that any given concrete *R* is defined in such a way that it is compatible with `valid_strict_ballot`, given these definitions.

Of course, Meek's method is an algorithm, not just something subject to mathematical analysis, so at some point it is important for its basic types to be suitable for time and space efficient implementation. This is not an issue, though, as one can simply later implement the same functions using more software-engineering appropriate types, prove equivalence with the more mathematical-analysis appropriate types, and perform code extraction on the former.

## 4.3 Subballots

Consider the following induction rule for finite sets from the Isabelle/HOL standard library:

```
lemma finite_psubset_induct [consumes 1, case_names psubset]:
  assumes finite: "finite A"
    and major: "⋀A. ⟦finite A; ⋀B. B ⊂ A ⟹ P B⟧ ⟹ P A"
  shows "P A"
```

Figure 4.4: Induction rule for finite sets in the Isabelle/HOL standard library.

Note that $A \subset B$ in Isabelle/HOL represents a *strict* subset, with $A \subseteq B$ as the non-strict counterpart. We would like to produce a similar induction rule for ballots. The problem we have is that ballots are not simply their carrier sets, they also have the ordering information provided by $\mathcal{G}$, and so this lemma cannot be used as-is. We need our own, and the first step involves providing a ballot-specific version of the subset relation for sets, a $subballot(b, b')$, or more accurately $subballot((\mathcal{L}', \mathcal{G}'), (\mathcal{L}, \mathcal{G}))$, like $A \subset B$ for sets. The following definition works for this purpose:

**Definition 2** (Subballots)**.** *A ballot $(\mathcal{L}', \mathcal{G}')$ is a subballot of a ballot $(\mathcal{L}, \mathcal{G})$ iff*

- *The set of listed candidates is a proper subset: $\mathcal{L}' \subset \mathcal{L}$*
- *The set of candidates greater than some candidate listed on the subballot is the same as with the larger ballot, minus all those candidates who are not on the subballot: $\forall c \in \mathcal{L}'.\ \mathcal{G}'(c) = \mathcal{G}(c) - (\mathcal{L} - \mathcal{L}')$*

```
definition subballot where
  "subballot ℒ' 𝒢' ℒ 𝒢 ≡ ℒ' ⊂ ℒ ∧ (∀c∈ℒ'. 𝒢' c = 𝒢 c - (ℒ - ℒ'))"
```

Figure 4.5: The definition of a subballot in our development.

This could be more precisely named `proper_subballot` but we do not have any reason to use non-proper subballots, and so we opted for this name.

Do note, however, that the definition in Isabelle/HOL does not actually require its arguments to be valid ballots. Related to this, an important theorem immediately suggests itself, analogous to the fact that sub-orders of linear orderings are linear orderings:

```
lemma subballot_valid:
  fixes 𝓛 :: "'c set"
  assumes valid: "valid_strict_ballot 𝓛 𝒢"
      and sub: "subballot 𝓛' 𝒢' 𝓛 𝒢"
  shows "valid_strict_ballot 𝓛' 𝒢'"
```

Figure 4.6: All subballots of valid strict ballots are themselves valid strict ballots.

We then prove some simple lemmas of which the following are a sample:

```
lemma exists_subballot:
  assumes "𝓛 ≠ {}"
  shows "∃𝓛' 𝒢'. subballot 𝓛' 𝒢' 𝓛 𝒢"

lemma exists_subballot':
  assumes "𝓛' ⊂ 𝓛"
  shows "∃𝒢'. subballot 𝓛' 𝒢' 𝓛 𝒢"

lemma remove_one_less_than:
  assumes "valid_strict_ballot (insert c 𝓛') 𝒢"
      and "subballot 𝓛' 𝒢' (insert c 𝓛') 𝒢"
      and "k ∈ 𝒢 c"
  shows "𝒢' k = 𝒢 k" "c ∉ 𝒢' k"
```

Figure 4.7: A sample of formally proven lemmas about subballots.

## 4.4 Ballot induction

We now arrive at one of the important devices needed for more complex reasoning about ballots: ballot induction. We present it below, using curried functions to closer match the formal development and ease presentation.

**Theorem** (Ballot induction)**.** *The ballot* $(\mathcal{L}, \mathcal{G})$ *satisfies some property P if*

- *The ballot is valid:* $valid(\mathcal{L}, \mathcal{G})$
- *Every valid ballot whose every subballot satisfies P itself satisfies P.*
  $\forall \mathcal{L}'\ \mathcal{G}'.valid(\mathcal{L}', \mathcal{G}') \wedge (\forall \mathcal{L}''\ \mathcal{G}''.\ subballot(\mathcal{L}'', \mathcal{G}'', \mathcal{L}', \mathcal{G}') \longrightarrow P(\mathcal{L}'', \mathcal{G}''))$
  $\longrightarrow P(\mathcal{L}', \mathcal{G}')$

```
lemma ballot_induct:
  assumes valid: "valid_strict_ballot ℒ 𝒢"
      and major:
        "⋀ℒ' 𝒢'. ⟦valid_strict_ballot ℒ' 𝒢';
                  ⋀ℒ'' 𝒢''. subballot ℒ'' 𝒢'' ℒ' 𝒢' ⟹ P ℒ'' 𝒢''⟧
                  ⟹ P ℒ 𝒢"
  shows "P ℒ 𝒢"
```

Figure 4.8: The ballot induction rule in Isabelle/HOL.

The form of our ballot induction lemma is analogous to `finite_psubset_induct` (see Figure 4.4), except in place of finiteness we require validity (which implies finiteness for this induction rule to work), and in place of the proper subset relation we instead have the proper subballot relation. We will see how one goes about using this in the proofs of later theorems.

## 4.5 Directly above and below candidates on ballots

In this section we briefly cover two predicates characterising candidates as *directly above* or *directly below* another candidate on a ballot. These two predicates were originally more key to our proof strategy for rankings, but this became less the case as our formalisation progressed.

Our proof plan originally involved using induction to show that the listed candidates spanned the range of ranks from $\{0..\mathcal{L}-1\}$: when inserting a new candidate, there is a unique candidate above or below them, use this to show no other candidate shares their rank, combine with the facts that those listed above will have the same rank as before insertion and those below will have rank one more. This would have probably worked, but our actual approach is somewhat simpler (see Section 4.8) and simply came about naturally in the course of proving basic results.

Nevertheless, there are a few results proven using these predicates that we use as the basis for uniqueness results for rankings, which are themselves then used to prove the key results needed for the votes invariant.

**Definition 3** (Directly above)**.** *A candidate $c$ is directly above another $c'$ on a ballot $(\mathcal{L},\mathcal{G})$ iff*

- *$c$ and $c'$ are listed: $c,c' \in \mathcal{L}$*

- *The candidates listed greater than $c'$ are exactly those listed greater than $c$ plus $c$ itself:*

$$\mathcal{G}(c') = \mathcal{G}(c) \cup \{c\}$$

**Definition 4** (Directly below)**.** *A candidate $c$ is directly below another $c'$ on a ballot $(\mathcal{L}, \mathcal{G})$ iff*

- *$c$ and $c'$ are listed: $c, c' \in \mathcal{L}$*
- *The candidates listed lesser than $c$ plus $c$ itself are exactly those listed (strictly) lesser than $c'$:*

$$\mathcal{L} - \mathcal{G}(c) = (\mathcal{L} - \mathcal{G}(c')) - \{c'\}$$

To get a sense for the definitions, consider the ballot *abcde*. We can see that candidate $b$ is listed directly above candidate $c$ because they are both listed and the candidates listed greater than $c$ ($= \{a, b\}$) are exactly those listed greater than $b$ ($= \{a\}$) plus $b$ ($= \{a, b\}$). In reverse, we can see that candidate $c$ is listed directly below candidate $b$ because they are both listed and the candidates listed lesser than $c$ ($= \{d, e\}$) plus $c$ itself ($= \{c, d, e\}$) are exactly those listed lesser than $b$ ($= \{c, d, e\}$).

In Isabelle/HOL we write this as follows, using a special notation to ease having $\mathcal{L}$ and $\mathcal{G}$ as additional parameters:

```
definition above where
  "c above[ℒ,𝒢] c' ≡ c' ∈ ℒ ∧ insert c (𝒢 c) = 𝒢 c'"

definition below where
  "c below[ℒ,𝒢] c' ≡ c ∈ ℒ ∧ c' ∈ ℒ ∧ ℒ - 𝒢 c = (ℒ - 𝒢 c') - {c'}"
```

Figure 4.9: The predicates for “is directly above” and “is directly below” in Isabelle/HOL.

For these definitions, we only include what is necessary in the context of a valid ballot as this is the only context for their practical use. The following lemma provides reassurance that our definition of `above`, which does not require $c$ to be listed, is correct:

```
lemma above_in1:
  assumes "c above[ℒ,𝒢] c'"
      and "valid_strict_ballot ℒ 𝒢"
  shows "c ∈ ℒ"
```

Figure 4.10: Lemma demonstrating that the definition for `above` is sufficient in the context of a valid ballot.

We follow the definitions of these predicates with numerous other lemmas e.g. about reflexivity, symmetry, and transitivity results that one would expect from successor-predecessor predicates such as these. We are then able to prove a number of uniqueness results along the lines of `ex_unique_above` listed below.

```
lemma ex_unique_above:
  assumes valid: "valid_strict_ballot ℒ 𝒢"
      and "c ∈ ℒ"
      and "𝒢 c ≠ {}"
  shows "∃!k. k above[ℒ,𝒢] c"
```

Figure 4.11: Lemma stating that for every candidate which is not the top-most listed candidate on a ballot, there exists a unique candidate directly above them.

## 4.6 Uniqueness results and applying ballot induction

We will use `unique_last_pref` to demonstrate the use of ballot induction for a lemma that is core to our results about rankings. This states that there is a candidate listed last, and that this candidate is unique on the ballot:

```
lemma unique_last_pref:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "ℒ ≠ {}"
  shows "∃!c∈ℒ. 𝒢 c = ℒ - {c}"
```

Figure 4.12: There is a unique last-listed candidate on any non-empty valid ballot.

The proof of this lemma follows a common pattern of ballot induction and so we will briefly dwell on it. We begin by opening up the induction by inducting on $\mathcal{L}, \mathcal{G}$ as a ballot.[2] Inside the induction proof, we have two newly fixed $\mathcal{L}, \mathcal{G}$, which by the induction premises we know represents a valid non-empty ballot, and by the induction hypothesis (IH) we know that:

$$\forall \mathcal{L}'\ \mathcal{G}'.\ subballot(\mathcal{L}', \mathcal{G}', \mathcal{L}, \mathcal{G}) \wedge \mathcal{L}' \neq \{\} \longrightarrow \exists! c.\ c \in \mathcal{L}' \wedge \mathcal{G}'(c) = \mathcal{L}' - \{c\}$$

To apply the IH, we thus need a non-empty subballot. The easiest way to get one is by removing a single candidate from the ballot. We thus obtain some arbitrary, fixed $c'$ and $\mathcal{L}', \mathcal{G}'$ which satisfy:

- $\mathcal{L} = \mathcal{L}' \cup \{c\}, c' \notin \mathcal{L}'$
- $subballot(\mathcal{L}', \mathcal{G}', \mathcal{L}, \mathcal{G})$

A common bit of boilerplate is at this point to show that this obtained $\mathcal{G}'$ satisfies:

$$\forall c \in \mathcal{L}'.\ \mathcal{G}'(c) = \mathcal{G}(c) - \{c'\}$$

In other words, we define it, based on the fact that we know the definition of $\mathcal{L}'$, being $\mathcal{L}$ with $c'$ removed, and the fact about the pair being a subballot of the original ballot. In the proof of this particular lemma we then case split on whether $\mathcal{L}' = \{\}$, another common next step, especially if a premise of the lemma is ballot non-emptiness as then this case split is necessary to even apply the IH. For this particular lemma, if $\mathcal{L}' = \{\}$, we can conclude that $\mathcal{L} = \{c'\}$ and clearly $c'$ is our unique, last-listed candidate.

[2] In Isabelle: `induct ℒ 𝒢 rule: ballot_induct`

If $\mathcal{L}' \neq \{\}$, then we can apply the induction hypothesis to obtain a candidate $c$ which is the unique last-listed candidate on the subballot. We know that $c' \neq c$ because $c$ is listed on $\mathcal{L}'$ and $c'$ is not. We can show that it must be the case that either $c$ remains the last-listed candidate on the larger ballot (one could say "after inserting $c'$"), and if not then this must now be $c'$. If $c$ remains the last-listed then it must still be unique because it was unique on the subballot plus, using the fact that ballots are total orders, we can use the fact that $c' \neq c$, and for $c$ to be last it must be that $c \notin \mathcal{G}(c')$, to conclude $c' \in \mathcal{G}(c)$. Given $c'$ is greater than some candidate (specifically, $c$), the expression characterising last-ness does not apply to $c'$. Similar reasoning applies for the case where $c'$ is last on the larger ballot, and this completes the proof.

## 4.7 Pure ordering-specific properties and automation

Before moving to natural-number rankings, we will discuss the general lemma `ex_greatest_sat_cand`. This states that for any subset of a ballot, if it contains at least one candidate satisfying a property, then there is a maximally-listed candidate in that subset satisfying said property. It is an important lemma that comes up in the proof of the votes invariant (see Section 5.5.3). This is because most of the component functions of Meek's method (see Section 5.2) are implemented in such a way that hopeful candidates become important for analysis. In particular, if we can find a maximally-listed *hopeful* candidate $h$ on some ballot, then we know that none of that ballot's mass goes to any candidates listed less than $h$, as $h$ receives whatever is left of the ballot after those listed above them have taken their share (apportioned by the weights).

**Theorem** (Maximal listed candidate satisfying a property)**.** *If $(\mathcal{L}, \mathcal{G})$ is a valid ballot, $X \subseteq \mathcal{L}$, and there is some $x \in X$ such that $P(x)$, then there exists a unique $y \in X$ such that $P(y)$ and all $z \in X$ where $z \neq y \wedge P(z)$ are listed less than $y$.*

```
lemma ex_greatest_sat_cand:
  assumes valid: "valid_strict_ballot ℒ 𝒢"
      and Xsub: "X ⊆ ℒ"
      and ex: "∃x∈X. P x"
  shows "∃!x∈X. P x ∧ (∀y∈X. y ≠ x ∧ P y ⟶ x ∈ 𝒢 y)"
```

Figure 4.13: If there is a candidate on a ballot satisfying a property, there is a unique, highest-listed candidate satisfying that property.

This could be reframed so that it is only stated over $\mathcal{L}$, with subset membership being subsumed by the property $P$, but this is a handy form to have it in.

Consider the case when $P = (\lambda c.w_c = 1)$ and $X = \mathcal{G}(k)$, and we have some $c \in \mathcal{G}(k)$ where $w_c = 1$. That is, $c$ is a hopeful candidate listed greater than $k$. We can then obtain a $c'$ greater than $k$ such that $w_{c'} = 1$, and all other hopeful candidates must be listed less than $c'$.[3] Importantly, as $c'$ is the top-listed hopeful candidate, we *know* that they must get at least some votes, as there is nobody listed above them that is hopeful (i.e. nobody that can "suck up" everything coming to them), and everybody below them must get no votes. We can use this to simplify, for example, sums over votes.

## 4.8 Rankings

We finish this discussion of ballots by introducing functions for extracting and stating the rankings of candidates expressed as natural numbers.

**Definition 5** (Candidate rank)**.** *A candidate c has rank n, rank*$(c) = n$*, iff there are n candidates listed greater than c on the ballot.*

**Definition 6** (Ranked $n^{th}$)**.** *The candidate ranked* $n^{th}$ *on a ballot, ranked*$(n)$*, is the unique candidate listed on the ballot who has n candidates listed greater than them.*

The candidates listed greater than some candidate $c$ is of course the set $\mathcal{G}(c)$, and counting how many candidates are listed greater than $c$ is thus equivalent to taking the cardinality: $|\mathcal{G}(c)|$. Equating candidate rank with cardinality means, unlike in real-world elections, our "highest rank" is 0, not 1.

Indeed, while the ordinal "first" is the usual view of the top-ranked candidate, as the first on the list, and ordinal "first" is often associated with the cardinal 1, it is not necessary for these to match. Indeed, in computer science and mathematical settings it is often much more natural to start from 0. Here this has an intuitive interpretation which is equivalent to its actual implementation, as the number of those listed greater on the ballot; one can read traditional list-indexing in computer science the same way, the index of an item in a list is the number of items preceding it in the list.

So we have *rank* and *ranked*, the latter of which is merely the inverse of *rank*. There is little else to say about these two notions when viewed abstractly, though the Isabelle versions merit some further remarks (besides the fact that we have to explicitly carry around $\mathcal{L}$ and $\mathcal{G}$):

[3] It may or not be the case that $c' = c$.

```
definition rank where
  "rank 𝒢 c ≡ card (𝒢 c)"

definition ranked where
  "ranked ℒ 𝒢 n ≡ SOME c. c ∈ ℒ ∧ rank 𝒢 c = n"
```

Figure 4.14: Functions for working with natural-number rankings on ballots.

Notice that we do not define `ranked` as the inverse of `rank`, which would be:

$$\texttt{ranked}\ \mathcal{G}\ \texttt{n} \equiv\ \texttt{inv (rank}\ \mathcal{G}\texttt{) n}$$

This does not work because, in Isabelle, it reduces to `SOME c. card (`$\mathcal{G}$` c) = n`, which can pick candidates both in $\mathcal{L}$ and outside $\mathcal{L}$, and breaks the uniqueness result that only one candidate satisfies the predicate $(\lambda c.\ |\mathcal{G}(c)| = n)$ for any given $n < |\mathcal{L}|$. That is, we need to explicitly restrict the domain of the choice operator to $\mathcal{L}$, which does not happen by simple inverse.

We could use either `THE` or `SOME` here in our definition, but it is best practice if one is presuming that it is possible to prove uniqueness to not embed this in the definition, and instead prove the equivalence. We will come back to this equivalence shortly in Section 4.8.2.

### 4.8.1 Sanitising the terminology

As with real-world elections, the lower a candidate's natural-number ranking, the higher-up they are listed on the ballot. This can occasionally be confusing, especially when parsing the mathematics versus the theorem name, and so we have endeavoured to stick to a consistent vocabulary, which for clarity we outline here along with the associated mathematical formulae:

- `c` is listed greater than `c'` $\longleftrightarrow$ `c ∈ `$\mathcal{G}$` c'`
- `c` is listed lesser than `c'` $\longleftrightarrow$ `c ∈ `$\mathcal{L}$` - `$\mathcal{G}$` c' - {c'}`
- `c` is ranked higher than `c'` $\longleftrightarrow$ `rank `$\mathcal{G}$` c < rank `$\mathcal{G}$` c'`
- `c` is listed (directly) above `c'` $\longleftrightarrow$ `c above[`$\mathcal{L}$`,`$\mathcal{G}$`] c'`
- `c` is listed (directly) below `c'` $\longleftrightarrow$ `c below[`$\mathcal{L}$`,`$\mathcal{G}$`] c'`

In particular, "ranked higher/listed greater" is chosen over "ranked greater/listed higher" as this seems to more closely match real-world terminology. Likewise "ranked lower" is chosen if one wishes to view the same fact from the other direction, from the point of view of $c'$. The terminology is also clarified through the relationship in the following theorem:

```
lemma greater_higher_ranked:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "c' ∈ ℒ"
      and "c ∈ 𝒢 c'"
  shows "rank 𝒢 c < rank 𝒢 c'"
```

Figure 4.15: A lemma for illustrating the terminology used.

Candidate $c$ is *ranked higher* than $c'$, meaning their natural-number ranking is *less*. We also say $c$ is *listed greater* than $c'$. Finally, $c$ may or may not be listed *above* $c'$.

Note again that we develop both the set-based formulation and the formulation based on ranking (derived from the set-based one) because *both* are necessary for the formal development; this work is not just here as additional work which may be useful to others but not us.

### 4.8.2 Uniqueness of *ranked*

The uniqueness of *ranked* can be stated in Isabelle/HOL like so:

```
lemma ranked_THE:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "n < card ℒ"
  shows "ranked ℒ 𝒢 n = (THE c. c ∈ ℒ ∧ rank 𝒢 c = n)"
```

Figure 4.16: Uniqueness of *ranked*.

Proving it is a direct application of Isabelle's standard library lemma `the_equality`, which we can use to reframe what we need to prove like so:

**Theorem** (Uniqueness of *ranked*)**.** *Given a valid ballot $(\mathcal{L}, \mathcal{G})$ and a position on the ballot $n < |\mathcal{L}|$, there is a unique listed candidate at that position. Equivalently, given validity and $n < |\mathcal{L}|$, it is the case that the following are true:*

- *The candidate ranked $n^{th}$ is indeed actually a listed candidate: $ranked(n) \in \mathcal{L}$*
- *The rank of the candidate ranked $n^{th}$ is $n$: $rank(ranked(n)) = n$*

- *For each listed candidate, if their rank is n, they must also be the $n^{th}$ ranked candidate:* $\forall c \in \mathcal{L}.\ rank(c) = n \longrightarrow ranked(n) = c$

These are three of a larger collection of key results for connecting the functions *rank* and *ranked*, and are what allows us to focus on proving numerous results about *rank* and then connect these to results about *ranked* using connective glue such as `ranked_THE` and the various results it depends on.

It is, in the end, the function *ranked* which is more important for us, as it provides a *sequence* into the ballot representation that is very useful for the proof of the votes invariant (Section 5.5.3). But key results about *ranked*, including very basic ones like `ranked_THE`, are not provable until we have a number of these key results about *rank*, which we will go through some of now.

### 4.8.3 Key properties of *rank*

Defining *rank* directly in terms of cardinality allows us to use everything we have already proven about ballots prior to the treatment of rankings, and also make use of the bounty of set-theoretical results which already exist in Isabelle/HOL.

For example, earlier on (see Section 4.6) we discussed `unique_last_pref`, a lemma which expresses the fact that there is unique candidate which is listed last. We can trivially translate this from a set-based lemma into one based on rankings and prove a fact which states that there is a unique candidate which is *ranked lowest*:[4]

```
lemma unique_rank_lowest:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "ℒ ≠ {}"
  shows "∃!c∈ℒ. rank 𝒢 c = card ℒ - 1"
```

The fact that this is provable so easily using Isabelle/HOL's automated proof tools supports the claim that the set-based and rank-based representations are sufficiently well-connected by existing lemmas. We also prove when removing a candidate affects (or not) another candidate's ranking, which is important for case splits later:[5]

**Theorem** (Removing or inserting lesser listed candidates leaves rank the same)**.** *If $(\mathcal{L}', \mathcal{G}')$ is a subballot of $(\mathcal{L}' \cup \{c\}, \mathcal{G})$, and k is listed greater than c, then $rank(k)$ is equal on both ballots.*

[4] One could also use the terminology "ranked last" here.

[5] Note that when asserting facts like $subballot(\mathcal{L}', \mathcal{G}', \mathcal{L}' \cup \{c\}, \mathcal{G})$, whether this holds depends only on whether $\mathcal{G}'$ is a sub-greater-than function of $\mathcal{G}$, as $\mathcal{L}'$ is necessarily a subset of $\mathcal{L}' \cup \{c\}$.

