UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Research project report title page

Candidate **2486B**

"Formalization of the Gale-Shapley Stable Matching Algorithm in Isabelle/HOL"

Submitted in partial fulfilment of the requirements for the Computer Science Tripos, Part III

Graduate Education Office, Room FS03
Department of Computer Science and Technology, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD

Abstract

The Gale-Shapley algorithm is the best-known solution to the stable matching problem. Proofs for its termination and correctness exist on paper but not in machine-checked, formalized form. This is the first complete formalization of the Gale-Shapley algorithm, which amounts to a constructive proof in Isabelle/HOL that a solution always exists for the stable matching problem.

Wordcount: 10,965

Contents

1 Introduction

Formal verification is the gold-standard for verifying the correctness of software and hardware systems. In the context of algorithms, this involves obtaining a mathematical proof for the algorithm's conformance to a set of properties, or its formal specification.

To illustrate further, consider how by running it through a wide set of test cases, we can obtain reasonable confidence that an algorithm behaves and outputs as expected. This is not exhaustive, however, as the input space is usually infinite. On the other hand, writing a formal specification for an algorithm describes its behavior (e.g. properties we expect to be held by its outputs) on arbitrary inputs. Thus, formal verification, which aims for a proof of conformance to the specification, verifies the algorithm's behavior on the whole input space.

Proofs for mathematical theorems are mostly written and checked by hand, leaving space for mistakes or disagreement. Advocates for formalization have thus developed interactive theorem provers, or proof assistants, allowing proofs to be written on and checked by machine. So long as there is consensus on the correctness of the proof assistant's (typically small) kernel, there is consensus on all results checked by the proof assistant.

The same situation applies to proofs concerning algorithms as well. Proposals of algorithmic solutions to classical problems are typically accompanied by hand-crafted proofs of the algorithm's termination and correctness. Formalizing an algorithm is thus the task of translating these English-language proofs to a form understood and checked by a machine proof assistant, which necessarily also involves a precise, machine-understandable specification of what we meant by termination, correctness, or any other property we deem interesting. This process, though quite arduous, achieves a far higher level of rigor and confidence, and typically helps discover edge cases and subtleties missed by handwritten proofs.

Isabelle/HOL (Higher-Order Logic) is a generic proof assistant equipped with a high level of automation, supporting inductive definitions, recursive functions, etc. For our purposes, after expressing an algorithm in purely functional, recursive form in SML-like syntax, and the properties we wish to prove about it in HOL, we already have the starting point we need for an Isabelle/HOL-based formalization. The proof is then written using Isar, Isabelle's formal proof language. In addition to being fully machine-checkable, Isar's syntax also helps expose the logical deduction / induction / case analysis-based proof steps in a human-readable form.

There is a reference for Isabelle/HOL syntax and built-in functions in the appendix, organized in order of appearance in this document.

The Gale-Shapley algorithm [1] is the classical solution to the stable matching problem. Given N men and N women, with each person's completely-ordered preferences for the N members of the opposite sex, one wishes to find a matching, a bijection between the men and the women, such that it is *stable*. An unstable pair in a matching is a pair $(\mathrm{man\ A}, \mathrm{woman\ B})$, each currently married to $(\mathrm{woman\ A}, \mathrm{man\ B})$ respectively in the matching, and not to each other, such that $\mathrm{man\ A}$ and $\mathrm{woman\ B}$ both prefer each other over their current partners. A matching is called *stable* if it contains no unstable pairs.

For example, consider men A, B, and women C, D. Given the following preferences, from most preferred to least preferred:

Men's preferences:		|	Women's preferences:	
A: C, D	B: D, C	|	C: A, B	D: B, A

$(A, D), (B, C)$ would be an unstable matching, because A and C both prefer each other over whom they are currently married to. The correct solution would be $(A, C), (B, D)$.

The Gale-Shapley algorithm, further explained in later sections, ***always finds a stable matching given any 2N individuals' preferences***. A formalization of the Gale-Shapley algorithm, the goal of my efforts, thus amounts to a machine-checked proof of this fact. The algorithm has a wide range of real-world applications, the most well-known being the task of assigning medical school graduates to their first hospital appointments [2].

My project thus involves:

- An implementation of the Gale-Shapley algorithm in Isabelle/HOL's flavor of ML, resembling the standard pseudocode for the algorithm as closely as possible

- A specification for the key properties of a stable matching algorithm, namely:
  - (For well-formatted inputs) At termination, the output is indeed a matching. In other words, all the men and women are married monogamously.
  - No unstable pair exists in the outputted matching.
- Proof of my implementation's conformance to the above 2 key properties.

This achieves the following objectives:

- A formalization of the Gale-Shapley algorithm in Isabelle/HOL, further affirming its correctness with very high confidence
- A discovery of subtleties and additional interesting properties of the Gale-Shapley algorithm
- An Isabelle/HOL theory reusable in larger projects dependent on the solution of the stable matching problem
- An implicit confirmation of the usability and feasibility of Isabelle/HOL, in its current state as of April 2021, for the purpose of formalizing a small but substantial real-world algorithm, even for complete beginners (see below).

Please note that, while the terminology used up till now explicitly assigns "men" and "women" to the 2 sets that we intend to match stably, we do understand that the Gale-Shapley algorithm applies to the abstract mathematical problem, and thus we could have used other examples such as applicants to vacancies or incoming requests to servers. Notably, my choice of gendered and heteronormative constructs associated with the "stable marriage problem", while intrinsic to the long-established literature and in this sense conventional, does sound archaic, and in some cases might be inappropriate or offensive, to the modern reader. Please accept this as a disclaimer and an apology in advance, and allow me to continue to use these conventional terms in the rest of this report, for the purpose of consistency with existing literature, and certainly without any intention to comment on the nature of real interpersonal relationships.

2 Starting Point

I had zero experience in formal verification before the commencement of this project at the end of Michaelmas term 2020. I had to learn Isar, as well as all the Isabelle/HOL proof approaches and tactics, from scratch.

I was not enrolled in L21: Interactive Formal Verification, the ACS module serving as a tutorial to Isabelle/HOL. The above learning process thus took place entirely outside of the 5 modules I took.

I had no prior knowledge of the stable matching problem nor the Gale-Shapley algorithm.

3 The Gale-Shapley Algorithm

3.1 Standard Pseudocode

```
1. Let all men be unmarried.
2. While there is still an unmarried man m:
    a) Let m propose to the woman w he most prefers and has not yet
       proposed to.
    b) If w is unmarried: let (m, w) be married.
    c) Else, w is married to m':
      i.    If w prefers m over m': divorce (m', w), such that m' is
            now unmarried. Then let (m, w) be married.
     ii.    Else: do nothing.
```

We can already notice a few nontrivial subtleties. Firstly, how do we know that the while loop eventually terminates, and everyone gets married? Secondly, where would stability come from? Thirdly, the pseudocode implicitly asserts that so long as a man is currently unmarried, he must still have someone left to propose to. Is this assertion correct, and if so, why is it the case? These are issues we must resolve in order to achieve a complete formalization.

But first, we must translate the above into a purely-functional and recursive form.

3.2 Implementation in Isabelle/HOL

The men and women are modelled using natural numbers, that is, 0-based indexes.

```
type_synonym person = "nat"
type_synonym man = "person"
type_synonym woman = "person"
```

Then, we need to specify the format of inputs to the function, the preferences:

```
type_synonym pref_matrix = "(person list) list"
```

Should we ever need a container `c` to hold information about each of the N members of one of the sexes, simply use a length-N list `c` such that `c!i` stores information about person `i`. Each man or woman's preferences is a list of persons, ordered from most preferred to least preferred. Hence our `pref_matrix` is a `(person list) list`.

For example, the matrix storing the men's preferences, `MPrefs` = $[[0,1],[1,0]]$, says that man 0 prefers woman 0 over woman 1, and man 1 prefers woman 1 over woman 0.