One writes theorems such as these in Isabelle as follows:

```
lemma rank_insert_same:
  assumes "valid_strict_ballot (insert c ℒ') 𝒢"
      and "subballot ℒ' 𝒢' (insert c ℒ') 𝒢"
      and "k ∈ 𝒢 c"
  shows "rank 𝒢 k = rank 𝒢' k"
```

where statements such as "the rank of $k$ on $\mathcal{G}$" or "the rank of $k$ on both ballots" is of coursed expressed by passing $\mathcal{G}$ directly to the *rank* function.

Moving on to the focus of this section, we can isolate two important results.

**Theorem** (There are lower-ranked candidates for all lower rankings)**.** *For any given candidate $c$ listed on a ballot $(\mathcal{L}, \mathcal{G})$, and any rank $m$ lower than or at $c$ on the ballot, $rank(c) \leq m < |\mathcal{L}|$, there is a candidate $k$ such that:*

- *$k$ is listed: $k \in \mathcal{L}$*
- *Either $k$ is itself $c$, or $k$ is listed lesser than $c$: $k \notin \mathcal{G}(c)$*
- *The rank of $k$ is $m$: $rank(k) = m$*

Consider the case where $rank(c) = 0$, then for every rank $m \in \{0..|\mathcal{L} - 1|\}$ there is some candidate with that rank. In other words, *rank* in the domain $\mathcal{L}$ is surjective on $\{0..|\mathcal{L} - 1|\}$. This result thus conceptually leads directly to the next one:[6]

**Theorem** (The rankings of those lesser are unique and contiguous)**.** *If $c$ is a candidate listed on the ballot $(\mathcal{L}, \mathcal{G})$, then the rank of $c$, with the rank of the candidate below $c$, and so on, are exactly the natural numbers $rank(c)$ through to $|\mathcal{L}| - 1$, i.e.:*

$$rank(\mathcal{L} - \mathcal{G}(c)) = \{rank(c)..|\mathcal{L}| - 1\}$$

Here *rank* is applied to a set, overloaded in the conventional way functions are in pen-and-paper mathematics. The second lemma has a very important corollary: if $c$ is the top-ranked candidate then $\mathcal{G}(c) = \{\}$ and $rank(c) = 0$, and so[7]

$$rank(\mathcal{L}) = \{0..|\mathcal{L}| - 1\} \tag{4.1}$$

That is, every rank from 0 through to one less than the size of the ballot is assigned to *some* candidate on the ballot. Indeed, we can prove thank *rank* is bijective between $\mathcal{L}$

[6] In actuality, the next result follows from different although similar sources.

[7] We originally proved this fact in a 200-line ballot induction proof not dissimilar from the proofs that appear in Section 5.5.3 concerning the votes invariant. We do not discuss this proof here because of this similarity and also because later results actually made it redundant.

and $\{0..|\mathcal{L}|-1\}$. This is very close to one of the essential facts we need for *ranked*, discussed in the following section.

### 4.8.4 Key properties of *ranked*

After proving that *rank* is a bijection, several uniqueness results follow, such as there being a unique candidate for every rank and a unique rank for every candidate. However, we will not elaborate further on these as they are basic results which follow immediately from the bijectivity of *rank* on $\mathcal{L}$. The key batch of lemmas for *ranked* are ones of the form $rank(ranked(n)) = n$, if $n$ is in the proper range. In Isabelle/HOL:

```
lemma rank_of_ranked:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "n < card ℒ"
  shows "rank 𝒢 (ranked ℒ 𝒢 n) = n"
```

Figure 4.17: One of a handful of "glue lemmas" connecting *rank* and *ranked*.

We have already shown that if $n < |\mathcal{L}|$, $ranked(n)$ must be listed, and we have also shown that listed candidates have unique ranks. We have that $rank(ranked(n)) = n$ as above, also that $ranked(rank(c)) = c$ if $c$ is listed. Lemmas such as `ranked_THE` as we discussed earlier can now be proven. For example the following fact from earlier which was needed to prove it:

*Given $n < |\mathcal{L}|$, for each listed candidate, if their rank is n, they must also be the $n^{th}$ ranked candidate*: $\forall c \in \mathcal{L}.\ rank(c) = n \longrightarrow ranked(n) = c$

*Proof.* Fix $c \in \mathcal{L}$ with $rank(c) = n$. Immediately, we have $c = ranked(rank(c)) = ranked(n)$. □

Finally, there are the three results which will be used in the proof of the votes invariant in Section 5.5.3. First, that the candidate ranked top (when such a candidate exists), in $0^{th}$ position, has nobody listed above them:

```
lemma G_ranked_0_empty:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "ℒ ≠ {}"
  shows "𝒢 (ranked ℒ 𝒢 0) = {}"
```

Figure 4.18: The first and most trivial of three key results for *ranked*.

Second, those ranked higher than the $n^{th}$ candidate are the same as those ranked 0 through to $n-1$.

```
lemma G_ranked_n_eq_ranked:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "0 < n"
      and "n < card ℒ"
  shows "𝒢 (ranked ℒ 𝒢 n) = ranked ℒ 𝒢 ` {..n - 1}"
```

Figure 4.19: The second of three key results for *ranked*.

Third and finally, those ranked from 0 through to one less than the rank of $c$ are exactly those listed greater than $c$:

```
lemma valid_ranked_image_gt:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "c ∈ ℒ"
      and "rank 𝒢 c ≠ 0"
  shows "ranked ℒ 𝒢 ` {..rank 𝒢 c - 1} = 𝒢 c"
```

Figure 4.20: The third of three key results for *ranked*.

## 4.9 A minimal treatment of election context

What defines "an election" depends on one's purpose. It could encompass the whole process, from voting method (machine, paper, bead, etc) to vote counting to election auditing. For our purposes, however, we only care about the algorithm used to count the ballots, Meek's method, the input into that algorithm, and the tweakable parameters of the algorithm (e.g. the stopping parameter, ε).

The principal thing we will cover in this section is *election context*, which we define as being the input to an election along with its parameters.

### 4.9.1 The election context locale

The *inputs* to an election are only ever two things: the candidates running, and the submitted ballots, perhaps with additional metadata. A method may or may not have parameters, so in the general case we are only concerned with the inputs. We do not consider a situation where there are no ballots to count, i.e. when the set of ballots is empty. Note also that in this context we must fix an $\mathcal{L}$ and $\mathcal{G}$ which now each take a ballot as an initial argument:

**Definition 7** (Election context)**.** *An election context is a set of candidates $C$ and a set of ballots $B$ along with ballot accessors $\mathcal{L}$ and $\mathcal{G}$ satisfying the following:*

- *Each ballot is valid:* $\forall b \in B.\ valid(\mathcal{L}(b), \mathcal{G}(b))$
- *There is at least one candidate listed on each ballot:* $\forall b \in B.\ \mathcal{L}(b) \neq \{\}$
- *There is at least one ballot:* $B \neq \{\}$
- *There are finitely many ballots:* $finite(B)$
- *There are finitely many candidates:* $finite(C)$
- *Only candidates are listed on ballots:* $\bigcup\{\mathcal{L}(b).\ b \in B\} \subseteq C$

Recall that the function *valid* takes the set of candidates listed on a ballot and the greater-than function for that *specific ballot* directly as arguments, so we have to pre-apply them with the ballot $b$. The Isabelle/HOL translation of this definition of election context contains no surprises, but we present it for completeness' sake:

```
locale election_context =
  fixes cands :: "'c set"
    and ballots :: "'b set"
    and ℒ :: "'b ⇒ 'c set"
    and 𝒢 :: "'b ⇒ 'c ⇒ 'c set"
    and listed :: "'c set"
  defines listed_def: "listed ≡ ⋃{ℒ b |b. b ∈ ballots}"
  assumes all_valid: "b ∈ ballots ⟹
                      valid_strict_ballot (ℒ b) (𝒢 b)"
      and all_nonempty: "b ∈ ballots ⟹ ℒ b ≠ {}"
      and ballots_nonempty: "ballots ≠ {}"
      and finite_ballots: "finite ballots"
      and finite_cands: "finite cands"
      and listed_cands: "listed ⊆ cands"
```

Figure 4.21: The locale capturing election context.

There is a choice regarding representation here. If listed candidates are the only ones that are relevant, given that any candidate not listed on any ballot may as well not be standing, one might be tempted to just draw an equivalence between the set of *listed* candidates to be the set of *running* candidates. There are two related reasons not to do so.

First, it is bad practice to exclude an aspect of representation that one *assumes* will have no real effect on theoretical results, even if it seems obvious. One should not presume what one can prove. For a simple example, we can prove, rather than assume, that we do not need to consider unlisted candidates when summing or taking products over the types of functions that are common to the contexts in which we take sums and products:

```
lemma sum_cands_listed:
  assumes "⋀c. c ∈ cands ⟹ f c = g listed c"
      and "⋀X c. c ∉ X ⟹ g X c = 0"
  shows "(Σc∈cands. f c) = (Σc∈listed. f c)"

lemma prod_cands_listed:
  assumes "⋀c. c ∈ cands ⟹ f c = g listed c"
      and "⋀X c. c ∉ X ⟹ g X c = 1"
  shows "(Πc∈cands. f c) = (Πc∈listed. f c)"
```

Figure 4.22: Sums and products over candidates can be reduced to sums and products over listed candidates.

Second, it is bad practice to exclude an aspect of representation for minor efficiency gains, by whatever measure of efficiency (proof length, number of variables, etc), if doing so negatively affects representational and proof intuition or if it results in a representation that unnecessarily strays from what is expected or typical.

We exclude empty ballots as they are not theoretically interesting outside of the notion of spoiled ballots and tactical voting, though this locale could indeed be generalised to cover empty ballots without too much additional pain. Such a generalisation would force a great deal of lemmas to consider whether a ballot is empty or not, adding conceptually shallow visual noise. Furthermore, to fully integrate empty ballots into the rest of the formalisation, it would either force implementations to begin by "filtering out" spoiled ballots (including empty ones), or consider them immediately exhausted.

On the other hand, in principle Meek's method is the STV method best suited to handling empty ballots given it uses a dynamic quota, where throwing a large number of ballots away immediately would not be an issue in the way it would be with a static quota. Nevertheless, we have chosen to maintain exclusion of empty ballots for the duration of the thesis.

### 4.9.2 Working within an election context permits more elegant notation

Having $\mathcal{L}$ and $\mathcal{G}$ as fixed constants available throughout the locale `election_context`[8] allows us to define notation that leaves them implicit in that context. For instance:

[8] As opposed to previously where they were fixed constants introduced for each top-level lemma in the ballots theory.

```
definition ballot_eq where
  "x =ᵦ y ≡ x = y ∧ x ∈ ℒ b ∧ y ∈ ℒ b"

definition ballot_gt where
  "x >ᵦ y ≡ x ∈ ℒ b ∧ y ∈ ℒ b ∧ x ∈ 𝒢 b y"

definition ranked_first where
  "ranked_first b y ≡ ∀x∈ℒ b. x ≤ᵦ y"
```

Working with a more elegant notation like this not only improves readability, but can serve as a much better basis for priming one's intuitions about viable approaches to proof and necessary theorems; the latter is particularly the case with symbolic, infix notation that more readily suggests to one's intuition various results regarding symmetry, transitivity, and so on.

### 4.9.3 Lifting statements to the higher level by connecting definitions

The definitions introduced in the previous section hide away what was previously exposed, particularly $\mathcal{G}$. In order to facilitate proof purely at the level of this more intuitive notation, we introduce a handful of lemmas along the lines of the following.

```
lemma gt_compr_𝒢:
  assumes "b ∈ ballots"
      and "k ∈ ℒ b"
  shows "{c. c >ᵦ k} = 𝒢 b k"
```

Figure 4.23: The greater-than predicate generates the greater-than function.

All of these of course follow trivially by basic set-theoretical results.

## 4.10 Related work

Any work which represents a voting method will typically develop some representation for ballots and election context. In more general social choice settings a ballot is called a *preference ordering*, and a set of ballots a *preference profile*. Mapping from the preference profile to produce a *social ordering*, which fairly represents an aggregation of the individual preference orderings, is the goal of voting theory.

Ghale et al. represent ballots using a list of candidates as well as a fractional value associated with the ballot. This fractional value is directly associated with the ballot

because they implement a procedural approach to counting votes whereby candidates have, at various stages, piles of fractional ballots currently assigned to them which they pass on only partial amounts of according to a "transfer value" (re)computed by considering the surplus of the candidate this fraction of the ballot is currently assigned to, a kind of ad-hoc candidate weight. We discussed the list approach to representing ballots in Section 4.2.1.2 alongside further discussion of the approach we have actually used and other potential representations.

Eberl's work in randomised social choice theory [21, 20], which formalises some key results in Isabelle/HOL, represents ballots as *preference relations*, each relation $\succeq_i$ being associated with some candidate $i$, permitting indifference or equal-preference or "non-strict ballots" in our terminology. This is an appropriate representation for a social choice setting where most results directly concern this binary relation, for example: if every one of the $m$ agents[9] prefers alternative $x$ to alternative $y$, which we can write $(\forall i \leq m.\ x \succeq_i y)$, the social ordering $(\succeq)$ should prefer $x$ to $y$, i.e. $x \succeq y$.

Finally, Nipkow [60] formalises the two most seminal impossibility theorems in social choice theory in Isabelle/HOL: Arrow's Impossibility Theorem [4] and the Gibbard-Satterthwaite Theorem [28]. They note that preference profiles are total preorders, but represent them using a utility function, a mapping from alternatives to real numbers. We have not discussed this approach in the thesis. This is a representation which is appropriate for a general setting, especially one in which representing 'underlying' voter preference, including intensity of preference, is relevant. One can use the utility function as a proxy for a total preorder over alternatives in the obvious way, and evaluate the socially chosen alternative using the utility function. A complication in applying this directly to STV is that in STV the preference profiles (ballots) *do not rank alternatives*. The alternatives in STV are, properly speaking, sets of elected candidates of size $S$ (the number of seats), and yet one does not rank one's preferred sets of elected candidates but individual candidates.

## 4.11 Conclusion

In this chapter we developed a representation for strict ranked ballots using a carrier set $\mathcal{L}$ along with a greater-than function $\mathcal{G}$ which takes a candidate on the ballot and returns the set of candidates listed greater than them. We developed an induction rule

[9] 'Agent' is a more general way to refer to what we have called a voter, though we have typically just referred to the ballot regardless of the preferences of some voter which produced it.

for ballots and some key results about rankings that will be necessary for the following chapter.

Our ballot induction rule utilises subballots and is analogous to the induction principle used for finite sets with an additional aspect to cover candidate ranking. We develop a handful of key results for these initial definitions, including the fact that subballots are valid ballots. Following this, we prove that on non-empty ballots, there are unique candidates listed first and last, and on ballots with at least two candidates, top-ranked and bottom-ranked candidates excluded, there are unique candidates listed directly above and directly below every candidate.

Importantly, we developed some definitions for representing and reasoning about rankings. We define a candidate's rank in terms of the number of candidates listed greater than them on the ballot, meaning the rankings start at 0 for the highest ranked candidate. We establish the equivalence between the ranking of a candidate and the candidate with that ranking using the functions *rank* and *ranked*. We prove a number of basic and important results about *rank* and about *ranked*, which are important for the interpretation in the next chapter. For example, $\mathit{ranked}(\{0..|\mathcal{L}-1|\}) = \mathcal{L}$.

It is possible that if we had focused on automation more up-front, while initially slowing development, this would have sped up development significantly enough to pay for itself. However, not getting the automation we hoped for from initial experiments posed a risk with respect to sinking time into getting this to work, with significant work required to finalise the results of Chapter 3 and Chapter 5 outstanding, and so we made the judgement to focus on proving the key results of this chapter and move on.

Finally, we developed a minimal wrapper around the ballots theory for capturing *election context*, within which we will provide the interpretation of the locales developed in Chapter 3, to follow now in the next chapter.

# Chapter 5

# Tying it all together: Meek's method concretely

In this final chapter covering the work done we cover our implementation of the components of Meek's method of STV, a specific surplus transfer round satisfying our abstraction in Chapter 3, as well as a general proof that the implementation satisfies our abstraction for all valid elections.

## 5.1 Overview

In this section we provide an overview of the chapter covering the aims, motivations and other key aspects when it comes to verifying Meek's algorithm concretely.

### 5.1.1 Aims and objectives

The aims of this chapter are to demonstrate that a typical implementation of Meek's method has a unique solution in the transfer round by discussing the verification of the correctness of the locales specified in Chapter 3. We then use this to demonstrate that for a specific election a solution to a transfer round, which we compute analytically within the theorem proving environment, is unique.

#### 5.1.1.1 Specific objectives

Our specific objectives are to concretise the proofs of correctness from Chapter 3 by:

- Implementing the component functions of Meek's method: the votes, quota, and excess.
- Proving these component functions, given a set of valid strict ballots and the usual weight update function, satisfy the abstract characterisation of Meek's method and the transfer round given in Chapter 3. As sub-objectives, this involves the following:
  - Proving a sizeable number of preliminary lemmas to facilitate interpretation of the locales.
  - Proving that each assumption of the locale characterising Meek's method at the top-level is satisfiable.
- Giving a specific example which we can solve analytically, showing that this is a solution given the concrete component functions, and then showing this solution is unique using the results of Chapter 3.

### 5.1.2 Motivation

By providing an interpretation of the locales from Chapter 3 we give evidence that the assumptions of the locale are:

1. Possible to prove for a concrete implementation, and hence logically consistent.

2. Satisfiable for a specific example using that concrete implementation without significant additional development, i.e. all the assumptions can be dispatched by invoking the more general lemmas for the specific example.

3. General, i.e. applicable to all valid elections.

### 5.1.3 Hypothesis and evaluation

Our hypothesis is that the three motivations given above – namely the possibility of giving an interpretation, the ease of use for specific examples, and the generality of the assumptions – can be formally demonstrated in Isabelle.

By actually proving each assumption holds for a concrete implementation, we show that such an exercise is possible, if perhaps onerous. By providing a model – in Isabelle/HOL parlance, an interpretation – we show the assumptions are consistent. By proving that an implementation – which only needs the additional context of a set of valid strict ballots – is a model of the locales, we show the assumptions apply to all valid elections.

### 5.1.4 Novel concepts and ideas

This chapter contains several novel contributions. First and foremost, we connect the proof of the uniqueness of the solution vector in non-trivial surplus transfer rounds to a concrete implementation, and thus open the door to verified, extracted code for Meek's method, something which has never previously existed.

Connecting these two aspects requires, of course, providing a functional implementation of Meek's method in Isabelle/HOL. No such functional implementation has been provided anywhere before as far as we are aware. Meek's method was implemented imperatively in Pascal [41] originally and later in Python in OpenSTV.[1] We believe the functional implementation of Meek's method reveals the most about its elegance as a

[1] OpenSTV is now the proprietary OpaVote [64].

method. Such an implementation also allows us to fill in a few gaps left by Woodall in the proofs that animated much of the previous chapter.

Finally, we prove the *votes invariant* for Meek's method, which all STV methods satisfy. Specifically, for Meek's method this reads:

$$\forall C\ B\ w.\ |B| = \Sigma_{c \in C} V_c(B, w) + E(B, w) \tag{5.1}$$

where $C$ is the set of candidates, $B$ is the set of ballots, $w$ is the weight vector i.e. the state in Meek's method, and $V$ and $E$ are the votes and excess functions. It states that the votes in circulation among the candidates, plus the excess votes, is constant and equal to the total number of ballots. See Section 5.5.3 for a full discussion of this invariant. Equivalently:

$$\forall C\ B.\ |B| = \Sigma_{c \in C} V_c(B, \mathbf{0}) + E(B, \mathbf{0}) = E(B, \mathbf{0}) \tag{5.2}$$

$$\forall C\ B\ w\ w'.\ \Sigma_{c \in C} V_c(B, w) + E(B, w) = \Sigma_{c \in C} V_c(B, w') + E(B, w') \tag{5.3}$$

Proving this invariant was not a trivial task (See Section 5.5.3).

### 5.1.5 Structure of this chapter

In Section 5.2 we provide an implementation of the votes, quota, and excess and discuss the representation decisions involved, in Section 5.3 we provide a number of preliminary proofs regarding the components, in Section 5.4 we interpret the locale representing Meek's method generally from Chapter 3, and finally in Section 5.5 we cover the outline of the proof of the votes invariant for Meek's method. We wrap up by covering formalisation size in Section 5.6, some related work in Section 5.7, and summarise in Section 5.8.

## 5.2 Closed-form expressions for components and implementation

In this section we will present a concrete implementation of the basic functional components of Meek's method. These components appeared in the previous chapter as the functions `V_for`, `Q_for`, `E_for` (see Section 3.3.2), whose only argument was the weight vector. Here, it will be necessary to also provide more concrete ballot, using the representation discussed throughout Chapter 4, introduced in Section 4.2.

As discussed in the overview (Section 5.1), nobody has previously presented the closed-form expressions of these components, though Hill et al. [41] probably used something like them in their proof implicitly. They are easy to derive, however, by considering how much each ballot contributes to the quantities in turn.

The resulting expressions are of course of a purely functional, mathematical nature, unlikely to be anything one would naturally produce in an ordinary implementation of Meek's method in a standard programming language without giving thought to ease of mathematical analysis. Meek's method, because the state can be simply given by the ballots and the weights, and because it need not explicitly track the "votes so far", "quota so far" etc., uniquely lends itself to closed-form expressions of this sort.

### 5.2.1 The fraction of a ballot going to a candidate

The votes a candidate receives is the sum of the fraction of each ballot they receive, multiplied by their weight (or *keep-value*). Thus, we begin by defining the function which returns the fraction of a ballot a particular candidate receives given a weight vector. We will briefly discuss non-strict ballots and other representational extensions using the same method of derivation as we use here as future work in Section 6.2.8.

On a particular ballot, each candidate $k$ receives a fraction of the ballot from the candidate above them, keeps the ballot mass received in proportion to their weight (aka keep-value) $w_k$, and passes on what remains, which is what they receive in proportion to $1 - w_k$. For example, if the ballot is $abcd$, the fraction of the ballot received by each candidate is as shown in Table 5.1. The first listed candidate of course initially receives the whole ballot. Candidate $b$ keeps $w_b$ of the $1 - w_a$ they receive, hence a simple multiplication of the two, and so on.

| Candidate | Fraction received | Fraction kept |
|---|---|---|
| $a$ | $1$ | $w_a$ |
| $b$ | $(1-w_a)$ | $(1-w_a)w_b$ |
| $c$ | $(1-w_a)(1-w_b)$ | $(1-w_a)(1-w_b)w_c$ |
| $d$ | $(1-w_a)(1-w_b)(1-w_c)$ | $(1-w_a)(1-w_b)(1-w_c)w_d$ |

Table 5.1: Fraction of the ballot $abcd$ received, kept, and hence transferred by each of the candidates $a, b, c, d$.