Again using lists, we build the 2 data structures needed for storing the state of our computation:

```
type_synonym matching = "(woman option) list"
```

`matching` is the type of both `engagements`, the variable keeping track of our current marriages, as well as the output of our Gale-Shapley function (which is simply the value of `engagements` at termination). It is a container holding information about the men: for man `m`, `engagements!m` is a `(woman option)` with the following value:

```
engagements!m = None    ⟺   m is unmarried
engagements!m = Some w ⟺ (m, w) are married to each other
```

Lastly, we need a data structure to help keep track of the proposals each man has already made. Recall that `MPrefs` is a `pref_matrix` storing a list of women for each man. So, for each man, we use an index pointing at the woman he should next propose to. This would start at 0 and get incremented each time a proposal is made. Hence our data structure, `prop_idxs::int list`. Referring to the pseudocode, we thus have `w = (MPrefs!m)!(prop_idxs!m)`.

Additionally, let us create a helper function that finds the index of the first instance of an element in a list:

```
fun find_idx::"'a ⇒ 'a list ⇒ nat option" where
"find_idx _ [] = None" |
"find_idx term (x # xs) = (if term = x then Some 0 else
          (case find_idx term xs of None ⇒ None |
```

```
                              Some idx ⇒ Some (Suc idx)))"
```

And wrap it and alias it for clarity, creating the following additional functions:

```
findFreeMan engagements = find_idx None engagements
findFiance engagements w = find_idx (Some w) engagements
```

Furthermore, using `find_idx`, we can define the `prefers` relation given a person `p`, and the corresponding `pref_matrix` `MPrefs/WPrefs` storing his/her preferences, for 2 members p1 p2 of the opposite sex, as follows:

```
fun prefers::"person ⇒ pref_matrix ⇒ person ⇒ person ⇒ bool" where
"prefers p PPrefs p1 p2 = (
 case find_idx p1 (PPrefs!p) of None ⇒ False | Some idx_1 ⇒ (
 case find_idx p2 (PPrefs!p) of None ⇒ False | Some idx_2 ⇒
 idx_1 < idx_2))"
```

Note that, for well-formatted inputs, the `None` cases will never arise. As an example, `prefers w WPrefs m1 m2` says that `w` prefers `m1` over `m2`.

And so we arrive at our implementation, where the iterative while-loop in the pseudocode is converted into tail-recursive functional form:

```
function Gale_Shapley'::
"nat ⇒ pref_matrix ⇒ pref_matrix ⇒ matching ⇒ nat list ⇒ matching"
where
"Gale_Shapley' N MPrefs WPrefs engagements prop_idxs =
(if length engagements ≠ length prop_idxs then engagements else
(if sum_list prop_idxs ≥ N * N then engagements else

(case findFreeMan engagements of None ⇒ engagements |

 Some m ⇒ (let w = MPrefs!m!(prop_idxs!m);
   next_prop_idxs = prop_idxs[m:=Suc (prop_idxs!m)] in (
   case findFiance engagements w of
     None ⇒ Gale_Shapley' N MPrefs WPrefs
            (engagements[m:=Some w]) next_prop_idxs
   | Some m' ⇒ (if prefers w WPrefs m m'
```

```
        then Gale_Shapley' N MPrefs WPrefs
             (engagements[m:=Some w, m':=None]) next_prop_idxs
        else Gale_Shapley' N MPrefs WPrefs
              engagements next_prop_idxs))))))"

fun Gale_Shapley::"pref_matrix ⇒ pref_matrix ⇒ matching" where
"Gale_Shapley MPrefs WPrefs = (let N = length MPrefs in
 Gale_Shapley' N MPrefs WPrefs (replicate N None) (replicate N 0))"
```

The main difference between the pseudocode form and the Isabelle/HOL implementation is the first 2 lines of the definition of `Gale_Shapley'`, corresponding to 2 additional early-exit conditions. They check for either a mismatch in the length of the 2 lists `engagements` and `prop_idxs`, or the sum of the proposal indexes of all the men reaching or exceeding $N \times N$, in addition to the standard termination condition, having no unmarried men left, on the 3rd line.

These 2 conditions are added for two main reasons. The first reason is for termination certification (see 3.3). The second reason is that `Gale_Shapley'` should be able to detect cases where input parameters are clearly malformed, and exit without performing any computation by directly returning `engagements`. Indeed, since the length of both `engagements` and `prop_idxs` is the number of men, N, they should be equal. Detection of a mismatch demands early-exit. On the other hand, let us assume for the sake of argument that the inputs are not malformed, which includes assuming that the length of `prop_idxs` is N, and that each entry in `prop_idxs` is well-behaved and thus $\leq N$ (equality suggests the corresponding man has finished all proposals; $< N$ is the case where there are still proposals left to make, and the index is within bounds of the man's preference list; $> N$ is out-of-bounds and ruled out). With these assumptions, the sum across `prop_idxs` being $\geq N \times N$ corresponds to two possibilities. In the equality case, given our assumptions, we can deduce that every proposal index is $= N$, i.e. all men have finished all proposals, and thus we can no longer make any progress and should terminate. Note that this is an additional termination condition distinct from no-free-men-left of the pseudocode! In the strict case, we can deduce that there must exist at least one proposal index $> N$, i.e. out-of-bounds, and thus the input is malformed, and we should early-exit. Note that checking the sum of `prop_idxs` is still far weaker than a proper bounds check on all indexes: for N=2, [3,0] has sum 3 < $2 \times 2 = 4$, but 3 is clearly out-of-bounds.

Given these justifications for the 2 additional exit conditions, we still cannot avoid the important requirement that the function we actually prove properties about, `Gale_Shapley'`, deviates as little as possible from the standard form of the Gale-Shapley algorithm itself, because only then can we suggest that properties proven about `Gale_Shapley'` are also the properties of the Gale-Shapley algorithm. Unfortunately, the alterations made just by the 2 additional lines are quite substantial. The pseudocode does not check for the status of the proposals already made by any man (corresponding to `prop_idxs`); it is confident that any free man would still have someone to propose to. Though we should thus attempt to avoid such checks in our implementation as well, checking `sum_list prop_idxs` for early-exit is one such check. For termination, the pseudocode only checks for the existence of an unmarried man, while we, on top of `findFreeMan engagements` = None, additionally check for the well-formatted case of `sum_list prop_idxs` $= N \times N$, i.e. all men finishing proposing, as discussed earlier. This is all unideal.

These concerns are resolved by the fact that the two early-exit conditions are *extraneous*, in the sense that they will never be true at any point in the execution of `Gale_Shapley'` on well-formatted inputs. Being confident that what they check for will never be true anyway, we can say that their addition to the implementation will not actually result in deviations from the algorithm's standard form during execution, and is thus permissible. In other words, extraneous conditions are permissible because they only cause deviations in form but not in substance – i.e. during execution or in the trace.

From what we know so far however, only the first condition is clearly extraneous. For well-formatted inputs, i.e. calling `Gale_Shapley'` via `Gale_Shapley`, the length of the 2 state variables will both be N and stay that way throughout execution. For all we know, the second condition might actually be reached by well-formatted inputs in the $= N \times N$ case, when all men are done with all proposals. (This says nothing about whether the men are all married at this point!) Also, how do we know that a man whose proposal index is $= N$ would not be chosen again as a free man, where `MPrefs!m!(prop_idxs!m)` would go out-of-bounds, as the maximal valid index is $N - 1$?

We will eventually state and prove a theorem firmly establishing that the 2nd extraneous condition is indeed extraneous – that it will in fact never be true for well-formatted inputs (see 6.8). We will also show that proposal indexes are always well-behaved such that `MPrefs!m!(prop_idxs!m)` never fails.

3.3 Certifying Termination

An important proof obligation that must be fulfilled before Isabelle accepts a recursive function definition as valid is to show that it terminates on all inputs. This is not straightforward, as we are not able at the moment to show that *there must exist some iteration after which all men are married*.