Any additional, unlisted candidate receives 0. It is easy to read off a closed-form

expression from this table:

$$\textit{frac-of}(b,c,w) = \begin{cases} \Pi_{k \in \mathcal{G}(b,c)}(1 - w_k) & c \in \mathcal{L}(b) \\ 0 & \textit{otherwise} \end{cases} \tag{5.4}$$

We have to condition on $c \in \mathcal{L}(b)$ (see Section 4.2 for a discussion of $\mathcal{L}, \mathcal{G}$) as the product over an empty set is by convention equal to 1. This convention also holds in Isabelle, hence the translation of this into Isabelle is direct, as follows:

```
definition frac_of :: "'c set ⇒ ('c ⇒ 'c set) ⇒ 'c ⇒ (real, '
    c::finite) vec ⇒ real" where
  "frac_of ℒ 𝒢 c w ≡ if c ∈ ℒ then Πk∈𝒢 c. (1 - w $ k) else 0"
```

Figure 5.1: Definition of *frac-of*.

In the next section, we discuss the difference between the pen-and-paper *frac-of* and its Isabelle formalisation `frac_of`.

#### 5.2.1.1 Remarks on $\mathcal{L}, \mathcal{G}$

In the mathematical listing for *frac-of* (Equation 5.4) we pass some ballot $b$, with $\mathcal{L}$ and $\mathcal{G}$ being available implicitly. In the Isabelle/HOL formalisation we require the carrier set and greater-than function representing the ballot to be passed explicitly, as for large parts of the theory on implementation there is no implicit $\mathcal{L}$ and $\mathcal{G}$ available. These definitions all exist at the top-level, unadorned by additional context. When using ballot accessors, i.e. an $\mathcal{L}$ and $\mathcal{G}$ which do not directly represent a ballot but require an abstract ballot to be passed to them, they need to be pre-applied in Isabelle/HOL: `frac_of (ℒ b) (𝒢 b) c w`.

As discussed in Section 4.2, sometimes $\mathcal{L}, \mathcal{G}$ do directly represent a ballot, as in the Isabelle implementation of *frac-of*, other times they take an additional ballot argument. We make flexible use of the notation for $\mathcal{L}, \mathcal{G}$ throughout, and it will be obvious how we are using it in context by the surrounding discussion and by whether $\mathcal{L}, \mathcal{G}$ take this additional argument.

We could have defined `frac_of` so that it took accessors $\mathcal{L}, \mathcal{G}$ and some ballot $b$ and applied the two functions to $b$, i.e. taking arguments `frac_of b ℒ 𝒢 c w`. This was indeed our original representation. However, in proofs, we regularly construct a new ballot by simply inserting or removing new candidates into some existing ballot $(\mathcal{L}, \mathcal{G})$. Doing this via some abstract ballot $b$ would at best introduce a number of syntactically messy indirections, and at worst make this impossible. The latter, because

fixing a pair of *accessors* $\mathcal{L}, \mathcal{G}$ sets the available individual ballots that are characterised with respect to some set of abstract ballots $B = \{b_1, b_2, ...\}$. That is, we would need new ballot accessors to represent a ballot not listed in $B$, or to pick values outside $B$ for each new ballot.

The final note worth remembered, is that this definition of *frac-of*, like all the others that will follow, puts *absolutely no requirements* on the arguments $\mathcal{L}, \mathcal{G}, w, c$. Although we label them the same here as we do in the context where $(\mathcal{L}, \mathcal{G})$ represent an actual ballot, they actually denote any set and function of the correct type. This is because this definition lives at the top-level of the Isabelle theory (rather than in a locale), without the surrounding context of any additional assumptions. Of course, the implementation does not make sense if $(\mathcal{L}, \mathcal{G})$ does not represent a strict ballot. It will only be later in the context of a set of valid, strict ballots (see Section 5.4) that we will then leverage the results of Chapter 3 and Chapter 4.

#### 5.2.1.2 Properties of *frac-of* observable by inspection

We can see by inspecting Equation 5.4 that if any candidate $k$ listed above $c$ has weight $w_k = 1$, then $\textit{frac-of}(b, c, w) = 0$. That is, if any candidate on the ballot is a *hopeful* candidate, that candidate receives any remaining votes coming from the candidates listed above them and passes nothing on, as expected.

We can also see that eliminated candidates listed above a candidate, with weight 0, contribute nothing to the votes the candidate receives, as we would expect. They are "passed over" by this function, as if they were not listed on the ballot to begin with. This equivalence between *elimination* and *not being listed* is, while trivial to notice here, an important property of Meek's method. See Section 5.3.2.

As long as $\textit{valid}(\mathcal{L}, \mathcal{G})$ is the case (see Section 4.2), the weight $w_c$ of the candidate itself is uninvolved in the votes $\textit{frac-of}(b, c, w)$ that it receives because a candidate cannot be listed above itself. Thus, it is immediately obvious that there is no "feedback loop" in passing around ballot mass, outside of incorrectly implemented imperative versions.

If any of the weights of candidates listed above another are greater than 1, candidates may receive "negative votes". This seems strange, but it all balances out (proven in Section 5.5.1). Any candidate with a weight greater than 1 can be seen as "taking vote mass" from lower-listed candidates and/or the excess, which is invalid in normal circumstances.

Similarly, candidates with negative weight pass on all the votes and even more in

proportion to their negative weight. If the top-listed candidate had negative weight, the candidate listed immediately below them would receive more than the mass usually received from a single ballot. While this seems a digression into trivial truths about a simple product over a simple expression, it will be important in Section 5.5, where we deal with the votes invariant, and the non-triviality of proving this "balancing out" formally is revealed.

### 5.2.2 The votes component

The votes a candidate $c$ receives *and keeps* from a ballot $b$, given weight vector $w$, is exactly the votes received multiplied by its keep-value, the weight $w_c$. It is the third column in Table 5.1. The votes a candidate receives in total is the sum of each mass of ballot received and kept over the whole set of ballots. We may thus write it, following Woodall's candidate-subscript notation for the votes $V$, as:

$$V_c(B,w) = w_c \Sigma_{b \in B} \textit{frac-of}(b,c,w) \tag{5.5}$$

Note that $V_c(\{\},w) = 0$, no matter the weights $w$ or the candidate $c$. In Isabelle we split this into two different definitions, to allow for alternate implementations of $f$ stemming from alternate representations of ballots or alternative formulations for the same ballot representation:

```
definition
 votes' :: "'b set ⇒ ('b ⇒ 'c ⇒ (real, 'c::finite) vec ⇒ real)
    ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite) vec" where
  "votes' B f w ≡ χ c. w $ c * (Σb∈B. f b c w)"

definition
 votes :: "'b set ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c ⇒ 'c set) ⇒
  (real, 'c::finite) vec ⇒ (real, 'c::finite) vec" where
  "votes B ℒ 𝒢 w ≡ votes' B (λb. frac_of (ℒ b) (𝒢 b)) w"
```

Figure 5.2: Definition of the votes function $V$.

This subscript notation, implying $V(B,w)$ is some sort of vector of votes, does also nicely mirror the Isabelle, in which we use the vector accessor function `$` to get the candidate, rather than simple function application: `votes B ℒ 𝒢 w $ c`.

### 5.2.3 The excess component

The excess is the total mass of exhausted votes. That is, one adds up the contribution each ballot makes to the excess, which is how much of it is exhausted. To get the amount of mass a single ballot exhausts, we simply extend Table 5.1 by one additional row (see Table 5.2). Again, it is easy to read-off a closed-form expression for this:

| Candidate | Fraction received | Fraction kept |
|---|---|---|
| $a$ | $1$ | $w_a$ |
| $b$ | $(1-w_a)$ | $(1-w_a)w_b$ |
| $c$ | $(1-w_a)(1-w_b)$ | $(1-w_a)(1-w_b)w_c$ |
| $d$ | $(1-w_a)(1-w_b)(1-w_c)$ | $(1-w_a)(1-w_b)(1-w_c)w_d$ |
| Excess | $(1-w_a)(1-w_b)(1-w_c)(1-w_d)$ | N/A |

Table 5.2: Fraction of the ballot $abcd$ received, kept, and hence transferred by each of the candidates $a,b,c,d$, plus the excess.

$$E(B,w) = \Sigma_{b\in B}\Pi_{c\in\mathcal{L}(b)}(1-w_c) \tag{5.6}$$

```
definition excess where
  "excess B ℒ w ≡ Σb∈B. Πc∈ℒ b. (1 - w $ c)"
```

Figure 5.3: Definition of the excess function $E$.

We can read off, by inspection, several facts about the excess from this, knowing nothing else about Meek's method:

- If a hopeful candidate is listed on every ballot, there is no excess.
- If every candidate is eliminated, this becomes $E(B,\mathbf{0}) = \Sigma_{b\in B}1 = |B|$, i.e. every ballot is completely exhausted.
- How much of each individual ballot is exhausted depends only on which candidates are listed, not their order (because multiplication is commutative). Hence, there is no *direct* link for voters between their ranking and how much of their ballot will be exhausted or "wasted".

- If every ballot lists every candidate in the full set of candidates $C$, the excess can be written $|B|\Pi_{c\in C}(1-w_c)$. In combination with the first point, if there is any hopeful candidate in such a situation, no ballot is at all exhausted or "wasted" until every candidate is either eliminated or elected, which only occurs in the final step of the process.

We believe that this is one reason why it would be pedagogically valuable to provide people with a more mathematical definition of Meek's method, as it more easily allows one to read-off its properties and mentally play with its components. This is unlike any existing implementation, which judging by the implementation in OpenSTV and the original Pascal respectively, leaves the algorithm feeling somewhat more opaque and messy than it is in reality.

### 5.2.4 The quota component

For the formulation of the quota we have a choice to make. Either directly here in the implementation, or later when we leverage the more abstract results. Every quota in general takes the following form[2]:

$$Q(B,S,w) = r\left(\frac{|B| - \delta_E E(B,w)}{S + c_1} + c_2\right) + c_3 \tag{5.7}$$

where $r$ is a rounding function, which potentially includes the identity function, $S$ is the number of available seats, and $c_1 \in \mathbb{N}_0$, $c_2, c_3 \in \mathbb{R}_{\geq 0}$. For the vast majority of STV methods, the excess plays no part in the quota, and we may set $\delta_E = 0$, otherwise $\delta_E = 1$. Of course, in other methods the state may be represented by something other than weights $w$ as in Meek's method.

For Meek's method the dynamic, fractional Droop quota (Equation 5.8) where $r = (\lambda x.\ x), \delta_E = 1, c_1 = 1, c_2 = 0, c_3 = 0$ is always chosen[3] in practice, and so we only concretely implement this quota:

$$Q(B,S,w) = \frac{|B| - E(B,w)}{S+1} \tag{5.8}$$

[2] As far as we are aware, based on the literature [50, 75, 59, 58] and various non-academic writing and implementations on the web [63].

[3] By far the most common quota is the static, integral Droop quota, where $r = (\lambda x.\ \lfloor x \rfloor), \delta_E = 0, c_1 = 1, c_2 = 0, c_3 = 1$. While we proved in Section 3.5.1 that the dynamic version (the static version makes no sense for Meek's method) of this quota is fine for proving the first major theorem (Theorem 1.1), it could not be used to prove the second major theorem (Theorem 1.2) in Section 3.5.2. This explains its absence from this thesis.

In Isabelle/HOL we split it up into the fractional quota taking simple values that can be directly computed and a version that takes the state and uses it to calculate the simple values needed:

```
definition quota_hb' :: "nat ⇒ nat ⇒ real ⇒ real" where
  "quota_hb' T s E ≡ (T - E) / (s + 1)"

definition quota_hb :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ (real, '
    c::finite) vec ⇒ real" where
  "quota_hb B s ℒ w ≡ quota_hb' (card B) s (excess B ℒ w)"
```

Figure 5.4: Definition for the quota function $Q$.

### 5.2.5 Whole method

While we do not verify any aspect of the functional implementation of Meek's method, we provide our Isabelle/HOL implementation here for completeness. We suppress type annotations here for readability. First, elimination:

```
definition smallests where
  "smallests v ≡ {c. ∀k. v $ c ≤ v $ k}"

definition eliminate where
  "eliminate B ℒ 𝒢 w ≡
     w<(SOME c. w $ c > 0 ∧ c ∈ smallests (votes B ℒ 𝒢 w)) ↦ 0>"
```

Figure 5.5: The elimination round.

An implementation of elimination which was concerned with code extraction and not merely ability to reason about the definition would have to remove this use of `SOME`. The best option would likely be to add an argument providing a function which picks a candidate, hence make the choice of how to implement that function a problem for elsewhere in the development, perhaps even after code extraction (so that one could just substitute, say, a Scala standard library random-selection function).

The surplus transfer round is implemented as follows (the prefix ε in the theorem names exists because we have also implemented versions of these functions which are non-computational in a similar way to `eliminate`):

```
abbreviation update_one where
  "update_one elected B s ℒ 𝒢 w ≡ χ c.
     if c ∈ elected then
       w $ c * quota_hb B s ℒ w / votes B ℒ 𝒢 w $ c
     else
       w $ c"

function ε_solve where
  "ε_solve ε elected B s ℒ 𝒢 w = (
     if surplus B s ℒ 𝒢 w > ε then
       ε_solve ε elected B s ℒ 𝒢 (update_one elected B s ℒ 𝒢 w)
     else
       w)"
```

Finally we have Meek's method, implemented as a recursive function:

```
function ε_meek where
  "ε_meek ε B s ℒ 𝒢 w = (let reaches = reaches_quota B s ℒ 𝒢 w in
    if card reaches ≥ s then
      w
    else if surplus B s ℒ 𝒢 w > ε then
      ε_meek ε B s ℒ 𝒢 (ε_solve ε reaches B s ℒ 𝒢 w)
    else
      ε_meek ε B s ℒ 𝒢 (eliminate B ℒ 𝒢 w)
    )"
```

## 5.3 Preliminary results

This section covers a handful of lemmas illustrative of the level of abstraction, difficulty of proof, and general form of the results in this part of the formalisation, leading up to the interpretation of the locales. The vast majority of these lemmas cover the function *frac-of*.

### 5.3.1 Basic facts about *frac-of*

All of these lemmas exist outside of any enclosing context, which is to say that there are no additional assumptions on any constants or functions besides those shown in the lemma statement. For example, we can prove that if a hopeful candidate $c$ is listed greater than a candidate $c'$, then $c'$ receives no votes:

```
lemma fraq_eq_has1:
  assumes "c ∈ 𝒢 c'"
      and "w $ c = 1"
  shows "frac_of ℒ 𝒢 c' w = 0"
```

Figure 5.6: Candidates listed less than a hopeful candidate on a ballot receive no votes.

The main thing to note about this is that we do not actually know anything about $\mathcal{G}$. It does not represent one part of a ballot here, because there is no such assumption `valid_strict_ballot` $\mathcal{L}$ $\mathcal{G}$ characterising it as such. However, many such lemmas follow from the definitions (e.g. `frac_of` in the current case) without additional assumptions.

The next result is a general lemma characterising `frac_of` when a single candidate's weight is changed, which follows largely by unfolding *frac-of*, simplification, and refolding:

```
lemma frac_eq_neq_if:
  assumes c'_in: "c' ∈ ℒ"
  shows "frac_of ℒ 𝒢 c' (w ⌊c,r⌋) =
              (if c ∈ 𝒢 c'
                 then if w $ c ≠ 1
                   then frac_of ℒ 𝒢 c' w * (1 + r / (1 - w $ c))
                   else r * (∏k∈𝒢 c'-{c}. 1 - w $ k)
                 else frac_of ℒ 𝒢 c' w)"
```

Figure 5.7: Lemma showing an expression for votes received from changing a single weight in terms of the original votes received.

We condition on whether $c$ is listed greater than $c'$ and whether $c$ is non-hopeful in order to get at the three most important cases regarding how the value changes. There is no point in making $c' \in \mathcal{L}$ part of the if-then-else, as the case where this is not true is trivially 0 by definition.

Note also that $c \neq c'$ is not needed, though we will have this fact immediately when $c \in \mathcal{G}\ c'$. This tells us that if the weight of $c$ is changed but is not listed above $c'$, the fraction of the ballot $c'$ receives is unchanged. This also applies when $c = c'$ because the weight of $c'$ has nothing to do with how much of a ballot it receives, only how much it passes on.

Otherwise, if $c$ is hopeful, $c'$ received 0 previously and so we cannot write an informative expression in terms of *frac-of*. Then we have the most important case where there is an elegant multiplicative relationship between the old value and the

new one, which is the case relevant for the locale assumption regarding how the votes change with one change to the weight. In ordinary mathematical notation, this would be written:

$$\text{if } c' \in \mathcal{L}(b), c \in \mathcal{G}(b,c'), w_c \neq 1, \text{ then}$$
$$\textit{frac-of}(b,c',(w \downharpoonright c, r)) = \textit{frac-of}(b,c',w)\left(1 + \frac{r}{1-w_c}\right)$$

We conclude with some remarks summarising the rest of the preliminary results. Consider the ballot $wx..yc..z$, where '..' stands for any number of other candidates. We can in a sense "redefine" the function `frac_of` by capturing its recursive nature, which may be read "$c$ gets whatever $y$ got multiplied by $1 - w_y$". This is the more intuitive way we understood this function when laying out the tables in earlier sections:

```
lemma frac_for_two:
  assumes y_directly: "𝒢 y = 𝒢 c - {y}"
      and y_in: "y ∈ ℒ"
      and y_gt: "y ∈ 𝒢 c"
      and c_in: "c ∈ ℒ"
  shows "frac_of ℒ 𝒢 c w = frac_of ℒ 𝒢 y w * (1 - w $ y)"
```

Figure 5.8: Lemma stating the *frac-of* function for a candidate in terms of the candidate listed directly above it.

Note the assumption `y_gt` implies $\mathcal{G}\ c \neq \{\}$, meaning $c$ is not ranked first on the ballot. The assumption `y_directly` should by read as "$y$ is ranked directly above $c$".

This lemma is exemplary of where covering non-strict ballots would obscure the underlying relationship and add unenlightening syntactic noise. The assumptions, like `y_directly`, would have to take into account that both $y$ and $c$ may (or may not!) be part of equally-listed subsets of the ballot, divisors would be introduced into the conclusion, without forcing the ballot to be valid we would we have to add more assumptions to make sure division by 0 is not occurring, and so on. We had originally begun by proving properties like this for both the strict and non-strict cases, and it was specifically the above lemma that led us to abandon this approach.

We also provide some connection between the ranking functions of Section 4.8 and this implementation:

```
lemma rank_0_all_frac:
  assumes "valid_strict_ballot ℒ 𝒢"
      and "c ∈ ℒ"
      and "rank 𝒢 c = 0"
  shows "frac_of ℒ 𝒢 c w = 1"
```

Figure 5.9: Lemma showing that the top-ranked candidate receives the full ballot.

A number of lemmas follow these basic facts, characterising when this function is non-negative, less-or-equal 1, and so on. There is nothing interesting to say about the majority of this, however, so we end our discussion here.

### 5.3.2 The principle of elimination

There are several ways in which one could formulate a *principle of elimination* for Meek's method, or STV generally. Meek used an informally stated *elimination principle* as one of two principles *generative* of the method [52, 51] and as discussed earlier (Section 1.1). There, he phrased is thus:

> **Principle 1.** If a candidate is eliminated, all ballots are treated *as if that candidate had never stood.*

The most important direct consequences of this are at the level of definitions and representation. An example of one such consequence comes up if a ballot lists entirely eliminated candidates: that ballot should be treated as empty or as having listed values not representing candidates, and so the ballot should not count to the total number of submitted ballots and the quota should in turn be lower than if it did count to the total. Though note that "should not count to the total number of submitted ballots" here just means that all the ballot does is increase the excess by 1, which is equivalent to reducing the number of ballots $|B|$ by 1; we are not saying that empty ballots should be treated as not counting towards $|B|$ in the strict sense of the mathematical calculation of these quantities, but rather conceptually speaking. As the excess in Meek's method represents the partially or totally discounted ballots, we can state this requirement as a theorem:

**Theorem** (Elimination principle for excess)**.** *Ballots which only list eliminated candidates should contribute their whole mass to the excess.*

```
lemma eliminated_ballots_increase_excess_by_number_eliminated:
  fixes Elim :: "'c::finite set"
  assumes finB: "finite B"
                "⋀b. b ∈ B⟹ finite (ℒ b)"
  shows "excess B ℒ (1<Elim⟼ 0>) = card {b∈B. ℒ b⊆ Elim}"
```

Figure 5.10: An elimination principle consequence stated in Isabelle/HOL by making use of the weight vector of all 1s.

Arguing in this way, Meek claims that any static quota violates this principle, as it implicitly treats ballots which only list eliminated candidates as still contributing to and involved in the count. According to the elimination principle this is no more reasonable than considering empty ballots submitted as inputs to the algorithm in the first place as valid (and thus contributing to a higher quota than if they had not been considered). This would be undesirable for lots of obvious reasons, including election manipulation.

Our main purpose in highlighting this is conceptual (and partly to provide context for a recommendation for future work in Section 6.2.9). It would be interesting in the context of a general representation of STV to investigate whether one could state this principle as a constraint on the allowable models of the more general abstraction. Neither the lemma presented above or the one below are necessary for our key results. It is however also likely that greater use of lemmas like the following one would prove useful for automatic simplification, as it removes otherwise necessary case splits to do with dividing or multiplying by 0 if eliminated candidates' weights come into expressions regarding votes.

**Theorem** (Elimination principle for votes)**.** *Removing eliminated candidates from every ballot which lists them, and removing ballots which only list eliminated candidates, does nothing to change the amount of votes each remaining candidate receives.*

```
lemma eliminated_candidates_equiv_unlisted_votes':
  assumes Elim_elim: "⋀c. c ∈ Elim ⟹ w $ c = 0"
      and BElim_B: "BElim ⊆ B"
      and BElim_all_elim: "⋀b. b ∈ BElim ⟹ ℒ b ⊆ Elim"
      and finB: "finite B"
  shows "votes B ℒ 𝒢 w $ c =
         votes (B - BElim) (λb. ℒ b - Elim)
               (λb c. 𝒢 b c - Elim) w $ c"
```

Figure 5.11: A consequence of the principle of elimination for the design, and hence properties, of the component for counting votes.

## 5.4 Connecting implementation and abstraction

In the next sections, it will be shown that our implementation satisfies – for any election, i.e. a valid set of non-empty finite ballots – the assumptions of the locale given in Section 3.3.2, and that a specific transfer round satisfies the locale given in Section 3.4. Before continuing on to the more general interpretation of the locale in 3.3.2, we first provide a concrete model of both locales using a specific example deliberately constructed to require an immediate transfer round. Note that the specific example (Section 5.4.1) uses numerous lemmas from the more general proofs in the sections that follow (developed in Section 5.4.2 onwards).

### 5.4.1 A fully concrete model

The "equations to be solved at each step" in a transfer round for each elected candidate $c$ is $V_c(w) = Q(w)$. We demonstrated an analytical approach to solving this in Section 1.1, and promised a formal verification of an associated example, which we now describe. This will also serve as a sanity check for the development in Chapter 3 as the formalisation of this example lives outside any surrounding *additional* context, and so only depends on the logical consistency of the Isabelle standard library. It will also help to ground what the various types and functions can look like in practice.