Thankfully, our 2 extraneous early-exit conditions enable us to trivially certify the termination of `Gale_Shapley'` on the whole input space. In particular, notice that `sum_list prop_idxs` always increases by 1 after each iteration, and that we have hard-coded termination if it reaches or exceeds $N \times N$. Of course, being able to trivially certify the termination of `Gale_Shapley'` this way does not allow us to sidestep the real difficulty in proving the termination of the Gale-Shapley algorithm, as we will eventually need to prove that `Gale_Shapley` given well-formatted inputs `(MPrefs, WPrefs)` always returns a `matching ≡ (woman option) list` containing no None's. Nevertheless, this form of termination certification does inform us that Gale-Shapley takes $O(N^2)$ steps to complete.

In any case, the syntax for guiding Isabelle to make use of `sum_list prop_idxs` to certify termination for `Gale_Shapley'` is as follows:

```
function Gale_Shapley'::"…" where "Gale_Shapley' … = …"
  by pat_completeness auto
termination
  apply (relation "measure (λ(N, _, _, _, prop_idxs).
                               N * N - sum_list prop_idxs)")
  by (auto intro:termination_aid)
```

where `termination_aid` is the following lemma:

```
lemma termination_aid:
assumes "length engagements = length prop_idxs"
    and "findFreeMan engagements = Some m"
    and "next_prop_idxs = prop_idxs[m:=Suc(prop_idxs!m)]"
    and "sum_list prop_idxs < N * N"
shows "N * N - sum_list next_prop_idxs < N * N – sum_list prop_idxs"
```

the core of the proof of which is using the following lemma to argue that `m < length engagements`, which, given the assumptions, amounts to saying that `m` is also within the bounds of `prop_idxs`:

```
lemma find_idx_bound:
"find_idx term xs = Some idx ⟹ idx < length xs"
```

which is itself proven by structural induction on lists on `xs`.

The core element of all this is of course the `measure` keyword, which allows us to introduce a mapping from the arguments of the function we're trying to show termination for to a natural number that monotonically decreases across recursive calls, and implies termination when it reaches zero.

3.4 Specification in Isabelle/HOL

Having implemented `Gale_Shapley'` and fulfilled the Isabelle obligation to show termination on all of `(nat × pref_matrix × pref_matrix × matching × nat list)`, I am ready to state the end-goal of the formalization, the formal specification.

`Permutation.thy` in the Isabelle/HOL library provides an inductive definition for the relation between two lists differing only in order and not content, `<~~>`. Duplicates are allowed, e.g. `[1,1,2] <~~> [1,2,1]`, but I will not need this provision. `Permutation.thy` also states in `mset_eq_perm` that `<~~>` is equivalent to multiset equality, allowing us to compute `<~~>`:

```
theorem mset_eq_perm: "mset xs = mset ys ⟷ xs <~~> ys"
```

```
fun is_perm::"'a list ⇒ 'a list ⇒ bool" where
"is_perm A B = (mset A = mset B)"
```

```
lemma is_perm:"is_perm A B ⟷ A <~~> B" using mset_eq_perm by auto
```

Having the above, I can state the function computing whether or not an input `pref_matrix` is well-formatted with respect to N:

```
fun is_valid_pref_matrix::"nat ⇒ pref_matrix ⇒ bool" where
"is_valid_pref_matrix N PPrefs = (length PPrefs = N ∧
                                  Ball (set PPrefs) (is_perm [0 ..< N]))"
```

`is_valid_pref_matrix N PPrefs` says that `PPrefs` is a list of N length-N lists, where each is a permutation of the list `[0, 1, …, N-1]`, i.e. some ordering of all N members of the opposite sex.

I can now state the specification. Firstly, that the output is indeed a matching:

```
theorem is_matching:
  assumes "is_valid_pref_matrix N MPrefs" and "N ≥ 2"
  shows "(Gale_Shapley MPrefs WPrefs) <~~> map Some [0..<N]"
```

The output of `Gale_Shapley` is the value of `engagements` at termination. Recall that each entry of the list `engagements` is either None or `Some w`. To assert that no entry is None is to assert that all N men are married. We additionally need to make sure the marriages form a bijection, i.e. monogamy. Of course, the w's should also be valid indices for the women. The easiest way to condense all this into one statement is to assert that the output is a permutation of `[Some 0, Some 1, …, Some N-1]`, as done above.

I will eventually need to exclude trivial cases $N = 0, 1$. Additionally, I will find that this result does *not* depend on the format of `WPrefs`.

Secondly, that the matching outputted is stable:

```
abbreviation unstable where
"unstable MPrefs WPrefs engagements ≡ ∃m1 m2 w1 w2.
   m1 < length engagements  ∧  m2 < length engagements
 ∧ engagements!m1 = Some w1 ∧  engagements!m2 = Some w2
 ∧ prefers w1 WPrefs m2 m1  ∧  prefers m2 MPrefs w1 w2"

theorem stable:
"⟦is_valid_pref_matrix N MPrefs; is_valid_pref_matrix N WPrefs⟧
   ⟹ ¬ unstable MPrefs WPrefs (Gale_Shapley MPrefs WPrefs)"
```

Having the starting point and the end-goal, I can now start working towards a proof of the 2 theorems above – the main bulk of the formalization effort.

4 Computational Induction

For the uninitiated, here is a brief, non-rigorous overview of computational induction, a powerful way to prove properties of recursive total functions. It is a part of the Isabelle `function` package. A prerequisite for using the automatically generated computational induction proof rule is the aforementioned termination certification.

For a recursive function $f\ args$ with the following form:

$$f\ args = \text{Base}\ args \Rightarrow output \mid \text{Rec}\ args \Rightarrow \ldots (f\ args_1) \ldots$$

where Base and Rec are predicates of $args$, $output$ does not contain f, and of course $\forall args.\ \text{Base}\ args \lor \text{Rec}\ args$, to prove the statement $\text{P}\ args \Longrightarrow \text{Q}\ args$ by computational induction, we are allowed to use the induction hypothesis, $[\![\text{Rec}\ args; \text{P}\ args_1]\!] \Longrightarrow \text{Q}\ args_1$. This leads to the following steps:

1. Assume $\text{P}\ args$. Then:
2. Case Base
   a) Assume $\text{Base}\ args$. Show $\text{Q}\ args$.
3. Case Rec
   a) Assume $\text{Rec}\ args$. Show $\text{P}\ args_1$.
   b) Further assume $\text{Q}\ args_1$. Show $\text{Q}\ args$.

The procedure for functions with more than one base or recursive case is similar, where we would have one induction hypothesis per recursive case.

See 5.1.1 for an annotated example of computational induction in Isar.

5 GS'_arg_seq: the Argument Sequence

`Gale_Shapley'` is not expressive enough for stating crucial lemmas.

Consider `GS'_arg_seq` (argument sequence of `Gale_Shapley'`), where instead of being tail-recursive and returning the terminal state of `engagements` only, we return the whole history of the state of `(engagements, prop_idxs)`:

```
function GS'_arg_seq::
"nat ⇒ pref_matrix ⇒ pref_matrix ⇒ matching ⇒ nat list
 ⇒ (matching × nat list) list" where
"GS'_arg_seq N MPrefs WPrefs engagements prop_idxs =
(if length engagements ≠ length prop_idxs then
[(engagements, prop_idxs)] else
(if sum_list prop_idxs ≥ N * N then
[(engagements, prop_idxs)] else
(case findFreeMan engagements of None ⇒
[(engagements, prop_idxs)] |

 Some m ⇒ (let w = MPrefs!m!(prop_idxs!m);
   next_prop_idxs = prop_idxs[m:=Suc (prop_idxs!m)] in (
   case findFiance engagements w of
     None ⇒ (engagements, prop_idxs) # (GS'_arg_seq N MPrefs WPrefs
             (engagements[m:=Some w]) next_prop_idxs)
   | Some m' ⇒ (if prefers w WPrefs m m'
        then (engagements, prop_idxs) # (GS'_arg_seq N MPrefs WPrefs
             (engagements[m:=Some w, m':=None]) next_prop_idxs)
        else (engagements, prop_idxs) # (GS'_arg_seq N MPrefs WPrefs
              engagements next_prop_idxs)))))))"
  by pat_completeness auto
termination
  apply (relation "measure (λ(N, _, _, _, prop_idxs).
                              N * N - sum_list prop_idxs)")
  by (auto intro:termination_aid)
```