All in all, it takes around 250-300 lines to set up the example, show it implements a valid election, calculate the initial allocation of first preferences, show that $v_a = 4/13, v_b = 1/3$ is a solution to the transfer round, and finally show this solution is unique.

As a reminder, our example consisted of the set of ballots $B = \{ab, a, abc, bc, ba\}$ and number of seats $S = 2$, with a weight vector $w = \mathbf{1}$. First, we need to represent this

example formally in Isabelle/HOL.

```
datatype cand = Alice | Bob | Claire

lemma cand_UNIV:
  "(UNIV :: cand set) = {Alice, Bob, Claire}"
using cand.exhaust by blast

instance cand :: finite
proof
  show "finite (UNIV :: cand set)"
    by (simp add: cand_UNIV)
qed
```

As the `'c` type variable we have been using has to satisfy the sort constraint `finite`, we are forced to either define a new type with explicit constructors such as `cand` above, or introduce a new type which is a subset of the natural numbers, e.g.:

```
typedef cand' = "{0::nat,1,2}"
  morphisms from_cand cand
by auto

instance cand' :: finite
<proof>
```

We implement each of the relevant quantities in the obvious way (here the suffix `_ex` simply means "example"):

```
abbreviation ballots :: "(cand list) set" where
  "ballots ≡
    {[Alice,Bob],[Alice],[Alice,Bob,Claire],[Bob,Claire],[Bob,Alice]}"

abbreviation cands_ex where
  "cands_ex ≡ {Alice, Bob, Claire}"

abbreviation elected_ex where
  "elected_ex ≡ {Alice, Bob}"

abbreviation gt_cand :: "cand list ⇒ cand ⇒ cand set" where
  "gt_cand b c ≡
       if c ∈ set b
           then set (takeWhile (λc'. c' ≠ c) b)
           else {}"

abbreviation votes_ex :: "(real, cand) vec ⇒ (real, cand) vec"
   where
  "votes_ex w ≡ votes ballots set gt_cand w"

abbreviation seats_ex :: nat where
  "seats_ex ≡ 2"

abbreviation quota_ex :: "(real, cand) vec ⇒ real" where
  "quota_ex w ≡ quota_hb ballots seats_ex set w"

abbreviation excess_ex :: "(real, cand) vec ⇒ real" where
  "excess_ex w ≡ excess ballots set w"
```

As we have already mentioned, the results in this section leverage many of the general results from Section 5.4 onwards: we implement the components for this example in terms of the generally implemented components `votes`, `excess`, and so on.

The following lemma works even if ballots list candidates multiple times, even in a non-consecutive order. Thus, our implementation of $\mathcal{L}, \mathcal{G}$ accepts more ballots in the form of a `cand list` as valid than we may want to, in fact accepting *any* list of candidates as valid, but it works for our purposes because we do not need equal ballots to be equal values.

```
lemma cand_lists_valid:
  "valid_strict_ballot (set b) (gt_cand b)"
```

It is obvious why this works for any list for $\mathcal{L}$, as the Isabelle function `set` removes duplicates, but it is not immediately obvious why it works for $\mathcal{G}$ i.e. `gt_cand` in the above formalisation. To see why it works, consider $[a,b,a,c]$, then those listed above $a$ are $\{\}$, those above $b$ are $\{a\}$, those above $c$ are $\{a,b\}$ – the extra $a$ is irrelevant. Any additional listing of a candidate after the first listing is ignored by using Isabelle's `takeWhile` function, because it stops at the first matching candidate. Given ballot validity, it is trivial to show that what we have is at least a valid *election context* (we will elide interpretation proofs after this):

```
interpretation election_with_seats cands_ex ballots set gt_cand
    cands_ex seats_ex
proof -
  show "election_with_seats cands_ex ballots set gt_cand seats_ex"
  proof
    show "⋀b. b ∈ ballots ⟹ valid_strict_ballot (set b) (gt_cand
    b)"
      using cand_lists_valid by presburger
    show "⋀b. b ∈ ballots ⟹ set b ≠ {}"
      by blast
    show "ballots ≠ {}"
      by simp
    show "finite ballots"
      by auto
    show "0 < seats_ex"
      by simp
    show "finite cands_ex"
      by simp
    show "all_in ballots set ⊆ cands_ex"
      by auto
  qed
next
  show "cands_ex ≡ all_in ballots set"
    <proof>
qed
```

The quota in our example comes out to be $5/3 \approx 1.66$, and $a$ and $b$ are both elected

with $V_a(w) = 3$ and $V_b(w) = 2$. We need to solve $V_a(v) = Q(v), V_b(v) = Q(v)$ for some $v$ subject to $v_a \in [0,1], v_b \in [0,1], v_c = w_c$, where in this particular case $w_a = w_b = w_c = 1$. We solved this abstractly in Section 1.1, and we now we leverage our formal results to show that the solution derived in that section is correct. We start by calculating the initial values after the initial allocation of first preferences (i.e. at $w = \mathbf{1}$), all provable by invoking a single command (with some simple helper lemmas):

```
lemma prior_votes_ex:
  "votes_ex 1 $ Alice = 3"
  "votes_ex 1 $ Bob = 2"
  "votes_ex 1 $ Claire = 0"

lemma prior_excess_ex:
  "excess_ex 1 = 0"

lemma prior_quota_ex:
  "quota_ex 1 = 5/3"
```

As two candidates exceed the quota, we are in a transfer round (for any reasonable $\varepsilon$). As discussed in Section 1.1, there are two solutions to the system of equations, and the only valid one of the two is $v_a = 4/13, v_b = 1/3, v_c = 1, V_a(v) = V_b(v) = Q(v) = 44/39$. We can easily show that this is a solution:

```
abbreviation sol_ex :: "(real, cand) vec" where
  "sol_ex ≡ χ c.
       if c = Alice then
           4/13
       else if c = Bob then
           1/3
       else
           1"

lemma sol_ex_votes:
  "votes_ex sol_ex $ Alice = 44/39"
  "votes_ex sol_ex $ Bob = 44/39"

lemma sol_ex_excess:
  "excess_ex sol_ex = 21/13"

lemma sol_ex_quota:
  "quota_ex sol_ex = 44/39"

lemma solution_given_sol_ex:
  "solution_given 1 sol_ex (votes_ex sol_ex) (quota_ex sol_ex)
   cands_ex elected_ex"
```

Each of these is provable in a single command, aside from `solution_given_sol_ex` which requires a handful. We can now interpret all the various locales for this example:

```
lemma feasible_ex:
  "feasible 1 (votes_ex 1) (quota_ex 1) cands_ex elected_ex"

lemma cands_all_in_ex:
  "all_in ballots set = cands_ex"

interpretation meek_ex: abstract_meek_carrier votes_ex quota_ex
   "1::nat" "0::real" excess_ex "card ballots" seats_ex cands_ex

interpretation meektransfer_fixes_ex: meektransfer_fixes_carrier
   quota_ex "1::nat" "0::real" excess_ex "card ballots" seats_ex
   cands_ex votes_ex "1::(real, cand) vec" elected_ex

abbreviation w_at_ex where
  "w_at_ex ≡ meektransfer_fixes_carrier.w_at quota_ex votes_ex
   (1::(real, cand) vec) elected_ex"

lemma w_at0_1_ex:
  "w_at_ex 0 = 1"

lemma nontriviality_ex:
  assumes w_ge0: "⋀c. c ∈ cands_ex ⟹ 0 ≤ w $ c"
      and w_le1: "⋀c. c ∈ cands_ex ⟹ w $ c ≤ 1"
  shows "(Σc∈cands_ex. votes_ex (1<elected_ex ⟼ w>) $ c) ≥ 1"

interpretation transfer_ex: meektransfer_carrier quota_ex "1::nat"
   "0::real" excess_ex "card ballots" seats_ex cands_ex 1
   elected_ex votes_ex

interpretation transfer_strictin_ex: meektransfer_strictinc_carrier
   quota_ex "1::nat" "0::real" excess_ex "card ballots" seats_ex
   cands_ex 1 elected_ex votes_ex
```

None of the locales besides `meektransfer_fixes_ex` can be fully interpreted automatically in Isabelle, each requiring multiple invocations of its `smt` command. However, they are trivial to prove by the user, as they all go through via locale unfolding and then applying `sledgehammer` on each goal, instantiating variables as needed.

Finally, we invoke Theorem 2 (see Section 3.6) to show that this solution is unique:

```
abbreviation solution_given_for_ex where
  "solution_given_for_ex ≡ meektransfer_fixes_ex.solution_given_for"

lemma unique_solution_ex:
  assumes sol_w: "solution_given_for_ex 1 w"
      and c_cand: "c ∈ cands_ex"
  shows "w $ c = sol_ex $ c"
proof -
  have q: "quota_ex = (λw. (real (card ballots) - excess_ex w) /
    real (seats_ex + 1) + 0)"
     using meek_ex.quota_form by linarith

  obtain v where theorem2:
      "solution_given_for_ex 1 v"
      "(∀w'. solution_given_for_ex 1 w' ⟶ (∀c∈cands_ex. w' $ c = v
    $ c))"
     using transfer_strictin_ex.unique_solution [OF q, of 0]
    w_at0_1_ex less_numeral_extra(1) by auto
  have "w $ c = v $ c" if "c ∈ cands_ex" for c
     using sol_w that theorem2(2) by presburger
  moreover have "sol_ex $ c = v $ c" if "c ∈ cands_ex" for c
     using solution_given_sol_ex that theorem2(2) by presburger
  ultimately show ?thesis
     using c_cand by presburger
qed
```

Figure 5.12: Proof of the uniqueness of the solution vector for the example.

### 5.4.2 The components' implementation is a model of the abstract representation

In this section we prove that *for any valid election*, the components laid out in Section 5.2 are a model of our locale in Chapter 3. We provide some brief commentary for each of the lower-level equivalents of the assumptions of the locale, except the votes invariant which we discuss further in Section 5.5. The order of presentation is in terms of increasing reliance on ballot validity, and hence increasing dependence on our representation of ballots and validity, from no dependence at all to needing full validity. We elide discussion of the statements and proofs of the continuity assumptions and the assumptions for non-candidates, as they follow fairly easily.

In general, we make the assumptions of these lemmas as weak as possible, so that they can be applied with greater ease in this initial theorem development before interpretation. Of course, once we interpret the locales in the context of a valid election, this degree of weakness becomes unnecessary.

First, a brief digression on the development of the locales.

#### 5.4.2.1 On locale refinement

It is not worth going into any of the chronological details, but it is the process of interpretation that led us to correct a few mistakes in our locale assumptions, in an iterative process of refining the implementation and the abstraction. We started our development on the abstraction side. When we later implemented the components, we realised some of the premises of the assumptions of the locale were unnecessarily strong or just not needed e.g. in `V_change` and `V_winc` we originally required the weight to be positive,[4] rather than non-zero and non-negative respectively. This then simplified some of the theorem development inside the locale. This implementation-abstraction refinement process is a distinct advantage of working within a theorem proving environment.

Sometimes, we discovered a proof of a locale assumption that did not depend on the structure of ballots, and thus could potentially be moved to a theorem in the locale, removing locale assumptions and thus increasing the usability and generality of the locale. For example, as discussed in Section 3.5.2.1, we originally assumed (assumption's premises elided) that $\forall w.\ V_c(w)/w_c \leq |B|$, and we had also assumed rather than derived the fact that if $w_c = 0$, $V_c(w) = 0$, the latter of which was possible to prove after relaxing some premises on the weights in other locale assumptions.

Increasing the strength of the locale by removing assumptions then allowed us to remove some development from the implementation side, sometimes a sizeable amount. The resulting proofs once moved to the abstraction were also, as one might expect when one is forced to reason simply in terms of bounds and not nitty-gritty ballot structure, much more elegant and legible.

We expect that gathering evidence for the sufficiency and minimality of the locale assumptions in this way is the only practical way to do so, and given the number of refinements we have applied we feel confident in claiming the locale assumptions are

[4] This was not a major impediment so we did not initially give sufficient thought to weakening the premises at this point, but just proving this fact on the implementation side makes it obvious what is necessary to assume.

both sufficient to prove anything one might want to which depends only on the bounds of the component functions, and minimal enough to be relatively easily implementable and crucially *extendable and generalisable* for a diversity of specific implementations within and outwith the general framework provided by Meek's method.

#### 5.4.2.2 `E_lower`

We prove the assumption showing that the excess is bounded below by the number of ballots multiplied by the product of the proportion of votes received that each candidate passes on:

```
theorem E_lower:
  assumes ge0: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≤ 1"
  shows "excess B ℒ w ≥ card B * (∏c∈all_in B ℒ. 1 - w $ c)"
```

No assumptions about the validity of the ballot are needed. Proving this is simple, we just have to start by removing the inner product's dependence on the ballot bound by the outer sum:

$$\begin{aligned} E(B,w) &= \Sigma_{b\in B}\Pi_{c\in\mathcal{L}(b)}(1-w_c) \\ &\geq \Sigma_{b\in B}\Pi_{c\in\bigcup\{\mathcal{L}(b).\ b\in B\}}(1-w_c) \\ &= |B|\Pi_{c\in\bigcup\{\mathcal{L}(b).\ b\in B\}}(1-w_c) \end{aligned}$$

Seeing how one moves from the first line to the second just requires recognising the fact that, as long as $w$ is valid, the excess is lower (or the same) the more candidates that are listed on ballots.

#### 5.4.2.3 `V_winc`

The lower-level statement of the assumption characterising when the votes function increases when another candidate's weight is decreased:

```
theorem V_winc:
  assumes "c' ≠ c"
      and "w $ c' ≥ 0"
      and "r ≥ 0"
      and le1: "⋀b k. ⟦b ∈ B; c' ∈ ℒ b; k ∈ 𝒢 b c'-{c}⟧⟹
                        w $ k ≤ 1"
  shows "votes B ℒ 𝒢 (w ⌊c,r) $ c' ≥ votes B ℒ 𝒢 w $ c'"
```

Note $c$ is not necessarily listed greater than $c'$; the lemma follows regardless. The proof follows naturally from unfolding definitions and applying the most (intuitively) obvious simplifications at each step.

The weight of $c'$ cannot be negative because then if they received more vote mass than before, they would actually get less votes and the inequality would have to flip. The amount of decrease, $r$, cannot be negative because, again, the inequality would have to flip.

The assumption `le1` states that for every ballot $c'$ is listed on, those greater than $c'$, except $c$, have weight less-or-equal 1. By inspecting the proof of this lemma we see that this could be further weakened to only restrict the weights of those candidates *between* $c'$ and $c$ on the ballot. However, this was unnecessary for easing initial development, and so we have not further refined the theorem.

#### 5.4.2.4 `E_winc`

The assumption showing that the excess increases when a candidate's weight is decreased:

```
theorem E_winc:
  assumes "r ≥ 0"
      and le1: "⋀b k. ⟦b ∈ B; c ∈ ℒ b; k ∈ ℒ b; k ≠ c⟧ ⟹ w $ k ≤ 1"
      and "finite B"
  shows "excess B ℒ (w ⌊ c,r) ≥ excess B ℒ w"
```

We need the set of ballots to be finite here in order to split up the sum over products one gets when unfolding. More fundamentally, if the set of ballots were infinite, the sum with which the excess is defined could diverge, and in that case this inequality would not be provable.

We don't need to say anything about $c$'s weight as the lemma follows regardless of its value, and hence we include $k \neq c$ to weaken the assumption `le1`.

The proof follows by unfolding and simplification of $E(w \lfloor c,r)$, which follows naturally once one starts by splitting sum over the disjoint sets in $B = \{b \in B.\ c \in \mathcal{L}(b)\} \cup \{b \in B.\ c \notin \mathcal{L}(b)\}$.

#### 5.4.2.5 `E_upper`

```
theorem E_upper:
  assumes c_le1: "w $ c ≤ 1"
      and ge0: "⋀c. c ∈ all_in B ℒ ⟹ w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ ⟹ w $ c ≤ 1"
      and c_min: "⋀k. k ∈ all_in B ℒ ⟹ w $ c ≤ w $ k"
      and "finite B"
      and ballots_nonempty: "⋀b. b ∈ B ⟹ ℒ b ≠ {}"
  shows "excess B ℒ w ≤ card B * (1 - w $ c)"
```

Note $c$ is not necessarily a candidate. We originally made this generalisation purely to allow interpretation of a non-carrier set version of the locale as well, where $c$ could be any candidate in `UNIV`, which was our original representation. We have since deprecated this representation of favour of using carrier sets everywhere, but kept the original generality of a number of lemmas. We further discuss this representation decision in Section 5.4.2.9.

This is provable in much the same way as `E_lower`, by unfolding, splitting up $B$, and simplifying. They key step in the proof, where $B_c := \{b \in B.\ c \in \mathcal{L}(b)\}$, is:

$$\Sigma_{b \in B - B_c} \Pi_{k \in \mathcal{L}(b)} (1 - w_k) \leq |B - B_c|(1 - w_c)$$

This ultimately follows because $\forall b \in B - B_c.\ \Pi_{k \in \mathcal{L}(b)} (1 - w_k) \leq 1 - w_c$, provable using `c_min`, and using this to show:

$$\begin{aligned} E(B, w) = \ldots = \\ \Sigma_{b \in B - B_c} \Pi_{k \in \mathcal{L}(b)} (1 - w_k) + |B_c|(1 - w_c) \leq \\ |B - B_c|(1 - w_c) + |B_c|(1 - w_c) = \\ |B|(1 - w_c) \end{aligned}$$

#### 5.4.2.6 `V_change`

The assumption showing the relationship between the votes for a candidate before and after their weight is decreased:

```
theorem V_change:
  assumes self_not_gt: "⋀b. ⟦b ∈ B; c ∈ ℒ b⟧ ⟹ c ∉ 𝒢 b c"
      and wnon0: "w $ c ≠ 0"
  shows "votes B ℒ 𝒢 (w ⦇c := r⦈) $ c =
         votes B ℒ 𝒢 w $ c * (1 - r / w $ c)"
```

It is this result that Woodall does not derive in Theorem 1. For completeness, we provide its derivation, which follows the formal proof. We have

$$
\begin{aligned}
& V_c(w \downarrow c, r) = (w \downarrow c, r)_c \Sigma_{b \in B} \textit{frac-of}(b, c, (w \downarrow c, r)) && \text{by simple unfolding} \\
&= (w_c - r) \Sigma_{b \in B} \textit{frac-of}(b, c, (w \downarrow c, r)) && \text{by definition of } \texttt{dec1} \\
&= (w_c - r) \Sigma_{b \in B} \textit{frac-of}(b, c, w) && \text{as } c \text{ is not listed greater than itself } (\texttt{self\_not\_gt}) \\
&= (w_c - r)(V_c(w)/w_c) && \text{by the definition of } V \text{ and } \texttt{wnon0} \\
&= V_c(w)(1 - r/w_c) && \text{by rearranging}
\end{aligned}
$$

where $1 - r/w_c$ is the exact proportion $c$'s vote decreases with a decrease of $r$ in their weight, as desired. This also works for $r = 0$ and $r < 0$. In the case where $r = 0$, the equation clearly simplifies to $V_c((w \downarrow c, r)) = V_c(w)$ as expected, and when $r$ is less than 0 one can see that the minus becomes a plus and $c$'s vote increases, as expected.

In case it is not clear, $c$ not being listed greater than itself is invoked in the third line because the fraction a candidate receives is unaffected by their own weight as long as this holds.

#### 5.4.2.7 `self_winc`

The assumption showing that a candidate's vote increases (weakly, as usual) if their weight is increased, even if they are eliminated:

```
theorem self_winc:
  assumes "w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ ⟹ w $ c ≤ 1"
      and "r ≤ 0"
      and not_gt_self: "⋀b. ⟦b ∈ B; c ∈ ℒ b⟧ ⟹ c ∉ 𝒢 b c"
      and gt_listed: "⋀b c k. ⟦b ∈ B; c ∈ 𝒢 b k; k ∈ ℒ b⟧ ⟹
                              c ∈ ℒ b"
  shows "votes B ℒ 𝒢 w $ c ≤ votes B ℒ 𝒢 (w ↓ c,r) $ c"
```

This cannot be elegantly subsumed into `V_winc` because $r \geq 0$ is assumed in `V_winc`, and one cannot generalise that without flipping the inequality. Note also that this lemma requires both that candidates are not listed above themselves, and that the candidates listed greater than any other listed candidate are all listed on the ballot.

Clearly all of this would still apply to non-strict ballots. In fact, none of these theorems until we get to the votes invariant require even transitivity of candidates, i.e. an actual ranking!

#### 5.4.2.8 Interpretation

Finally, we can prove that our implementation is a model of the abstraction. This will be brief, as most of the leg-work has been provided by earlier sections. We introduce a new context, an election context *with seats* and within this context prove that our characterisation of (the component functions of) Meek's method is correct.

```
locale election_with_seats = election_context cands ballots ℒ 𝒢
    listed
  for cands :: "'c::finite set"
  and ballots :: "'b set"
  and ℒ :: "'b ⇒ 'c set"
  and 𝒢 :: "'b ⇒ 'c ⇒ 'c set"
  and listed :: "'c set" +
  fixes seats :: nat
  assumes seats_gt0: "seats > 0"
begin

lemma all_in_eq_listed:
  "all_in ballots ℒ = listed"
using listed_def by auto

interpretation meek_interp: abstract_meek_carrier "votes ballots (ℒ
    :: 'b ⇒ 'c set) 𝒢" "quota_hb ballots seats ℒ" "1 :: nat" "0 ::
    real" "excess ballots ℒ" "card ballots" seats "all_in ballots ℒ"
```

Figure 5.13: Interpretation of the Meek locale for any valid election with seats.

It is clear what most of these parameters are, except the 1 and 0: recall that the locale we defined required an additive denominator plus an extra amount to be added to the quota, here was pass 1 for the denominator and 0 for the part to add on, as with the standard quota for Meek's method. The assumptions it requires us to prove are precisely those in Section 5.4.2.2 through to Section 5.4.2.7 (apart from the few we elided). Refer back to Section 3.3.2 for details.

This, together with the concrete example, concludes the major goal of this chapter: show that the Meek's method locale is consistent, feasible to interpret, and is generically applicable to any valid election, along with proving the same for the transfer round locale for an explicit example. As we have already seen with the concrete example, with this interpretation we can now use any of the lemmas and theorems proven within the locale for this specific implementation.

#### 5.4.2.9 Carrier sets, clarity, and automation

Without going into too much detail, it is worth remarking upon the impact of representation as it pertains to carrier sets. In our original development, apart from the fact that we had originally represented Meek's method using the enlisting variant in the locale (see Chapter 3.4), we also represented the set of candidates like so:

```
abbreviation cands :: "'c set" where
  "cands ≡ UNIV"
```

Figure 5.14: Original representation for candidates in the Meek locales.

This makes some sense, in particular because the type variable `'c` is forced to be of sort `finite` in our development, and so it is likely to represent the whole set of candidates in practice. This choice eliminates the need for assumptions about non-candidates, and eliminates the need to include $c \in cands$ in so many assumptions, lemmas, and case splits. It also allows us to work with direct vector equality and limits, rather than sub-vector equality and limits of projections.