The output of (`GS'_arg_seq N MPrefs WPrefs` *`init_args`*) is a full trace of the computation ran by "`result = Gale_Shapley' N MPrefs WPrefs` *`init_args`*", [*init_args*, args1, args2, …, (result, _)]. The increased expressiveness of GS'_arg_seq is clear by example from the statement of the following lemma, impossible with just Gale_Shapley'.

```
lemma GS'_arg_seq_prev_prop_idx_same_or_1_less:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs"
      and "Suc i < length seq" and "seq!Suc i = (X, Y)"
      and "seq!i = (X_prev, Y_prev)" and "k = Suc k_1" and "Y!m = k"
    shows "Y_prev!m = k ∨
           Y_prev!m = k_1 ∧ findFreeMan X_prev = Some m"
```

5.1 Theorem: GS'_arg_seq_computes_GS'

Of course, as our specifications concern Gale_Shapley' and Gale_Shapley only, we need the following to link GS'_arg_seq back to Gale_Shapley'.

```
theorem GS'_arg_seq_computes_GS':
  assumes "seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs"
      and "length seq = Suc i" and "seq!i = (X, Y)"
    shows "X = Gale_Shapley' N MPrefs WPrefs engagements prop_idxs"
```

5.1.1 Lemma: GS'_arg_seq_last_eq_terminal

To prove GS'_arg_seq_computes_GS', I first prove the lemma GS'_arg_seq_last_eq_terminal using computational induction:

```
abbreviation is_terminal where
"is_terminal N engagements prop_idxs ≡
 length engagements ≠ length prop_idxs ∨
 sum_list prop_idxs ≥ N * N ∨
 findFreeMan engagements = None"
```

The above abbreviation concatenates the 3 base cases for readability.

```
lemma GS'_arg_seq_last_eq_terminal:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
```

```
 i < length seq; seq!i = (X, Y)⟧
   ⟹ is_terminal N X Y ⟷ length seq = Suc i"
proof
  show "⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
         i < length seq; seq!i = (X, Y);
         is_terminal N X Y⟧ ⟹ length seq = Suc i"
  proof (induction N MPrefs WPrefs engagements prop_idxs
         arbitrary: seq i rule:GS'_arg_seq.induct)
    …
    …
  qed
next
  show "⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
         i < length seq; seq!i = (X, Y);
         length seq = Suc i⟧ ⟹ is_terminal N X Y"
  proof (induction N MPrefs WPrefs engagements prop_idxs
         arbitrary: seq i rule: GS'_arg_seq.induct)
    case (1 N MPrefs WPrefs engagements prop_idxs)
    show ?case
    proof (cases i)
      case 0
      with "1.prems"(4) have "¬ length seq > 1" by …
      with "1.prems"(1) GS'_arg_seq_length_gr_1
      have "is_terminal N engagements prop_idxs" by …
      with "1.prems"(1,3) 0 show ?thesis by …
    next
      case (Suc i_1)
      with "1.prems"(4) have "length seq > 1" by …
```

Proof similar to the other direction below

fact 0: `i = 0`

.prems(4) is the 4th premise, i.e. `length seq = Suc i`, which with case `i = 0` gives `length seq = 1`

`GS'_arg_seq_length_gr_1`: the fact that
`length (GS'_arg_seq N MPrefs WPrefs engagements prop_idxs) > 1 ⟷ ¬ is_terminal N engagements prop_idxs`