However, the most natural set to work with at the level of implementation is never `UNIV`, it is `all_in ballots` $\mathcal{L}$ (see Section 5.2), as no candidate which is not actually listed on any ballot makes any difference to any of the lemmas we have proven. We may thus wish to interpret the locale with `all_in ballots` $\mathcal{L}$ representing the full set of candidates, which is what we do (without issue).

We managed to interpret the `UNIV` version with results proven only over this set, but the presence of `UNIV` if one uses the universal set for the set of candidates on the locale side in the assumption `E_lower` (see Section 5.4.2.2) necessitated jumping through hoops to prove, and it was only by accident that there was a way of doing this; not necessarily the case that one will be as lucky for all possible future assumptions involving `UNIV`, and so we have deprecated this approach.

Starting with the carrier set also has the advantage of being more easily extendable to infinite sets of candidates, potential or actual, if anyone wishes to take up such a task in the future.

## 5.5 An important invariant: votes-in-circulation plus excess

Without needing to prove some theorem which requires total ballot validity, we would be rightly suspicious of whether the set of assumptions of the locale are anything close to sufficiently characterising the component functions of Meek's method. The votes invariant is such a theorem, requiring ballots to be a – in our case strict – total order over a subset of candidates.

This section, through our investigation and proof of this invariant, highlights several key representational decisions. First, our representation of ballots lends itself very nicely to a ballot induction rule that is comfortable to work with for both proof *exploration* as well as for legibility in the final Isabelle/Isar proof, even for large proofs such as this. At least, provided one is used to reading $(\mathcal{L},\ \mathcal{G})$ as a ballot, one whose structure we can easily play with using usual set-theoretical functions.

Second, it exposes the not so fortunate tension between the nature of a ballot as a *set* of elements and the nature of a ballot as a *sequence* of elements. Sometimes we want to take products over sets of candidates on ballots satisfying various predicates, or treat it quite simply as a set of candidates. Sometimes, we want to map over a ballot in an *order-dependent way*; a list representation for ballots is a common one for this purpose. However, it was (and is) our belief that a *set-based* representation of ballots is more appropriate[5] for reasoning, and analysis in Isabelle/HOL specifically, deriving sequence-based representation from this. We also discussed why we did not choose a relational approach, where one has a binary relation on outcomes denoting preference, in Chapter 4. On the implementation side, the $(\mathcal{L},\ \mathcal{G})$ approach is justified by simply being the most natural way to implement the components, which we would have to define and use regardless.

Third, this is the first major use case for the whole of the development in Chapter 4 outside of the context of that chapter. We require several key results about both `rank` and `ranked` from that theory. The latter function in particular provides us with our map from ballot-as-*set* to ballot-as-*sequence*.

[5] We leave to one side implicit representations based on underlying "objective" or subjective preference or utility, such as those based on a utility function taking an outcome and returning a real number, common in the game-theoretic side of social choice theory.

### 5.5.1 Ballot mass preservation

As a prelude to proving the votes invariant (see Section 5.5), we show that for each ballot[6] $(\mathcal{L}, \mathcal{G})$, if one sums the amount of this ballot that each candidate receives, with the amount of the ballot that is exhausted, we get a total ballot mass of 1. That is, ballot mass is preserved.

$$\forall \mathcal{L}\ \mathcal{G}\ c\ w.\ valid(\mathcal{L}, \mathcal{G}) \longrightarrow \Sigma_{c \in \mathcal{L}} w_c frac\text{-}of(\mathcal{L}, \mathcal{G}, c, w) + \Pi_{c \in \mathcal{L}}(1 - w_c) = 1 \quad (5.9)$$

```
lemma sum_fracs_eq1:
  fixes ℒ :: "'c::finite set"
  assumes valid: "valid_strict_ballot ℒ 𝒢"
  shows "(Σc∈ℒ. w $ c * frac_of ℒ 𝒢 c w) + (Πc∈ℒ. 1 - w $ c) = 1"
```

Figure 5.15: Lemma showing that ballot mass is preserved.

This lemma says nothing about whether some candidates receive negative ballot mass, merely that it all balances out in the end. That is, we prove that ballots do not contribute any more or less than they should to the votes in circulation, regardless of the weight vector $w$. The formal proof takes 250 lines of Isar.

As our proof is novel and important to the votes invariant, we will give a breakdown of its main steps next. We begin by inducting on the ballot:

- The base case of an empty ballot happens to follow trivially despite being somewhat nonsensical, as a sum over an empty set is 0, and the product is 1. Thus, for *non-empty* ballots we can interpret the equality as saying that no more nor less than the ballot's unit mass is distributed between candidates and the excess, while for the empty ballot the expression simply happens to hold with no uncontrived interpretation of the fact; it simply allows us to not add an assumption about emptiness.

- The inductive step with IH of the form $\forall \mathcal{L}'\ \mathcal{G}'.\ subballot(\mathcal{L}', \mathcal{G}', \mathcal{L}, \mathcal{G}) \longrightarrow V(\mathcal{L}', \mathcal{G}') + e(\mathcal{L}') = 1$ proceeds by obtaining a subballot $(\mathcal{L}', \mathcal{G}')$ of $(\mathcal{L}, \mathcal{G})$, with $\mathcal{L} = \mathcal{L}' \cup \{c\}$ for some new,[7] fixed $c$, and splitting the ballot up into those candidates $\sigma = \{c' \in \mathcal{L}'.\ \mathcal{G}'\ c' = \mathcal{G}\ c'\}$ whose rank does not change after the

[6] We refer to $(\mathcal{L}, \mathcal{G})$ and do not make an indirection via some $b$ in order to maintain mutual intelligibility with the Isabelle/Isar proof.

[7] In particular, not the $c$ of the lemma statement, which the obtained $c$ shadows.

removal of $c$ from $\mathcal{L}$, and those whose rank does change, $\mathcal{L}' - \sigma$. We have $V(\mathcal{L}', \mathcal{G}') + e(\mathcal{L}') = 1$ by the inductive hypothesis, meaning the subballot preserves vote mass.

- If $V'(\mathcal{L}_1, \mathcal{L}_2, \mathcal{G})$ is defined[8] as the amount of votes a ballot $(\mathcal{L}_2, \mathcal{G})$ would contribute to the candidates in the set $\mathcal{L}_1$, and $e(\mathcal{L})$ is the amount that would be exhausted by the candidates in the set $\mathcal{L}$, that is if:
  - $V'(\mathcal{L}_1, \mathcal{L}_2, \mathcal{G}) =_{def} \Sigma_{c \in \mathcal{L}_1} w_c \textit{frac-of}(\mathcal{L}_2, \mathcal{G}, c, w)$,
  - $V(\mathcal{L}, \mathcal{G}) = V'(\mathcal{L}, \mathcal{L}, \mathcal{G})$,
  - $e(\mathcal{L}) = \Pi_{c \in \mathcal{L}} 1 - w_c$,

  then through algebraic manipulation we can reduce
  - $V(\mathcal{L}, \mathcal{G}) + e(\mathcal{L})$, to
  - $1 + w_c(\textit{frac-of}(\mathcal{L}, \mathcal{G}, c, w) - V'(\mathcal{L}' - \sigma, \mathcal{L}', \mathcal{G}') - e(\mathcal{L}'))$
- We prove the expression $1 + w_c(...)$ equals 1 by showing that $\textit{frac-of}(\mathcal{L}, \mathcal{G}, c, w) = V'(\mathcal{L}' - \sigma, \mathcal{L}', \mathcal{G}') + e(\mathcal{L}')$, and thus essentially complete the proof.

One should read the main equivalence, $\textit{frac-of}(\mathcal{L}, \mathcal{G}, c, w) = V'(\mathcal{L}' - \sigma, \mathcal{L}'\mathcal{G}') + e(\mathcal{L}')$, we wish to prove as "the amount $c$ receives on $(\mathcal{L}, \mathcal{G})$ is equal to what candidates listed less than $c$ receive on $(\mathcal{L}', \mathcal{G}')$ plus the excess on $(\mathcal{L}', \mathcal{G}')$".

This makes sense intuitively. Viewing the ballot $(\mathcal{L}', \mathcal{G}')$ as an original ballot which we "insert into" to get to $(\mathcal{L}, \mathcal{G})$, we can see that a new candidate $c$ when inserted anywhere in $(\mathcal{L}', \mathcal{G}')$, "sucks up" everything that was going to those now below them plus the excess. Recall that "votes received" refer to *frac-of* (not $V$), which is what a candidate receives prior to passing any of it on. Suppose the ballot is *abdefg*, and we insert candidate $c$ to get *abcdefg*. What *defg* and the excess received before, spread between them according to their weights, was everything *ab* passed on, and now $c$ stands in the way, receiving all of that.

This completes the proof. The key is applying ballot induction, recognising that the important quantity is that which $c$ receives and which all those previously below $c$ received, and algebraic manipulation to get to a point where this equivalence expression between pre- and post-insertion is involved (in the course of which, the inductive hypothesis is used), which we can prove. The only thing one needs to take our word on

[8] Note we use subscripts 1 and 2 to avoid overloading $\mathcal{L}$ in the definition of $V'$.

is that this algebraic manipulation is valid, given the assumption that the ballot is valid and nothing else (i.e. nothing about the weights). This was far from trivial to prove and it would be potentially be productive to investigate alternative induction strategies for ballots in future work.

Proving the equivalence $\mathit{frac\text{-}of}(\mathcal{L},\mathcal{G},c,w) = V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')+e(\mathcal{L}')$ is what requires the important facts about the ballot being a strict, linear order on a subset of ballots. We will proceed to prove this in the following section.

### 5.5.2 Proof that an inserted candidate receives all that was going to those below

The equality

$$\mathit{frac\text{-}of}(\mathcal{L},\mathcal{G},c,w) = V'(\mathcal{L}'-\sigma,\mathcal{L}'\mathcal{G}')+e(\mathcal{L}')$$

follows easily if $c$ is ranked first, as then $\sigma = \{\}$ and we have $V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')+e(\mathcal{L}') = V(\mathcal{L}',\mathcal{G}')+e(\mathcal{L}')$ which equals 1 by the induction hypothesis.

The case where $c$ is not ranked first is more difficult. In this case, we are forced to consider those listed above $c$ and part of the proof in this case relies on an equality holding that can only be proven true if we take advantage of the fact that the ballot is a linear order over a subset of candidates. Our proof of this subsumes the $\sigma = \{\}$ case, so there is no case-split in the formal proof.

The proof proceeds as follows. We will focus on the second step in another section, the first step after the unfolding, as it is the most substantial. That is, the most substantial besides the problem of constructing a correct proof as a whole in the first place.

$$
\begin{aligned}
&\textit{frac-of}(\mathcal{L},\mathcal{G},c,w) = \Pi_{k\in\mathcal{G}(c)}(1-w_k) && \text{by unfolding and } c\in\mathcal{L} \text{ via } \mathcal{L}=\mathcal{L}'\cup\{c\}\\
&= 1-\Sigma_{k\in\mathcal{G}(c)}w_k\Pi_{k'\in\mathcal{G}(k)}(1-w_{k'}) && \text{see Section 5.5.2.1}\\
&= \begin{cases} 1-\Sigma_{k\in\mathcal{G}(c)}w_k\Pi_{k'\in\mathcal{G}(k)}(1-w_{k'}) & \text{if } k\in\mathcal{L}'\\ 1-\Sigma_{k\in\mathcal{G}(c)}w_k * 0 & \text{otherwise}\end{cases} && \text{given } \forall k\in\mathcal{G}(c).\ k\in\mathcal{L}'\\
&= 1-\Sigma_{k\in\mathcal{G}(c)}w_k\textit{frac-of}(\mathcal{L}',\mathcal{G},k,w) && \text{by def. of } \textit{frac-of}\\
&= 1-V'(\mathcal{G}(c),\mathcal{L}',\mathcal{G}) && \text{by def. of } V'\\
&= 1-V'(\sigma,\mathcal{L}',\mathcal{G}) && \text{as by def. of } \sigma \text{ and some ballot facts we have } \sigma=\mathcal{G}(c)\\
&= 1-V'(\sigma,\mathcal{L}',\mathcal{G}') && \text{as } \mathcal{G}'(c)=\mathcal{G}(c) \text{ on } \sigma(=\mathcal{G}(c))\\
&= V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')+e(\mathcal{L}') && \text{by the induction hypothesis}
\end{aligned}
\tag{5.10}
$$

The final step is an equivalence that is proven in the penultimate step in the series of equational reasoning steps leading up to proving ballot mass preservation:

$$
\begin{aligned}
&V(\mathcal{L},\mathcal{G})+e(\mathcal{L}) = ...\\
&= V'(\sigma,\mathcal{L}',\mathcal{G}')+V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')+e(\mathcal{L}')+\\
&\ w_c(\textit{frac-of}(\mathcal{L},\mathcal{G},c,w)-V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')-e(\mathcal{L}'))\\
&= 1+w_c(\textit{frac-of}(\mathcal{L},\mathcal{G},c,w)-V'(\mathcal{L}'-\sigma,\mathcal{L}',\mathcal{G}')-e(\mathcal{L}'))
\end{aligned}
$$

which follows by the induction hypothesis, and one can see by inspection by eliminating the sub-expression multiplied by the weight $w_c$ obtains our needed fact. We will now address the second step in Proof 5.10 as promised.

#### 5.5.2.1 Connecting natural-number orderings and the ballot representation

The most difficult and most important part of the proof of ballot mass preservation relies on key properties of `ranked` and general lemmas concerning sums and products over sequences. If we zoom in on this key step in the proof we have the following reasoning, with the name of the lemma associated with each step listed alongside.

$$\begin{aligned}
&\Pi_{k\in\mathcal{G}(c)}(1-w_k) = \Pi_{i<|\mathcal{G}(c)|}(1-w_{ranked(i)}) && \texttt{prod_seq_to_set}\\
&= 1-\Sigma_{i<|\mathcal{G}(c)|}w_{ranked(i)}\Pi_{j<i}(1-w_{ranked(j)}) && \texttt{prod_one_minus_sequence_lt}\\
&= 1-\Sigma_{k\in\mathcal{G}(c)}w_k\Pi_{k'\in\mathcal{G}(k)}(1-w_{k'}) && \texttt{sum_prod_seq_to_set}
\end{aligned}$$

In ordinary mathematical notation, we can write each of the lemmas involved like so:

- `prod_seq_to_set`: *For $f : C \to \mathbb{R}$, finite $X$, and a sequence $(s_n)_{n\in\mathbb{N}}$ of elements of $C$ satisfying $\{s_n\}_{n\in\{0..|X|-1\}} = X$ we have:*

$$\Pi_{i<|X|} f(s_i) = \Pi_{x\in X} f(x) \tag{5.11}$$

- `prod_one_minus_sequence_lt`: *If $n > 0$, we have:*

$$\Pi_{i<n}(1-s_i) = 1-\Sigma_{i<n}s_i\Pi_{j\leq i}(1-s_j) \tag{5.12}$$

- `sum_prod_seq_to_set`: *For $f : C \to \mathbb{R}$, $g : C \to \mathbb{R}$, finite $X$, and a sequence $(s_n)_{n\in\mathbb{N}}$ of elements of $C$ together with a function $Y : C \to \mathcal{P}(C)$ satisfying $\{s_n\}_{n\in\{0..|X|-1\}} = X$, $Y(s_0) = \{\}$, $\forall n > 0.\ n \leq |X|-1 \longrightarrow Y(s_n) = \{s_m\}_{m\in\{0..n-1\}}$, and $\forall x \in X.\ finite(Y(x))$, we have:*

$$\Sigma_{i<|X|} f(s_i)\Pi_{j<i} g(s_j) = \Sigma_{x\in X} f(x)\Pi_{y\in Y(x)} g(y) \tag{5.13}$$

The arbitrary function $f$ can be generalised, as we have shown in the formal proof, to take any argument type, and it can take any return type which implements the functionality of a commutative multiplicative monoid in `prod_seq_to_set`, and the functionality of a commutative ring in `sum_prod_seq_to_set`. The type constraints of the function $g$ in the latter lemma are the same as those of $f$. We provide the Isabelle/HOL notation (eliding type annotations) for one of the lemmas here, to facilitate the discussion that follows:

```
lemma sum_prod_seq_to_set:
  fixes f :: "'a ⇒ 'b::{ring,comm_monoid_mult}"
  assumes fin: "finite X"
      and seq_image: "seq ` {0 .. card X - 1} = X"
      and Y_seq0: "Y (seq 0) = {}"
      and Y_seqn: "⋀n. ⟦n ≤ card X - 1; n > 0⟧ ⟹
                        Y (seq n) = seq ` {.. n - 1}"
      and finite_Y: "⋀x. x ∈ X ⟹ finite (Y x)"
  shows "(Σi<card X. f (seq i) * (Πj<i. g (seq j))) =
         (Σx∈X. f x * (Πy∈Y x. g y))"
```

Figure 5.16: Lemma connecting a sum and product over a sequence to a sum and product over a set of elements related by functions expressing an ordering over the elements.

The sequence we need which satisfies these requirements is, as we have already seen, the sequence which identifies the top-ranked candidate with natural number 0, the next with natural number 1, and so on. That is, the sequence *ranked*. If we re-read each of the requirements of the lemma `sum_prod_seq_to_set` with the following associated ballot identifications then we can understand them more intuitively:

$$\texttt{seq} \equiv \texttt{ranked}\ \mathcal{L}\ \mathcal{G}$$
$$\texttt{X} \equiv \mathcal{G}\ \texttt{c} \qquad \texttt{Y} \equiv \mathcal{G}$$
$$\texttt{f} \equiv \lambda\texttt{c.}\ \texttt{w \$ c} \qquad \texttt{g} \equiv \lambda\texttt{c.}\ \texttt{1 - w \$ c}$$

then the requirements read as:

- `fin` and `finite_Y`: basic ballot finiteness.
- `seq_image`: The candidates listed greater than $c$ must be exactly those candidates identified by the rankings 0 through to one less than $c$'s own ranking.
- `Y_seq0`: The top-ranked candidate must have nobody listed above them.
- `Y_seqn`: The candidates listed greater than the $n^{th}$ ranked candidate, for all $n$ with $0 < n < |\mathcal{G}(c)|$, must be exactly those with rankings 0 through to one less than $n$.

These are precisely the results we developed in Section 4.8.

**5.5.2.1.1 Proof of the sequence-to-set results** As we have seen, there are two lemmas relating the function *ranked*, ballot-as-sequence, to the functions $(\mathcal{L}, \mathcal{G})$, ballot-as-sets, in the proof of ballot mass preservation: lemmas `prod_seq_to_set` and

`sum_prod_seq_to_set`. We will outline a proof for the latter of these below. We omit a proof of the former as it is conceptually very similar but simpler and less interesting. A reminder of what we need to prove in `sum_prod_seq_to_set`, with the assumptions elided:

$$\Sigma_{i<|X|} f(s_i) \Pi_{j<i} g(s_j) = \Sigma_{x \in X} f(x) \Pi_{y \in Y(x)} g(y) \tag{5.14}$$

We can see by inspection that if $c$ is top-ranked – i.e. $|\mathcal{G}(c)| = 0$ with $X \equiv \mathcal{G}(c)$ – this trivially follows because $\Sigma_{i<0} f(i) = 0$ for any $f$ by standard convention as well as in Isabelle/HOL. The case where $c$ is not ranked first is more involved, and currently takes 800 lines of Isabelle.

Some of this is proof bloat, down to undesirable case-based reasoning forced by the fact that splitting the sets along the lines of $\{..m\} = \{..n-1\} \cup \{n..m\}$ for any $n, m \in \mathbb{N}$ with $n \leq m$ does not work when $n = 0$, as in Isabelle/HOL functions are total on the type and hence $0 - 1$ resolves to 0 for the natural numbers – as is most reasonable – and hence $\{..n-1\} = \{0\}$ and so the two sets $\{..n-1\}$ and $\{n..m\}$ are not disjoint. Disjointness is required several times during the proof.

We will merely give an outline of the proof here, because while it was fairly laborious, its details are not particularly insightful. It proceeds as follows:

*Proof.* Case split on emptiness of $X$, trivially dispatching the empty case. For the non-empty case, induct on $X$.

The singleton case introduces $x$ (with $X$ replaced by $\{x\}$), and we can deduce $x = s_0$ from the induction premises. The case follows as we can deduce that the LHS is equal to $f(x)$ due to $\Pi_{j<0}(...) = 1$ being true for any inner function and the fact $s_0 = x$, and the RHS is equal to $f(x)$ as we can conclude $Y(x) = \{\}$ from the fact $Y(s_0) = \{\}$ and the fact that $Y(s_0)$ is finite (needed as infinite sets also have cardinality 0 in Isabelle/HOL).

The induction case, which introduces a newly bound $X$ and an $x$, proceeds by first obtaining the place in the sequence $n$ where $s_n = x, n \leq |X|$. We then break down the proof by working through the expressions this known fact about $x$.

The induction hypothesis has to work for any arbitrary sequence $(s_n)_{n \in \mathbb{N}}$ and greater-than function $Y$ (remembering $Y$ will be instantiated as $\mathcal{G}$) satisfying the induction premises, i.e. they are quantified in the IH, and the key pair of functions to apply it to are the following: $s' \equiv \lambda i.$ if $i < n$ then $s_i$ else $s_{i+1}$ and $Y' \equiv \lambda y.\ Y(y) - \{x\}$. These essentially represent the sequence (e.g. ballot-ranking) and greater-than function (e.g.

$\mathcal{G}$) "before inserting $x$", as is the common pattern for the traditional ballot-specific induction proofs too, where in the proof patterns more typical in this development we instead obtain $\mathcal{L}', \mathcal{G}'$ in a set-centric rather than sequence-centric way. Once we apply the IH to these two functions, the rest of the proof follows naturally, albeit at some length. □

**5.5.2.1.2 Proof of the purely sequence-based result** The lemma concerning just sequences, `prod_one_minus_sequence_lt`, is easily provable from a less-or-equal version, which we prove by induction on $n$:

$$\Pi_{i\leq n}(1-s_i) = 1-\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j)$$

*Proof.* The base case follows by reduction of both sides to $1-s_0$. In the inductive case we, of course, have to prove the following:

$$\Pi_{i\leq n+1}(1-s_i) = 1-\Sigma_{i\leq n+1}s_i\Pi_{j\leq i}(1-s_j)$$

This follows by expanding the left-hand size, invoking the induction hypothesis and simplifying the expression, and then doing the same to the right-hand side:

$$\begin{aligned}
&\Pi_{i\leq n+1}(1-s_i) = (1-s_{n+1})\Pi_{i\leq n}(1-s_i) && \text{extract n+1 from the product}\\
&= (1-s_{n+1})(1-\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j)) && \text{by IH}\\
&= 1-s_{n+1}-(1-s_{n+1})(\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j)) && \text{by distributivity}
\end{aligned}$$

also

$$\begin{aligned}
&\Sigma_{i\leq n+1}s_i\Pi_{j\leq i}(1-s_j) = s_{n+1}\Pi_{j<n+1}(1-s_j)+\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j) && \text{extract n+1}\\
&= s_{n+1}\Pi_{j\leq n}(1-s_j)+\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j) && \text{reindex product}\\
&= s_{n+1}(1-\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j))+\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j) && \text{by IH}\\
&= s_{n+1}+(1-s_{n+1})(\Sigma_{i\leq n}s_i\Pi_{j\leq i}(1-s_j)) && \text{by rearranging}
\end{aligned}$$

hence $\Pi_{i\leq n+1}(1-s_i) = 1-\Sigma_{i\leq n+1}s_i\Pi_{j\leq i}(1-s_j)$. □

In Isabelle/HOL:

```
lemma prod_one_minus_sequence:
  fixes seq :: "nat ⇒ real"
  shows "(Π i≤n. 1 - seq i) = 1 - (Σ i≤n. seq i * (Π j<i. 1 - seq j))"
```

Figure 5.17: A lemma expanding the product of 1 minus an index into a sequence.

### 5.5.3 Proving the invariant

With ballot mass preservation finally proven (Sections 5.5.1 and 5.5.2), we can leverage this to prove the votes invariant. We presented the general theorem in Equations (5.1), (5.2) and (5.3). Here we will prove a version which states that no matter the weight vector, the sum of votes in circulation plus the excess is always exactly equal to the number of ballots, effectively a combination of Equations (5.2) and ( 5.3) which we presented at the start of Section 5.1.4:

$$\forall C\ B\ w.\ |B| = \Sigma_{c \in C} V_c(B,w) + E(B,w) \tag{5.15}$$

The statement of this theorem exists within an election context (with seats). Hence, we already have a fixed set of valid ballots in scope. We prove the invariant with the modest generalisation that it is true for any subset of the ballots in the election, in order for it to be as general as possible inside the election context.[9]

```
theorem votes_invariant:
  assumes "B ⊆ ballots"
  shows "card B = (Σ k∈all_in B ℒ. votes B ℒ 𝒢 w $ k) + excess B ℒ w"
```

Figure 5.18: Theorem showing the votes invariant holds for any subset of the ballots and any weight vector.

The case where the initial sum is over the whole set of candidates rather than just those listed follows easily as a corollary.[10] Note that in this version of the theorem we also permit $B = \{\}$, unlike if we proved it for all of *ballots*, which is non-empty within the election context. The proof proceeds by induction on the finite set $B$, with the induction rule for finite sets being analogous to our induction rule for ballots, as discussed in Section 4.4.

[9] We could have proven it for any finite set of valid ballots, outside of this context, removing requirements like non-empty ballots, but this would be a trivial and needless generalisation at this stage.

[10] The first summation could technically be over any superset of $B$ which is still a subset of *ballots*.

We begin by performing a case split on whether $B = \{\}$. The empty case follows trivially as both the votes in circulation and the excess are defined in terms of a sum over ballots, which reduces to 0 when $B = \{\}$.

In the non-empty case we have to consider the induction step, which consists of a new fixed $B$ satisfying the induction hypothesis. The induction hypothesis states that all proper subsets of $B$ maintain the invariant. As $B$ is non-empty, we can pull out a ballot $b'$ and define $B'$ such that we have $B = B' \cup \{b'\}$. If we can somehow reduce the problem from sums over $B$ to sums over $B'$ and show this expression is equal to $|B'| + 1$ then we will be done, as $|B'| + 1 = |B|$. Indeed, through a series of steps involving unfolding and then rearranging we can show that:

$$\begin{aligned}
&\Sigma_{c \in allin(B)} V_c(B, w) + E(B, \mathbf{w}) = |B'| + \\
&\left.\begin{aligned}
&\Sigma_{k \in \mathcal{L}(b')} V_k(B', w) - \\
&\Sigma_{k \in allin(B') \cap \mathcal{L}(b')} V_k(B', w) + \\
&\Sigma_{k \in allin(B)} V_k(\{b'\}, w) + \\
&\Pi_{k \in \mathcal{L}(b')} (1 - w_k)
\end{aligned}\right\} \nabla
\end{aligned}$$

It remains to show that the part of the expression after $|B'|$ reduces to 1. We will label this expression $\nabla$ for convenience. First, we eliminate the presence of $B$ in the expression $\Sigma_{k \in allin(B)} V_k(\{b'\}, w)$. We can show that:

$$\begin{aligned}
&\Sigma_{k \in allin(B)} V_k(\{b'\}, w) = \\
&\Sigma_{k \in allin(B')} V_k(\{b'\}, w) + \\
&\Sigma_{k \in \mathcal{L}(b')} V_k(\{b'\}, w) - \\
&\Sigma_{k \in allin(B') \cap \mathcal{L}(b')} V_k(\{b'\}, w)
\end{aligned} \tag{5.16}$$

If we substitute this into $\nabla$ we now have it so that each sub-expression, each sum or product, is concerned exclusively with the votes going to the candidates listed on $b'$ or the votes going to candidates listed on $B'$ that come from those listed on $b'$, or some combination of this.

Recall that $B = B' \cup \{b'\}$. It may happen to be the case that $allin(B')$ and $\mathcal{L}(b')$ share no candidates in common, so the intersection in the last line cannot be reduced any further without considering the relationship on a case by case basis.

The case where $\mathcal{L}(b') = \{\}$ is excluded by the fact that no ballots can be empty in an election context. However, even if it were empty, the theorem would be provable

as all of the sums in $\nabla$ reduce to 0 and the singular product reduces to 1 as it is the product over an empty set, and we are done.

In the case where $\mathcal{L}(b') \neq \{\}$, it is not obvious how to proceed. We can get unstuck by considering what else in this context should be equal to 1, and whether that could help simplify what we need to prove.

In Section 5.5.1 we proved an equality which can do precisely this job, which is ballot mass preservation. If this more unwieldy expression is reducible to the amount which $b'$ contributes in ballot mass, then it must also be reducible to 1 by that result. Indeed, by substituting that expression for 1 in the equation we are trying to prove and bringing everything over to one side, we can complete the proof by showing that $\nabla - (\text{contribution of } b') = 0$.

$$
\left.\begin{array}{l}
\Sigma_{k\in\mathcal{L}(b')}V_k(B',w) - \Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(B',w) + \\
\Sigma_{k\in allin(B')}V_k(\{b'\},w) + \Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) - \\
\Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(\{b'\},w) + \Pi_{k\in\mathcal{L}(b')}(1-w_k) -
\end{array}\right\} \nabla
$$

$$
\begin{array}{ll}
\Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) - \Pi_{k\in\mathcal{L}(b')}(1-w_k) & \text{mass contributed by } b',\ =1 \\
= \Sigma_{k\in\mathcal{L}(b')}V_k(B',w) - \Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(B',w) + & \\
\Sigma_{k\in allin(B')}V_k(\{b'\},w) - \Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(\{b'\},w) & \text{by cancelling } (5.17)
\end{array}
$$

We will focus on the case $\mathcal{L}(b') \subseteq allin(B')$ to demonstrate the general approach to finalising this proof:

$$
\begin{array}{ll}
\Sigma_{k\in\mathcal{L}(b')}V_k(B',w) - \Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(B',w) + & \\
\Sigma_{k\in allin(B')}V_k(\{b'\},w) - \Sigma_{k\in allin(B')\cap\mathcal{L}(b')}V_k(\{b'\},w) & \text{by cancelling} \\
= \Sigma_{k\in\mathcal{L}(b')}V_k(B',w) - \Sigma_{k\in\mathcal{L}(b')}V_k(B',w) + & \\
\Sigma_{k\in allin(B')}V_k(\{b'\},w) - \Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) & \text{as } allin(B')\cap\mathcal{L}(b') = \mathcal{L}(b') \\
= \Sigma_{k\in allin(B')}V_k(\{b'\},w) - \Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) & \text{cancelling} \\
= \Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) + \Sigma_{k\in allin(B')-\mathcal{L}(b')}V_k(\{b'\},w) - & \\
\Sigma_{k\in\mathcal{L}(b')}V_k(\{b'\},w) & \text{summing out} \\
= \Sigma_{k\in allin(B')-\mathcal{L}(b')}V_k(\{b'\},w) & \text{cancelling} \\
= 0 & (5.18)
\end{array}
$$

The last step of this reasoning follows as if $b'$ is the only ballot, the amount of this ballot going to candidates not listed on $b'$ is clearly 0. Other cases can be similarly reduced to 0 in this way.

## 5.6 Size of formalisation

The Isabelle/HOL involved in proving the results of this chapter amounts to around 7,000 lines of code. We do not have a pen-and-paper reference because the act of connecting abstraction and implementation in this way is not something available to a pen-and-paper approach, and is a unique ability that working with an interactive theorem prover provides us.

## 5.7 Related work

The most relevant existing literature is, as with Chapter 3, the works of Ghale et al. [25, 26, 24] and Dawson et al. [17]. There is also the work Moses et al. to consider [55]. We covered generality of representation in our discussion of this in Chapter 3. Here, we will briefly cover the aspects of their work relevant to verified, executable code, as this is most relevant to our connecting implementation and abstraction, even without our own extracted code, especially for the purposes of evaluating whether our approach is suitable for extension to cover similar purposes.

Ghale et al. in *Modular Formalisation and Verification of STV Algorithms* [27] use the in-built code extraction mechanism of the dependently-typed interactive theorem prover Coq to generate Haskell code implementing some simple STV procedures.

Dawson et al. in *Machine-Checked Reasoning About Complex Voting Schemes Using Higher-Order Logic* [17] used the HOL4 theorem prover [71] to verify the correctness of a simple method for STV and was able to leverage this to (manually) produce executable code, though the approach did not scale beyond small elections.

In *No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes* Moses et al. show that one can produce a verifiable certificate for the Schulze method of counting [70] at large scale. They, like Ghale et al. and others, take the approach of encoding the computations as proof steps. The approach requires one to be able to logically specify the method in the way they do for simple STV methods, and this is again not clearly extendable to Meek's method. They distinguish *verification*, meaning proving the execution of the method is in gen-

eral correct, and *verifiability*, meaning the ex post facto ability to verify that a specific execution of the method was correct. In this thesis we have only focused on the former. Put another way, verification is a technical act, whereby we use mathematics to rigorously prove correctness, termination, and other desirable properties. Verifiability is a social act enabled by the technical design of protocols and algorithms, and is thus more related to various cryptographic properties of communication protocols and the like than algorithm verification of the kind we have covered here.

Also relevant are the existing, in-use but un-verified procedural implementations of Meek's method. These are Hill and Wichmann's implementation in the Pascal [40] and OpenSTV's[11] implementation of Meek's method in Python. In future it would be worthwhile evaluating these implementations by running them against verified, extracted code for as many realistic, generated elections as is possible.

Key to many of these approaches (see also Pattinson et al. [67], Verity et al. [77], and Beckert et al. [8]) is *formal runtime certificates*, states linked by rules as an inductive datatype, which provides an auditable output which one can inspect to ensure that the underlying computation proceeded correctly. Of course, one can also provide some quality-of-life tool to map this computer-verifiable certificate to a human-readable one.

[11] Now proprietary and called OpaVote. [64]

## 5.8 Conclusion

In this chapter we have demonstrated the consistency of the abstract representation of Meek's method by providing an interpretation that uses a functional implementation of each component of the algorithm. We demonstrated the generality of the locale by interpreting it using an arbitrary election context.

We provided two key interpretations. The first used a specific transfer round example that we first discussed in Section 1.1.1. We showed that the analytical solution to the system of equations that need to be solved for this transfer round is equivalent to the solution one gets by taking the limit of the sequence of weight vectors produced by iteratively updating the weight vector. In doing so, we showed that the general result from Section 3.6, regarding solution vector uniqueness, can be appropriately applied in this specific context. By providing a model we have shown that the assumptions in both locales are consistent (relative to the consistency of Isabelle/HOL).

The second more general interpretation required us to prove each of the assumptions of the locale `meekabbstract_carrier` for our concrete implementation. We showed that this is the case for all of the simpler assumptions (Section 5.4.2.2 through to Section 5.4.2.7) as well as the one more complex assumption, the votes invariant (Section 5.5.3). By doing this, we showed that the (locale) assumptions characterising Meek's method are general, as they apply for any arbitrary valid election context.

Finally, our proof of the votes invariant required proving a few key results connecting the set and sequence representations of ballots through the function *ranked* from Section 4.8.

One could with some additional work use what we have developed here to produce verified, executable Haskell, Scala, or ML code using Isabelle/HOL's own code extraction mechanism. It is not clear to us how easy our approach is to extend to produce formal certificates, of the sort discussed in the section on related work (see Section 5.7). As our approach is entirely functional, and does not proceed by state updates according to definite, named rules that one can easily capture as elements of a datatype, it would require some modification to support. Some aspects of what is needed for that particular approach (no doubt other approaches can also be fruitful) is more natural in dependently typed languages; consider for example the a protocol represented as a system of transitions embedded within a dependent type, alongside the fact that STVs can be viewed as transitioning back and forth between definite types of rounds. We leave further speculation about this aside as something perhaps worth

considering as future work.

This concludes our Isabelle/HOL formalisation. We have established the crucial fact that Meek's method of STV's uniquely involved surplus transfer round is correct, in that it converges on a unique solution vector. In the final chapter we will present avenues for future work and conclude with some final remarks.

# Chapter 6

# Conclusion and future work

We conclude in this chapter with some final remarks and a discussion of a number of promising avenues for future work.

## 6.1 Final remarks

In this thesis we have presented a formal verification of the correctness of the surplus transfer round of Meek's method for STV (Chapters (3, 5)). In so doing, we believe we have contributed a significant step towards formally verifying the whole method. Moreover, given that Meek's method is the most complicated variant of STV in practical use, our work also makes a notable contribution towards reasoning about the correctness of STV methods in general.

Noteworthy aspects of our work include the development of a theory of ballots (Chapter 4) suitable for reasoning about strictness using a set-based representation and with ballot induction, rankings, and their properties. Moreover, our novel representations for working with Meek's method should be useful for further analysis of STV methods.

Overall, we believe this work provides a general framework that others in the Automated Reasoning and Formal Verification community can build on to tackle related and broader challenges involving the formal verification of voting algorithms. We conclude by surveying the broad field of future work opened up by this development.

## 6.2 Future work, limitations, and challenges

In this section we briefly present some potential avenues for future work and discuss some of the associated challenges.

### 6.2.1 Full method verification

In this thesis we have principally focused on the surplus transfer round of Meek's method. Of course, for a fully verified Meek's method with code extraction, one would also have to tackle each of the other individual rounds: elimination, tie-breaking, initial allocation (more of a round in hand-counting than computer-counting), and mapping the final state to the set of elected candidates. After tackling the other individual rounds, which are all simple single-step operations, one would then have to deal with termination and correctness of the whole method.

Meek's method itself is not difficult to implement; we have implemented it in Isabelle/HOL but have not proven any important end-to-end properties of the overall method, such as full termination (i.e. termination of every round and the whole method).

#### 6.2.1.1 Termination

Termination overall would simply follow from proving that at every round, one of the following has to occur: one or more new candidates are elected, or a new candidate is eliminated. Through elimination, one moves closer to $S$ candidates being left. Through election, one heads closer to $S$ candidates being elected. One or the other has to meet $S$ eventually; if $S+1$ become elected a final tie-breaking round is of course necessary. The fractional Droop quota cannot elect more than $S+1$ candidates, so this is the only exceptional case that needs to be considered.

#### 6.2.1.2 Seat-filling

Trickier would be proving that Meek's method correctly fills all seats, meaning simply that the correct number of candidates is elected in the end and throughout the process once a candidate is marked as elected they remain elected throughout. One may also wish to prove as part of this for Meek's method that the final state being converged upon at the last surplus transfer round is one in which all non-eliminated candidates exactly meet the quota, perhaps alongside a theorem about "minimising excess" as Meek and others claim the method does (what this precisely means would need to be determined). We have considered what it would take to prove correct seat-filling, and we do think this could require a significant effort. For ordinary STV, the argument for termination is very similar to the argument for seat-filling: it is approached from both directions by election and elimination.

The problem is that Meek's method's surplus transfer round does not terminate when $V_c(w) = Q(w)$, for all elected $c$, but when $V_c(w) < Q(w) + \varepsilon$ for all elected $c$ (or equivalently one can sum all the surpluses and terminate when the sum is less than $\varepsilon$). We think that that this should not be a big issue. Consider the case where $\varepsilon$ is set so high that a surplus transfer round never occurs, then candidates are simply eliminated until $S$ are left. It thus seems likely that the same argument about an inevitable march towards seat-filling is true.

The Droop Proportionality Criterion (DPC) would also be an important property

to tackle if the work was expanded to more social-choice-theoretical results as part of a more general framework for STV. It is in theory true of all STV, and so not particularly important to focus on for work specifically focusing on Meek's method, though it would certainly be a good result to confirm for this specific case. For properties like the DPC (see Tideman [74], who notes the property as the "proportionality for solid coalitions"), which is a mathematical characterisation of whether a multi-winner method provides proportional representation in those seats specifically,[1] one may need to worry more about the role of $\varepsilon$ in *distorting* the "ideal" result. This brings us onto our second piece of future work.

### 6.2.2 Non-distortion

To prove properties like the proportionality criterion one may have to show that there is some sufficiently small $\varepsilon$ for any given election size such that the final outcome is the same as if one used the precise value for the solution vector.[2] Equivalently, that if one continued the iterative process, there is no later step at which the final outcome (not just the set of elected candidates at the end of the round) would change.

One can generalise the problem: show that, for all election sizes, i.e. for all bounds on $|B|$ and $|C|$, there is some sufficiently small $\varepsilon$ which cannot distort the result. This would be a remarkable result; while it is considered that distorted outcomes with an $\varepsilon$ of around 0.0001 are unlikely, it would be extremely beneficial for further trust in the method. Even more so if the proof of this were constructive, meaning one produced a function taking an election size and could produce an $\varepsilon$ which was non-distorting.

A related result that would be much easier to prove, by developing an example that can be scaled to any election size, would be to show that for any $\varepsilon$ there is some election with a distorted result. No existing work on the topic of distortion in Meek's method exists[3]; the approach has been one of fingers-crossed that election sizes typical for countries will not distort the result for an arbitrarily chosen $\varepsilon$ [40].

[1] We say "those seats specifically" because in practice, these are usually seats at just the constituency level, and STV is not applied to fill up an entire parliament. This is what arguably makes the method a compromise between proportional representation and local representation.

[2] Whether one gets this precise value for the solution vector by computational means, e.g. through solving it analytically, or non-computational means, e.g. through the choice function `THE`, is we think not relevant for this kind of proof.

[3] Work on the so-called "butterfly effect" [53, 42] in STV and Hill's response that it does not apply to Meek's method feels closest in spirit though has nothing to do with convergence and the stopping parameter.

### 6.2.3 Code extraction

Related to full method verification, code extraction for even just the surplus transfer round would be beneficial. What needs to be done, and it is unclear how much extra work this might be, is to connect up the implementation of the surplus transfer round with these results in such a way that code extraction is possible. Part of this will involve proving various functional equivalences because the vector type constructor `vec` that we use for analysis does not permit code extraction. Care would also have to be taken to manage the finiteness restriction on the type of candidates. Focusing on this plumbing work was simply too much of a time distraction from the more conceptually interesting and important results to be tackled in this thesis.

Nevertheless, given Isabelle's code extraction mechanism is sophisticated enough to provide some guarantees of at least partial correctness, through a shallow embedding of equivalent concepts [32] in various programming languages (Haskell, Scala, ML, OCaml) which treats generated code as a higher-order rewrite system represented using the intermediate language Mini-Haskell [33], it will be important work to pursue in order to popularise the results of this thesis.

### 6.2.4 Automation

Automating more proofs by providing powerful simplification sets or bespoke tactics would be beneficial to the future maintenance and expansion of the framework. There are least three possible focus areas:

- Ballots: connecting $\mathcal{G}$ with more general results about orderings, including an identification with the natural numbers so that sequence-like reasoning follows for a broader class of theorems without significant effort.

- Weight vectors: connecting sub-vectors, such as the weight vector considering only the candidates or only the elected candidates, with general ordering theorems. This would be useful for automation, but also for writing things like $(\forall c \in \mathcal{L}.\ w_c < w'_c)$ instead like $w <_{\mathcal{L}} w'$.

- Specific automation for the component functions of Meek's method, which takes into account things like: eliminated and unlisted candidates having no effect, hopeful candidates receiving all of what remains of a ballot, simplification routines to a canonical form for expressions involving definite weight decreases, and so on.

### 6.2.5 Generalisation

Warren's method, mentioned as an aside a couple of times in this thesis, is the same as Meek's method in terms of component functions but takes a different approach to updating the weights. Warren considered Meek's method for updating weights to be unfair, though acknowledged that most of the time the two methods will produce the same result. We can also point out that the point of the method for updating weights is to converge on the *unique* solution vector, so that the choice of update is only relevant when considering edge-cases, where stopping early distorts the result. It would thus be worthwhile to generalise the approach in Chapter 3 to cover any weight update function that is monotonically decreasing in the way necessary for convergence.

### 6.2.6 Verified optimisation

Optimisations for time and space complexity exist both for general methods of STV as well as Meek's method specifically. Any optimisation for Meek's method needs to be shown to be functionally equivalent to the simpler implementation, so that the latter can be used for code extraction while the former is used for analysis. For STV, one such optimisation is to eliminate candidates as soon as it is clear that they cannot become elected.

For Meek's method, as discussed in Section 1.1.1 and Section 5.2 and as noted by Hill [40], a common optimisation is to "enlist" candidates into the surplus transfer round as soon as they reach the quota. For this variant new candidates can become elected during the round, so instead of updating weights for $c \in \mathit{elected0}$, one updates as long as $V_c(w^{(i)}) \geq Q(w^{(i)})$. We determined that while this makes initial proof development in Chapter 3 and considerations around weight validity easier, it makes convergence much harder (as the non-triviality criterion may not be maintained, and the candidates involved in convergence dynamically changes). Hence, we followed Hill et al. and proved all of the results for the non-enlisting variant and leave proving their equivalence as future work.

For $\varepsilon = 0$, i.e. when solving the successive transfer rounds with the precise solution vector, we have developed an informal argument based on a proof by contradiction, considering the ways in which the two final vectors could differ, and then invoking solution-round uniqueness to derive a contradiction. Pursuing this argument formally could potentially be a fruitful avenue for research.

However, related to our comments on non-distortion, the property one would want

to prove is likely something more along the following lines: for all election sizes, there is an $\varepsilon$ such that the final outcome is the same whether one enlists candidates into the surplus transfer round or carries out successive rounds until there is no more surplus.

### 6.2.7 Rational weights

Meek's method can be entirely defined in terms of rational arithmetic. This includes the parameter $\varepsilon > 0$ which may just be chosen as an arbitrarily small rational. Given the fact that voting methods are only executed for elections, generally coming once every year at most, and that hand-counting typically takes hours, days, or weeks (depending on irregularities), we have considered it self-evident that implementing Meek's method using rational arithmetic is sensible. However, if it were the case that for very large elections calculation becomes time intensive due to many non-trivial greatest-common-divisor calculations, then floating-point approximation would have to be used.

But floating-point carries the danger of rounding which results in the total votes in circulation being non-constant throughout the execution of the method. The solution turned to in the open-source software OpenSTV as suggested in a paper by Lundell and Hill is quasi-exact arithmetic, which they suggest alongside other potential options, including:

> *"Use rational arithmetic, so that all values can be represented exactly. This approach is likely to be computationally expensive, and has not to our knowledge been implemented."* [50]

Our initial experiments show that there is reason for optimism, but we believe that it would be important to experimentally investigate the potential presence of scaling issues in using precise rational arithmetic for Meek's method applied to very large elections. If it is a problem, this immediately demands additional future work on verifying Meek's method on a subset of the rationals, whether floating-point, quasi-exact, or using something more sophisticated like dyadic rationals. The latter would likely be the most attractive option as it is a well-understood mathematical domain which has been used in existing Isabelle/HOL formalisations (e.g. Li et al. [49]).

### 6.2.8 Non-strict ballots

One could generalise *frac-of* to non-strict ballots with little difficulty. Proving all of the results in this thesis for non-strict ballots may, however, be more onerous than

is worth the payoff, seeing as non-strict ballots are rarely used for STV in practice. Nevertheless, we provide the equation one needs for *frac-of* to get this process started.

By introducing an additional function, $\mathcal{E}$, which takes a ballot and returns the sets of equally-listed candidates, we can write a simple closed form expression for *frac-of* and $E$ for non-strict ballots. For the ballot $ABC$, where $A$, $B$, and $C$ are sets of equally-listed candidates with for example $A = \{a_1, .., a_{n_A}\}$, we have

$$\mathcal{E}(ABC) = \{\{a_1, .., a_{n_A}\}, \{b_1, .., b_{n_B}\}, \{c_1, .., c_{n_C}\}\}$$

We also introduce $\mathcal{G}_{\mathcal{E}}$ which takes a ballot and a candidate $c$ and returns the sets of equally listed candidates above the equally-listed set which $c$ is in, for $ABC$ again: $\mathcal{G}_{\mathcal{E}}(ABC, c_1) = \{\{a_1, .., a_{n_A}\}, \{b_1, .., b_{n_B}\}\}$. And finally we introduce $\mathcal{L}_{\mathcal{E}}$ which takes a ballot and a candidate and returns those listed equally to it, for example $\mathcal{L}_{\mathcal{E}}(ABC, b_1) = \{b_1, .., b_{n_B}\}$.

$$\textit{frac-of}(b, c, w) = \begin{cases} \frac{\Pi_{K \in \mathcal{G}_{\mathcal{E}}(b,c)}(1 - \Sigma K)}{|\mathcal{L}_{\mathcal{E}}(b,c)|} & c \in \mathcal{L}(b) \\ 0 & otherwise \end{cases} \tag{6.1}$$

$$E(B, w) = \Sigma_{b \in B} \Pi_{K \in \mathcal{E}(b)}(1 - \Sigma K) \tag{6.2}$$

where $\Sigma K = \frac{\Sigma_{k \in K} w_k}{|K|}$.

### 6.2.9 Meek's method from first principles

Finally, a completely different but nevertheless interesting project, one which is impossible not to consider when pursuing a formalisation project such as this, is whether one could *derive* the form of Meek's method itself using Meek's two principles:

> **Principle 1.** If a candidate is eliminated, all ballots are treated *as if that candidate had never stood.*

> **Principle 2.** If a candidate has achieved the quota, he retains a fixed proportion of every vote received, and transfers the remainder to the next *non-eliminated* candidate, the retained total equalling the quota.

Given that the method is supposed to optimally implement the principles, something like this should be possible. We think that, starting with a very general characterisation of STV-likes, one could formalise these two principles, as well as other STV principles (see e.g. Aleskerov et al. [1]), and then prove that Meek's method satisfies the principles. Whether Meek's method is the *only* method to do so is another question. We

believe that we have identified that no STV principles, nor Meek's principles, justify the usual method of elimination of simply the candidate with the lowest votes, though more research needs to be done (indeed, CPO-STV and Schulze-STV were in-part invented to address this problem of premature elimination). One could also interrogate Warren's claim that his method satisfies its own, different principles, and does not satisfy those laid out by Meek.

It would be interesting to approach this question from both the discipline of mechanism design [56, 62], as well as by encoding the degree to which a method satisfies the various principles as a real number and applying traditional methods of optimisation and AI to evolving a method for vote-counting. Perhaps a combination of the two approaches similar to reinforcement mechanism design [72] might be worth considering.

# Appendix A

# Isabelle/HOL

## A.1 Definitions and induction rules

### A.1.1 Vectors

```
definition dec1 :: "(real, 'a) vec ⇒ 'a::finite ⇒ real ⇒ (real, '
    a) vec" ("_ ⌊ _,_" [100,100,100] 100) where
  "v ⌊ x,r ≡ χ y. if x = y then v $ x - r else v $ y"

definition vec_upd :: "('val, 'arg::finite) vec ⇒ 'arg ⇒ 'val ⇒
    ('val, 'arg) vec" ("_<_ ↦ _>" [100,100,100] 100) where
  "v<x ↦ val> ≡ vec_lambda ((vec_nth v)(x := val))"

definition repl_all :: "('b, 'a) vec ⇒ 'a::finite set ⇒ ('b, 'a)
    vec ⇒ ('b, 'a) vec" ("_<_⟼ _>" [100,100,100] 100) where
  "v<X⟼ v'> ≡ χ x. if x ∈ X then v' $ x else v $ x"

abbreviation repl_all_fn_N ("_<_|_⟼ _>" [100, 100, 100, 100] 100)
    where
  "w<n|X⟼ f> ≡ ((λw. w<X⟼ (χ c. f w c)>) ^^ n) w"

definition feasible :: "(real, 'c::finite) vec ⇒ (real, 'c) vec ⇒
    real ⇒ 'c set ⇒ 'c set ⇒ bool" where
  "feasible w V Q cands elected ≡
       ∀c∈cands. w $ c ≥ 0 ∧ w $ c ≤ 1 ∧ (c ∈ elected ⟶ V $ c ≥ Q)"

definition solution :: "(real, 'c::finite) vec ⇒ (real, 'c) vec ⇒
    real ⇒ 'c set ⇒ 'c set ⇒ bool" where
  "solution w V Q cands elected ≡
      ∀c∈cands. w $ c ≥ 0 ∧ w $ c ≤ 1 ∧ (c ∈ elected ⟶ V $ c = Q)"

definition feasible_given :: "(real, 'c::finite) vec ⇒ (real, '
    c::finite) vec ⇒ (real, 'c) vec ⇒ real ⇒ 'c set ⇒ 'c set ⇒
    bool" where
  "feasible_given w0 w V Q cands elected ≡
      feasible w V Q cands elected ∧ (∀c∈cands. c ∉ elected ⟶ w $
    c = w0 $ c)"
```

```
definition solution_given :: "(real, 'c::finite) vec ⇒ (real, '
    c::finite) vec ⇒ (real, 'c) vec ⇒ real ⇒ 'c set ⇒ 'c set ⇒
    bool" where
  "solution_given w0 w V Q cands elected ≡
      solution w V Q cands elected ∧
      (∀c∈cands. c ∉ elected ⟶ w $ c = w0 $ c)"
```

### A.1.2 Ballots

```
definition valid_strict_ballot :: "'c set ⇒ ('c ⇒ 'c set) ⇒
    bool" where
  "valid_strict_ballot ℒ 𝒢 ≡ finite ℒ ∧
     (∀x∈ℒ . 𝒢 x ⊆ ℒ ∧ x ∉ 𝒢 x ∧
        (∀y∈ℒ. (∀z∈ℒ. y ∈ 𝒢 x ∧ z ∈ 𝒢 y ⟶ z ∈ 𝒢 x) ∧
                (x ∈ 𝒢 y ∨ y ∈ 𝒢 x ∨ x = y)))"

definition subballot :: "'c set ⇒ ('c ⇒ 'c set) ⇒ 'c set ⇒ ('c
    ⇒ 'c set) ⇒ bool" where
  "subballot ℒ' 𝒢' ℒ 𝒢 ≡ ℒ' ⊂ ℒ ∧ (∀c∈ℒ'. 𝒢' c = 𝒢 c - (ℒ - ℒ'))"

lemma ballot_induct [consumes 1, case_names psubballot]:
  assumes valid: "valid_strict_ballot ℒ 𝒢"
      and major: "⋀ℒ 𝒢. ⟦valid_strict_ballot ℒ 𝒢;
                         ⋀ℒ' 𝒢'. subballot ℒ' 𝒢' ℒ 𝒢 ⟹ P ℒ' 𝒢'⟧
                        ⟹ P ℒ 𝒢"
  shows "P ℒ 𝒢"

definition above :: "'c ⇒ 'c set ⇒ ('c ⇒ 'c set) ⇒ 'c ⇒ bool"
    ("_ above[_,_] _" [100,100,100,100] 100) where
  "c above[ℒ,𝒢] c' ≡ c' ∈ ℒ ∧ insert c (𝒢 c) = 𝒢 c'"

definition below :: "'c ⇒ 'c set ⇒ ('c ⇒ 'c set) ⇒ 'c ⇒ bool"
    ("_ below[_,_] _" [100,100,100,100] 100) where
  "c below[ℒ,𝒢] c' ≡ c ∈ ℒ ∧ c' ∈ ℒ ∧ ℒ - 𝒢 c = (ℒ - 𝒢 c') - {c'}"

definition rank :: "('c ⇒ 'c set) ⇒ 'c ⇒ nat" where
  "rank 𝒢 c ≡ card (𝒢 c)"

definition ranks :: "'c set ⇒ ('c ⇒ 'c set) ⇒ ('c * nat) set"
    where
  "ranks ℒ 𝒢 ≡ (λc. (c, rank 𝒢 c)) ` ℒ"

definition ranked :: "'c set ⇒ ('c ⇒ 'c set) ⇒ nat ⇒ 'c" where
  "ranked ℒ 𝒢 n ≡ SOME c. c ∈ ℒ ∧ rank 𝒢 c = n"
```

### A.1.3 Meek's method

```
definition quota_hb' :: "nat ⇒ nat ⇒ real ⇒ real" where
  "quota_hb' T s E ≡ (T - E) / (s + 1)"
```

```
definition votes' :: "'b set ⇒ ('b ⇒ 'c ⇒ (real, 'c::finite) vec
    ⇒ real) ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite) vec"
    where
  "votes' B f w ≡ χ c. w $ c * (Σb∈B. f b c w)"

definition surplus' :: "(real, 'c::finite) vec ⇒ real ⇒ real"
    where
  "surplus' V Q ≡ Σc∈{k. V $ k > Q}. V $ c - Q"

definition quota_diff_vec' :: "(real, 'c::finite) vec ⇒ real ⇒
    (real, 'c::finite) vec" where
  "quota_diff_vec' V Q ≡ χ c. V $c - Q"

definition smallests :: "(real, 'c::finite) vec ⇒ 'c set" where
  "smallests v ≡ {c. ∀k. v $ c ≤ v $ k}"

definition ranked_last :: "'b ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c ⇒ 'c
    set) ⇒ 'c" where
  "ranked_last b ℒ 𝒢 ≡ THE c. 𝒢 b c = ℒ b - {c}"

definition excess :: "'b set ⇒ ('b ⇒ 'c set) ⇒ (real, '
    c::finite) vec ⇒ real" where
  "excess B ℒ w ≡ Σb∈B. Πc∈ℒ b. (1 - w $ c)"

definition quota_hb :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ (real, '
    c::finite) vec ⇒ real" where
  "quota_hb B s ℒ w ≡ quota_hb' (card B) s (excess B ℒ w)"

definition frac_of :: "'c set ⇒ ('c ⇒ 'c set) ⇒ 'c ⇒ (real, '
    c::finite) vec ⇒ real" where
  "frac_of ℒ 𝒢 c w ≡ if c ∈ ℒ then Πk∈𝒢 c. (1 - w $ k) else 0"

definition votes :: "'b set ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c ⇒ 'c
    set) ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite) vec" where
  "votes B ℒ 𝒢 w ≡ votes' B (λb. frac_of (ℒ b) (𝒢 b)) w"

definition reaches_quota :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ ('b
    ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ 'c set" where
  "reaches_quota B s ℒ 𝒢 w ≡
      {c. votes B ℒ 𝒢 w $ c ≥ quota_hb B s ℒ w}"

definition quota_diff_vec :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒
    ('b ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ (real, '
    c::finite) vec" where
  "quota_diff_vec B s ℒ 𝒢 w ≡
      quota_diff_vec' (votes B ℒ 𝒢 w) (quota_hb B s ℒ w)"

definition surplus :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c
    ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ real" where
  "surplus B s ℒ 𝒢 w ≡ surplus' (votes B ℒ 𝒢 w) (quota_hb B s ℒ w)"

definition eliminate :: "'b set ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c ⇒ '
    c set) ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite) vec" where
  "eliminate B ℒ 𝒢 w ≡
      w<(SOME c. w $ c > 0 ∧ c ∈ smallests (votes B ℒ 𝒢 w)) ↦ 0>"
```

```
definition solve :: "'c set ⇒ 'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒
    ('b ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ (real, '
    c::finite) vec" where
  "solve elected B s ℒ 𝒢 w ≡
      THE w_sol. (∀c∈elected. votes B ℒ 𝒢 w_sol $ c =
                              quota_hb B s ℒ w_sol) ∧
                              (∀c. c ∉ elected ⟶
                                    w_sol $ c = w $ c)"

function meek :: "'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ ('b ⇒ 'c ⇒ '
    c set) ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite) vec" where
  "meek B s ℒ 𝒢 w = (let reaches = reaches_quota B s ℒ 𝒢 w in
       if card reaches ≥ s then
           w
       else if surplus B s ℒ 𝒢 w > 0 then
           meek B s ℒ 𝒢 (solve reaches B s ℒ 𝒢 w)
       else
           meek B s ℒ 𝒢 (eliminate B ℒ 𝒢 w)
   )"

abbreviation update_one :: "'c set ⇒ 'b set ⇒ nat ⇒ ('b ⇒ 'c
    set) ⇒ ('b ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ (real,
    'c::finite) vec" where
  "update_one elected B s ℒ 𝒢 w ≡
      χ c. if c ∈ elected then
             w $ c * quota_hb B s ℒ w / votes B ℒ 𝒢 w $ c
           else
             w $ c"

function ε_solve :: "real ⇒ 'c set ⇒ 'b set ⇒ nat ⇒ ('b ⇒ 'c
    set) ⇒ ('b ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ (real,
    'c::finite) vec" where
  "ε_solve ε elected B s ℒ 𝒢 w = (
      if surplus B s ℒ 𝒢 w > ε then
        ε_solve ε elected B s ℒ 𝒢 (update_one elected B s ℒ 𝒢 w)
      else
        w)"

function ε_meek :: "real ⇒ 'b set ⇒ nat ⇒ ('b ⇒ 'c set) ⇒ ('b
    ⇒ 'c ⇒ 'c set) ⇒ (real, 'c::finite) vec ⇒ (real, 'c::finite)
    vec" where
  "ε_meek ε B s ℒ 𝒢 w = (let reaches = reaches_quota B s ℒ 𝒢 w in
       if card reaches ≥ s then
           w
       else if surplus B s ℒ 𝒢 w > ε then
           ε_meek ε B s ℒ 𝒢 (ε_solve ε reaches B s ℒ 𝒢 w)
       else
           ε_meek ε B s ℒ 𝒢 (eliminate B ℒ 𝒢 w))"
```

## A.2 Locales and contextual definitions

### A.2.1 Elections

```
locale election_context =
  fixes cands :: "'c set"
    and ballots :: "'b set"
    and ℒ :: "'b ⇒ 'c set"
    and 𝒢 :: "'b ⇒ 'c ⇒ 'c set"
    and listed :: "'c set"
  defines listed_def: "listed ≡ ⋃{ℒ b |b. b ∈ ballots}"
  assumes all_valid:
              "b ∈ ballots ⟹ valid_strict_ballot (ℒ b) (𝒢 b)"
      and all_nonempty: "b ∈ ballots ⟹ ℒ b ≠ {}"
      and ballots_nonempty: "ballots ≠ {}"
      and finite_ballots: "finite ballots"
      and finite_cands: "finite cands"
      and listed_cands: "listed ⊆ cands"
begin

definition ballot_eq :: "'c ⇒ 'b ⇒ 'c ⇒ bool" ("_ =_ _" [100,
    100, 100] 100) where
  "x =b y ≡ x = y ∧ x ∈ ℒ b ∧ y ∈ ℒ b"

definition ballot_gt :: "'c ⇒ 'b ⇒ 'c ⇒ bool" ("_ >_ _" [100,
    100, 100] 100) where
  "x >b y ≡ x ∈ ℒ b ∧ y ∈ ℒ b ∧ x ∈ 𝒢 b y"

definition ballot_le :: "'c ⇒ 'b ⇒ 'c ⇒ bool" ("_ ≤_ _" [100,
    100, 100] 100) where
  "x ≤b y ≡ x =b y ∨ y ∈ ℒ b ∧ x ∈ (ℒ b - 𝒢 b y)"

definition ballot_lt :: "'c ⇒ 'b ⇒ 'c ⇒ bool" ("_ <_ _" [100,
    100, 100] 100) where
  "x <b y ≡ x ≠ y ∧ x ≤b y"

definition ballot_ge :: "'c ⇒ 'b ⇒ 'c ⇒ bool" ("_ ≥_ _" [100,
    100, 100] 100) where
  "x ≥b y ≡ x =b y ∨ x >b y"

definition ranked_first :: "'b ⇒ 'c ⇒ bool" where
  "ranked_first b y ≡ ∀x∈ℒ b. x ≤b y"

definition ranked_last :: "'b ⇒ 'c ⇒ bool" where
  "ranked_last b x ≡ ∀y∈ℒ b. x ≤b y"

end
```

### A.2.2 Meek's method

```
locale abstract_meek_carrier =
  fixes V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec"
    and Q_for :: "(real, 'c) vec ⇒ real"
    and c1 :: nat
    and c2 :: real
    and E_for :: "(real, 'c) vec ⇒ real"
    and num_ballots :: "nat"
    and seats :: "nat"
```

```
    and cands :: "'c set"
  assumes
      V_change: "⟦c ∈ cands; w $ c ≠ 0⟧ ⟹
                 V_for (w ⌊c,r) $ c =
                 V_for w $ c * (1 - r / w $ c)"
      and V_winc: "⟦c ∈ cands; c' ∈ cands; c' ≠ c; w $ c' ≥ 0;
                    r ≥ 0; ⋀k. k ∈ cands ⟹ w $ k ≤ 1⟧ ⟹
                 V_for (w ⌊c,r) $ c' ≥ V_for w $ c'"
      and E_winc: "⟦c ∈ cands; r ≥ 0; ⋀k. k ∈ cands ⟹ w $ k ≤ 1⟧ ⟹
                 E_for (w ⌊c,r) ≥ E_for w"
      and E_lower: "⟦⋀c. c ∈ cands ⟹ w $ c ≥ 0;
                    ⋀c. c ∈ cands ⟹ w $ c ≤ 1⟧ ⟹
                  E_for w ≥ num_ballots * (∏c∈cands. 1 - w $ c)"
      and E_upper: "⟦c ∈ cands; ⋀c. c ∈ cands ⟹ w $ c ≥ 0;
                     ⋀c. c ∈ cands ⟹ w $ c ≤ 1;
                     ⋀k. k ∈ cands ⟹ w $ c ≤ w $ k⟧ ⟹
                   E_for w ≤ num_ballots * (1 - w $ c)"
      and self_winc: "⟦c ∈ cands; w $ c ≥ 0;
                      ⋀c. c ∈ cands ⟹ w $ c ≤ 1; r ≤ 0⟧ ⟹
                  V_for w $ c ≤ V_for (w ⌊c,r) $ c"
      and votes_invariant:
          "num_ballots = (Σk∈cands. V_for w $ k) + E_for w"
      and quota_form:
          "Q_for = (λw. (num_ballots - E_for w) / (seats + c1) +
                     c2) ∧ c2 ≥ 0 ∨
           Q_for = (λw. real_of_int ⌊(num_ballots - E_for w) /
                     (seats + c1)⌋ + c2) ∧ c2 > 0"
      and num_ballots_gt0: "num_ballots > 0"
      and seats_gt0: "seats > 0"
      and noncand_no_V_change: "⟦c ∈ cands; c' ∉ cands⟧ ⟹
                                V_for (w ⌊c',r) $ c = V_for w $ c"
      and noncand_no_Q_change: "c ∉ cands ⟹
                                Q_for (w ⌊c,r) = Q_for w"

locale meektransfer_fixes_carrier = abstract_meek_carrier V_for
  for V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec" +
  fixes transfer_weights :: "(real, 'c) vec"
    and elected0 :: "'c set"
  assumes elected_cands [simp]: "elected0 ⊆ cands"
begin

abbreviation update_one where
  "update_one w ≡
      w<elected0 ⟼ (χ c. w $ c * Q_for w / V_for w $ c)>"

definition w_at :: "nat ⇒ (real, 'c) vec" where
  "w_at i ≡ (update_one^^i) transfer_weights"

abbreviation V_at :: "nat ⇒ (real, 'c) vec" where
  "V_at i ≡ V_for (w_at i)"

abbreviation E_at :: "nat ⇒ real" where
  "E_at i ≡ E_for (w_at i)"

abbreviation Q_at :: "nat ⇒ real" where
  "Q_at i ≡ Q_for (w_at i)"
```

```
abbreviation feasible_for :: "(real, 'c::finite) vec ⇒ bool" where
  "feasible_for w ≡ feasible w (V_for w) (Q_for w) cands elected0"

abbreviation solution_for :: "(real, 'c::finite) vec ⇒ bool" where
  "solution_for w ≡ solution w (V_for w) (Q_for w) cands elected0"

abbreviation feasible_given_for :: "(real, 'c::finite) vec ⇒
    (real, 'c::finite) vec ⇒ bool" where
  "feasible_given_for w0 w ≡
      feasible_given w0 w (V_for w) (Q_for w) cands elected0"

abbreviation solution_given_for :: "(real, 'c::finite) vec ⇒
    (real, 'c::finite) vec ⇒ bool" where
  "solution_given_for w0 w ≡
      solution_given w0 w (V_for w) (Q_for w) cands elected0"

abbreviation feasible_at :: "nat ⇒ bool" where
  "feasible_at i ≡ feasible_for (w_at i)"

abbreviation solution_at :: "nat ⇒ bool" where
  "solution_at i ≡ solution_for (w_at i)"

abbreviation feasible_given_at :: "nat ⇒ bool" where
  "feasible_given_at i ≡ feasible_given_for (w_at 0) (w_at i)"

abbreviation solution_given_at :: "nat ⇒ bool" where
  "solution_given_at i ≡ solution_given_for (w_at 0) (w_at i)"

abbreviation surplus_at :: "'a ⇒ nat ⇒ real" where
  "surplus_at c n ≡ V_at n $ c - Q_at n"

end

locale meektransfer_carrier = meektransfer_fixes_carrier _ _ _ _ _
    _ _ V_for
  for V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec" +
  assumes elected_nonempty: "elected0 ≠ {}"
      and seats_not_exceeded: "card elected0 ≤ seats"
      and feasible0: "feasible_at 0"
      and E_cont: "isCont E_for w"
      and V_cont: "isCont V_for w"
      and nonelected_nonnegative: "⟦c ∈ cands; c ∉ elected0⟧ ⟹
                                   V_at 0 $ c ≥ 0"
      and vote_sum_ge1:
          "⟦⋀c. c ∈ cands ⟹ w $ c ≥ 0;
            ⋀c. c ∈ cands ⟹ w $ c ≤ 1⟧ ⟹
              (Σc∈cands. V_for ((w_at 0)<elected0⟼ w>) $ c) ≥ 1"

locale meektransfer_strictinc_carrier = meektransfer_carrier +
  assumes many_dec_V_sum_dec:
    "⋀w w_le. ⟦
       X ⊆ cands;
       X ≠ {};
       ⋀c. c ∈ X ⟹ w_le $ c < w $ c;
       ⋀c. ⟦c ∈ cands; c ∉ X⟧ ⟹ w_le $ c = w $ c;
```

```
       ⋀c. c ∈ cands ⟹ w_le $ c ≥ 0;
       ⋀c. c ∈ cands ⟹ w $ c ≤ 1;
       ⋀c. ⟦c ∈ cands; V_for w $ c ≤ 0⟧ ⟹ w_le $ c = w $ c⟧ ⟹
     (Σc∈X. V_for w_le $ c) < (Σc∈X. V_for w $ c)"

locale meektransfer_excess_inc_carrier =
    meektransfer_strictinc_carrier _ _ E_for
  for E_for :: "(real, 'c::finite) vec ⇒ real" +
  assumes E_inc_E_inc:
    "⟦X ⊆ Y; Y ⊆ cands;
      ⋀c. c ∈ cands ⟹ w_lt' $ c ≥ 0;
      ⋀c. c ∈ cands ⟹ w $ c ≤ 1;
      ⋀c. c ∈ cands-X ⟹ w $ c = w_lt $ c;
      ⋀c. c ∈ cands-Y ⟹ w_lt $ c = w_lt' $ c;
      ⋀c. c ∈ X ⟹ w_lt $ c < w $ c;
      ⋀c. c ∈ Y ⟹ w_lt' $ c < w_lt $ c;
      E_for w_lt > E_for w⟧ ⟹ E_for w_lt' > E_for w_lt"

locale meekelimination_carrier = abstract_meek_carrier V_for
  for V_for :: "(real, 'c::finite) vec ⇒ (real, 'c) vec" +
  fixes weights :: "(real, 'c) vec"
    and pick :: "'c set ⇒ 'c"
  assumes eliminatable: "∃c∈cands. weights $ c > 0"
      and w_g0: "c ∈ cands ⟹ weights $ c ≥ 0"
      and w_le1: "c ∈ cands ⟹ weights $ c ≤ 1"
      and valid_pick: "⟦X ⊆ cands; X ≠ {}⟧ ⟹ pick X ∈ X"
```

## A.3 Key theorems

### A.3.1 Theorem 1.1: all steps of the surplus transfer round are feasible

```
theorem all_feasible:
  "feasible_at i"

corollary all_feasible_given:
  "feasible_given_at i"

corollary V_sum_pos:
  "(Σc∈cands. V_at i $ c) ≥ 1"

corollary E_lt_max:
  "E_at i ≤ num_ballots - 1"

corollary Q_frac_ge:
  assumes "Q_at i = (num_ballots - E_at i) / (seats + c1) + c2"
  shows "Q_at i ≥ 1 / (seats + c1) + c2"

corollary Q_int_ge:
  assumes "Q_at i = ⌊(num_ballots - E_at i) / (seats + c1)⌋ + c2"
  shows "Q_at i ≥ c2"
```

```
corollary Q_pos:
  "Q_at i > 0"

corollary elected0_Vi_ge_Qi:
  assumes "c ∈ elected0"
  shows "V_at i $ c ≥ Q_at i"

corollary w_ge0:
  assumes "c ∈ cands"
  shows "w_at i $ c ≥ 0"

corollary w_le1:
  assumes "c ∈ cands"
  shows "w_at i $ c ≤ 1"
```

### A.3.2 Theorem 1.2: the sequence of weight vectors in the surplus transfer round converges

```
lemma monoconv_dec:
  assumes "bdd_below (range f)"
      and "decseq f"
  shows "convergent (f::nat ⇒ '
    a::{conditionally_complete_linorder,linorder_topology})"

lemma w_bdd_below:
  "bdd_below (range (λi. w_at i $ c))"

lemma w_decseq:
  "decseq (λi. w_at i $ c)"

theorem w_convergent:
  "convergent (λi. w_at i $ c)"

definition w_inf :: "(real, 'c) vec" where
  "w_inf ≡ χ c. (THE l. (λi. w_at i $ c)⟶l)"
```

### A.3.3 Theorem 2: the transfer round converges on a unique solution vector

```
theorem unique_solution:
  assumes gfq: "Q_for = (λw. (num_ballots - E_for w) / (seats + c1)
    + c2)"
      and "c1 > 0"
  shows "∃w. solution_given_for (w_at i) w
        ∧ (∀w'. solution_given_for (w_at i) w' ⟶
             (∀c∈cands. w' $ c = w $ c))"
```

#### A.3.3.1 Additional theorems

```
theorem surplus_sum_dec_general:
  assumes gfq: "Q_for =
                  (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
      and w_props: "⋀c. c ∈ cands-elected0 ⟹ w_le $ c = w $ c"
                   "⋀c. c ∈ elected0 ⟹ w_le $ c < w $ c"
                   "⋀c. c ∈ cands ⟹ w_le $ c ≥ 0"
                   "⋀c. c ∈ cands ⟹ w $ c ≤ 1"
      and elected_V_gt0: "⋀c. c ∈ elected0 ⟹ V_for w $ c > 0"
      and c1_gt0: "c1 > 0"
  shows "(Σc∈elected0. V_for w_le $ c - Q_for w_le) <
         (Σc∈elected0. V_for w $ c - Q_for w)"

theorem surplus_sum_always_dec:
  assumes gfq: "Q_for =
                  (λw. (num_ballots - E_for w) / (seats + c1) + c2)"
      and Q_dec: "Q_at (i + 1) < Q_at i"
      and i_lt_j: "i + 1 < j"
      and j_lt_k: "j < k"
      and c1_gt0: "c1 > 0"
  shows "(Σc∈elected0. surplus_at c k) <
         (Σc∈elected0. surplus_at c j)"
```

### A.3.4 Necessary theorems for proving the implementation models the locales

#### A.3.4.1 Non-candidates do not affect components

```
theorem noncand_no_E_change:
  assumes "c ∉ all_in B ℒ"
  shows "excess B ℒ (w ⌊ c,r) = excess B ℒ w"

theorem noncand_no_Q_change:
  assumes "c ∉ all_in B ℒ"
  shows "quota_hb B s ℒ (w ⌊ c,r) = quota_hb B s ℒ w"

theorem noncand_no_V_change:
  assumes "c ∈ all_in B ℒ"
      and "c' ∉ all_in B ℒ"
      and G_sub_L: "⋀b c. ⟦b ∈ B; c ∈ ℒ b⟧ ⟹ 𝒢 b c ⊆ ℒ b"
  shows "votes B ℒ 𝒢 (w ⌊ c',r) $ c = votes B ℒ 𝒢 w $ c"
```

#### A.3.4.2 Upper and lower bounds on excess

```
theorem E_lower:
  assumes ge0: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≤ 1"
  shows "excess B ℒ w ≥ card B * (∏c∈all_in B ℒ. 1 - w $ c)"

theorem E_upper:
```

```
  assumes c_le1: "w $ c ≤ 1"
      and ge0: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≤ 1"
      and c_min: "⋀k. k ∈ all_in B ℒ⟹ w $ c ≤ w $ k"
      and "finite B"
      and ballots_nonempty: "⋀b. b ∈ B⟹ ℒ b ≠ {}"
  shows "excess B ℒ w ≤ card B * (1 - w $ c)"
```

### A.3.4.3 Single changes to weights

```
theorem V_change:
  assumes self_not_gt: "⋀b. ⟦b ∈ B; c ∈ ℒ b⟧⟹ c ∉ 𝒢 b c"
      and wnon0: "w $ c ≠ 0"
  shows "votes B ℒ 𝒢 (w ⌊c,r) $ c =
         votes B ℒ 𝒢 w $ c * (1 - r / w $ c)"

theorem V_winc:
  assumes "c' ≠ c"
      and "w $ c' ≥ 0"
      and "r ≥ 0"
      and le1: "⋀b k. ⟦b ∈ B; c' ∈ ℒ b; k ∈ 𝒢 b c'-{c}⟧⟹
                w $ k ≤ 1"
  shows "votes B ℒ 𝒢 (w ⌊c,r) $ c' ≥
         votes B ℒ 𝒢 w $ c'"

theorem E_winc:
  assumes "r ≥ 0"
      and le1: "⋀b k. ⟦b ∈ B; c ∈ ℒ b; k ∈ ℒ b; k ≠ c⟧⟹ w $ k ≤ 1"
      and "finite B"
  shows "excess B ℒ (w ⌊c,r) ≥ excess B ℒ w"

theorem self_winc:
  assumes "w $ c ≥ 0"
      and le1: "⋀c. c ∈ all_in B ℒ⟹ w $ c ≤ 1"
      and "r ≤ 0"
      and not_gt_self: "⋀b. ⟦b ∈ B; c ∈ ℒ b⟧⟹ c ∉ 𝒢 b c"
      and gt_listed: "⋀b c k. ⟦b ∈ B; c ∈ 𝒢 b k; k ∈ ℒ b⟧⟹
                      c ∈ ℒ b"
  shows "votes B ℒ 𝒢 w $ c ≤ votes B ℒ 𝒢 (w ⌊c,r) $ c"
```

### A.3.4.4 Components' continuity

```
theorem excess_cont:
  fixes w :: "(real, 'c::finite) vec"
  assumes "finite B"
  shows "isCont (excess B ℒ) w"

theorem V_cont:
  assumes "finite B"
  shows "isCont (votes B ℒ 𝒢) w"
```

#### A.3.4.5 Votes invariant

```
theorem votes_invariant:
  assumes "B ⊆ ballots"
  shows "card B = (Σk∈all_in B ℒ. votes B ℒ 𝒢 w $ k) + excess B ℒ w"
```

### A.3.5 General locale interpretation

```
locale election_with_seats = election ballots ℒ 𝒢
  for ballots :: "'b set"
  and ℒ :: "'b ⇒ 'c::finite set"
  and 𝒢 :: "'b ⇒ 'c ⇒ 'c set" +
  fixes seats :: nat
  assumes seats_gt0: "seats > 0"
begin

interpretation meek_interp: abstract_meek_carrier "votes ballots (ℒ
    :: 'b ⇒ 'c set) 𝒢" "quota_hb ballots seats ℒ" "1 :: nat" "0 ::
    real" "excess ballots ℒ" "card ballots" seats "all_in ballots ℒ"

abbreviation meek_interp_on where
  "meek_interp_on bals ≡ abstract_meek_carrier (votes bals (ℒ :: 'b
    ⇒ 'c set) 𝒢) (quota_hb bals seats ℒ) (1 :: nat) (0 :: real)
    (excess bals ℒ) (card bals) seats (all_in bals ℒ)"

lemma subset_interp:
  assumes "bals ⊆ ballots"
      and "bals ≠ {}"
  shows "meek_interp_on bals"

end
```

# Bibliography


[1] Aleskerov, F. and Karpov, A. (2013). A new single transferable vote method and its axiomatic justification. Social Choice and Welfare, 40(3):771–786.

[2] Allen, S. F., Constable, R. L., Eaton, R., Kreitz, C., and Lorigo, L. (2000). The Nuprl open logical environment. In Automated Deduction-CADE-17: 17th International Conference on Automated Deduction Pittsburgh, PA, USA, June 17-20, 2000. Proceedings 17, pages 170–176. Springer.

[3] Arnon, D. S., Collins, G. E., and McCallum, S. (1984). Cylindrical algebraic decomposition I: The basic algorithm. SIAM Journal on Computing, 13(4):865–877.

[4] Arrow, K. J. (1950). A difficulty in the concept of social welfare. Journal of political economy, 58(4):328–346.

[5] Arrow, K. J., Sen, A., and Suzumura, K. (2010). Handbook of social choice and welfare, volume 2. Elsevier.

[6] Ballarin, C. (2006). Interpretation of locales in Isabelle: Theories and proof contexts. In International Conference on Mathematical Knowledge Management, pages 31–43. Springer.

[7] Ballarin, C. (2014). Locales: A module system for mathematical theories. Journal of Automated Reasoning, 52(2):123–153.

[8] Beckert, B., Bormer, T., Goré, R., Kirsten, M., and Schürmann, C. (2017). An Introduction to Voting Rule Verification. Trends in Computational Social Choice, page 269.

[9] Bertot, Y. and Castéran, P. (2013). Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media.

[10] Biere, A., Heule, M., and van Maaren, H. (2009). Handbook of satisfiability, volume 185. IOS press.

[11] Bishop, M. and Wagner, D. (2007). Risks of e-voting. Communications of the ACM, 50(11):120–120.

[12] Blanchette, J. C., Böhme, S., and Paulson, L. C. (2013). Extending sledgehammer with SMT solvers. Journal of automated reasoning, 51(1):109–128.

[13] Blanchette, J. C., Bulwahn, L., and Nipkow, T. (2011). Automatic proof and disproof in Isabelle/HOL. In International Symposium on Frontiers of Combining Systems, pages 12–27. Springer.

[14] Bove, A., Dybjer, P., and Norell, U. (2009). A brief overview of Agda - a functional language with dependent types. In TPHOLs, volume 5674, pages 73–78. Springer.

[15] Brandt, F., Conitzer, V., Endriss, U., Procaccia, A. D., and Lang, J. (2016). Handbook of computational social choice. Cambridge University Press.

[16] Church, A. (2013). Russelian simple type theory. The American Philosophical Association Centennial Series, pages 231–244.

[17] Dawson, J. E., Goré, R., and Meumann, T. (2015). Machine-checked reasoning about complex voting schemes using higher-order logic. In International Conference on E-Voting and Identity, pages 142–158. Springer.

[18] de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. (2015). The Lean theorem prover (system description). In Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25, pages 378–388. Springer.

[19] Droop, H. R. (1881). On methods of electing representatives. Journal of the Statistical Society of London, 44(2):141–202.

[20] Eberl, M. (2016a). The Incompatibility of SD-Efficiency and SD-Strategy-Proofness. Archive of Formal Proofs.

[21] Eberl, M. (2016b). Randomised Social Choice Theory. Archive of Formal Proofs.

[22] Endriss, U. (2017). Trends in Computational Social Choice. Lulu. com.

[23] Farrell, D. M. and McAllister, I. (2003). The 1983 change in surplus vote transfer procedures for the Australian senate and its consequences for the single transferable vote. Australian Journal of Political Science, 38(3):479–491.

[24] Ghale, M. K. (2019). Verified and Verifiable Computation with STV Algorithms. PhD thesis, The Australian National University (Australia).

[25] Ghale, M. K., Goré, R., and Pattinson, D. (2017). A formally verified single transferable voting scheme with fractional values. In International Joint Conference on Electronic Voting, pages 163–182. Springer.

[26] Ghale, M. K., Goré, R., Pattinson, D., and Tiwari, M. (2018). Modular formalisation and verification of STV algorithms. In International Joint Conference on Electronic Voting, pages 51–66. Springer.

[27] Ghale, M. K., Pattinson, D., and Norrish, M. (2019). Modular synthesis of verified verifiers of computation with STV algorithms. In Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, pages 85–94. IEEE Press.

[28] Gibbard, A. and others (1973). Manipulation of voting schemes: a general result. Econometrica, 41(4):587–601.

[29] Gordon, M. (2000). From LCF to HOL: a short history. In Proof, Language, and Interaction, pages 169–186.

[30] Guard, J. R., Oglesby, F. C., Bennett, J. H., and Settle, L. G. (1969). Semi-automated mathematics. Journal of the ACM (JACM), 16(1):49–62.

[31] Haftmann, F. (2013). Haskell-style type classes with Isabelle/Isar.

[32] Haftmann, F. and Bulwahn, L. (2022). Code generation from Isabelle/HOL theories. https://isabelle.in.tum.de/doc/codegen.pdf.

[33] Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems. In Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings 10, pages 103–117. Springer.

[34] Hales, T. C. (2005). A proof of the Kepler conjecture. Annals of mathematics, pages 1065–1185.

[35] Hales, T. C. (2006). Introduction to the Flyspeck project. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[36] Hare, T. (1857). The machinery of representation. W. Maxwell.

[37] Harrison, J. (2009a). HOL light: An overview. In Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22, pages 60–66. Springer.

[38] Harrison, J. (2009b). Without loss of generality. In International Conference on Theorem Proving in Higher Order Logics, pages 43–59. Springer.

[39] Harrison, J., Urban, J., and Wiedijk, F. (2014). History of interactive theorem proving. In Computational Logic, volume 9, pages 135–214.

[40] Hill, I. (2006). Implementing stv by meek's method. Voting Matters, 22:7–10.

[41] Hill, I., Wichmann, B., and Woodall, D. (1987). Algorithm 123: Single transferable by meek's method. The Computer Journal, 30(3):276–281.

[42] Hill, I. D. (2008). Miller's example of the butterfly effect under STV. Electoral Studies, 27(4):684–686.

[43] Hindley, J. R. (1997). Basic simple type theory. Cambridge University Press.

[44] Kammüller, F. (2000). Modular reasoning in Isabelle. In Automated Deduction-CADE-17: 17th International Conference on Automated Deduction Pittsburgh, PA, USA, June 17-20, 2000. Proceedings 17, pages 99–114. Springer.

[45] Kammüller, F., Wenzel, M., and Paulson, L. C. (1999). Locales: a sectioning concept for Isabelle. In Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs' 99 Nice, France, September 14–17, 1999 Proceedings 12, pages 149–165. Springer.

[46] Klein, G., Nipkow, T., and Paulson, L. (2003). The archive of formal proofs. https://www.isa-afp.org/.

[47] Kobayashi, H., Chen, L., and Murao, H. (2004). Groups, rings and modules. Archive of Formal Proofs. https://isa-afp.org/entries/Group-Ring-Module.html, Formal proof development.

[48] Kremer, S., Ryan, M., Smyth, B., et al. (2010). Election verifiability in electronic voting protocols. In ESORICS, volume 10, pages 389–404. Springer.

[49] Li, W., Passmore, G. O., and Paulson, L. C. (2019). Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. Journal of Automated Reasoning, 62:69–91.

[50] Lundell, J. and Hill, I. (2007). Notes on the Droop quota. Voting matters, 24:3–6.

[51] Meek, B. L. (1994a). A New Approach to the Single Transferable Vote (Paper 1, English version). Voting Matters.

[52] Meek, B. L. (1994b). A New Approach to the Single Transferable Vote (Paper 2, English version). Voting Matters.

[53] Miller, N. R. (2007). The butterfly effect under STV. Electoral Studies, 26(2):503–506.

[54] Miller, N. R. (2012). Monotonicity failure in irv elections with three candidates. In World meeting of the public choice societies.

[55] Moses, L. B., Goré, R., Levy, R., Pattinson, D., and Tiwari, M. (2017). No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes. In International Joint Conference on Electronic Voting, pages 66–83. Springer.

[56] Myerson, R. B. (1989). Mechanism design. In Allocation, information and markets, pages 191–206. Springer.

[57] New Zealand Government (2001). Local Electoral Act 2001, Schedule 1A.

[58] Newland, R. A. (1984). The STV quota. Representation.

[59] Newland, R. A. and Britton, F. S. (1997). How to conduct an election by the Single Transferable Vote. Electoral Reform Society of Great Britain and Ireland.

[60] Nipkow, T. (2009). Social choice theory in HOL. Journal of Automated Reasoning, 43(3):289–304.

[61] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). Isabelle/HOL: a proof assistant for higher-order logic, volume 2283. Springer Science & Business Media.

[62] Nisan, N. and Ronen, A. (2001). Algorithmic mechanism design. Games and Economic behavior, 35(1-2):166–196.

[63] O'Neill, Jeff (2019a). About OpaVote. OpaVote.

[64] O'Neill, Jeff (2019b). OpenSTV is now OpaVote. https://web.archive.org/web/*/https://www.opavote.com/openstv.

[65] Ornstein, J. T. and Norman, R. Z. (2014). Frequency of monotonicity failure under instant runoff voting: estimates based on a spatial model of elections. Public Choice, 161:1–9.

[66] Palmer, J. (2023). A copy of the development relevant to the thesis. https://github.com/JakeEP/Thesis. [Online, accessed 06-June-2023].

[67] Pattinson, D. and Schürmann, C. (2015). Vote counting as mathematical proof. In Australasian Joint Conference on Artificial Intelligence, pages 464–475. Springer.

[68] Schmoetten, R., Palmer, J. E., and Fleuriot, J. D. (2022). Towards formalising Schutz' axioms for Minkowski spacetime in Isabelle/HOL. Journal of Automated Reasoning, 66(4):953–988.

[69] Schulze, M. (2004). Free riding. Voting matters, 18(2-5):79.

[70] Schulze, M. (2018). The schulze method of voting. arXiv preprint arXiv:1804.02973.

[71] Slind, K. and Norrish, M. (2008). A brief overview of HOL4. In Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21, pages 28–32. Springer.

[72] Tang, P. (2017). Reinforcement mechanism design. In IJCAI, pages 5146–5150.

[73] The Independent (2003). Cosmic particles can change elections and cause planes to fall through the sky, scientists warn. https://www.independent.co.uk/news/science/subatomic-particles-cosmic-rays-computers-change-elections-planes-autopilot-a7584616.html. [Online, accessed 06-June-2023].

[74] Tideman, N. (1995). The single transferable vote. Journal of Economic Perspectives, 9(1):27–38.

[75] Tideman, N. (2017). Collective decisions and voting: the potential for public choice. Routledge.

[76] Various authors (2023). Isabelle documentation. https://isabelle.in.tum.de/documentation.html. [Online, accessed 06-June-2023].

[77] Verity, F. and Pattinson, D. (2017). Formally verified invariants of vote counting schemes. In Proceedings of the Australasian Computer Science Week Multiconference, page 31. ACM.

[78] Warren, C. (1994). Counting in STV elections. Voting matters, 1:12–13.

[79] Wenzel, M. (2007). Isabelle/Isar—a generic framework for human-readable proof documents. From Insight to Proof—Festschrift in Honour of Andrzej Trybulec, 10(23):277–298.

[80] Wenzel, M. (2021). The Isabelle System Manual. https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/system.pdf. [Online; accessed 19-July-2008].

[81] Wenzel, M., Paulson, L. C., and Nipkow, T. (2008). The Isabelle framework. In International Conference on Theorem Proving in Higher Order Logics, pages 33–38. Springer.

[82] Wichmann, B. (2000). Validation of Implementation of the Meek Algorithm for STV. Voting matters.

[83] Woodall, D. R. (1997). Monotonicity of single-seat preferential election rules. Discrete Applied Mathematics, 77(1):81–98.