`?thesis`: `is_terminal N X Y`
…: uses fact `GS'_arg_seq_0: (GS'_arg_seq N MPrefs WPrefs X' Y')!0 = (X', Y')`

fact Suc: `i = Suc i_1`

```
with "1.prems"(1) GS'_arg_seq_length_gr_1
have non_terminal:"¬ is_terminal N engagements prop_idxs" by …
then obtain m where m:"findFreeMan engagements = Some m" by …
let ?w = "MPrefs!m!(prop_idxs!m)"
show ?thesis
proof (cases "findFiance engagements ?w")
  case None
  with "1.IH"(1)[OF _ _ m refl refl None refl
                  i_1_bound[OF Suc _ "1.prems"(2)]
                  tl_i_1_eq[OF Suc _ "1.prems"(3)]]
        Suc "1.prems"(4,1) non_terminal m
```

```
"1.IH"(1): ⟦¬ length engagements ≠ length prop_idxs; ¬ sum_list prop_idxs ≥ N * N;
 findFreeMan engagements = Some ?m; ?w = MPrefs!?m!(prop_idxs!?m);
 ?next_prop_idxs = prop_idxs[?m:=Suc(prop_idxs!?m)]; findFiance engagements ?w = None;
 ?seq = GS'_arg_seq N MPrefs WPrefs engagements[?m:=Some ?w] ?next_prop_idxs;
 ?i < length ?seq; ?seq!?i = (X, Y); length ?seq = Suc ?i⟧ ⟹ is_terminal N X Y
```

This is the automatically generated induction hypothesis for the first recursive branch in the computational induction. The first 6 premises correspond to Rec, the predicate for the branch we've entered. The last 4 premises correspond to $\mathrm{P}\ args_1$, the 4 premises of the lemma we are trying to prove, but modified to correspond to the arguments passed to the recursive call.

`refl`: `?x = ?x`.
`tl_i_1_eq:"⟦i = Suc i_1; seq = x#xs; (seq!i) = X⟧ ⟹ (xs!i_1) = X"`
`i_1_bound:"⟦i = Suc i_1; seq = x#xs; i < length seq⟧ ⟹ i_1 < length xs"`

The above 2 lemmas are used to perform the $\mathrm{P}\ args \Longrightarrow \mathrm{P}\ args_1$ derivation. They are concerned with the situation `(x # xs)!(Suc i_1) = seq!i = xs!i_1`. As it happens, $\mathrm{Q}\ args$ and $\mathrm{Q}\ args_1$ are both `is_terminal N X Y`, so the $\mathrm{Q}\ args_1 \Longrightarrow \mathrm{Q}\ args$ step is not needed.

Let `?seq_tl = (GS'_arg_seq N MPrefs WPrefs engagements[m:=Some (MPrefs!m!(prop_idxs!m))] prop_idxs[m:=Suc(prop_idxs!m)])`. Then, after unification, `"1.IH"(1)[OF …]` becomes:

```
⟦¬ length engagements ≠ length prop_idxs; ¬ sum_list prop_idxs ≥ N * N;
 seq = ?x # ?seq_tl; length ?seq_tl = Suc i_1⟧ ⟹ is_terminal N X Y.
```

The first 2 premises are fulfilled by `non_terminal`. By examining .prems(1), which states that `seq = GS'_arg_seq` $args$, and the branch of `GS'_arg_seq` we've entered into via `non_terminal`, m, None, we are able to derive `seq = (engagements, prop_idxs) # (?seq_tl ≡ GS'_arg_seq` $args_1$`)`, which fulfills the 3rd premise. Additionally examining .prems(4) and `Suc` fulfills the last premise.

```
          show ?thesis by (simp add:Let_def)
        next
          case (Some m')
          show ?thesis
          proof (cases "prefers ?w WPrefs m m'")
            case True
            with "1.IH"(2)[OF _ _ m refl refl Some True refl
                             i_1_bound[OF Suc _ "1.prems"(2)]
                             tl_i_1_eq[OF Suc _ "1.prems"(3)]]
                 Suc "1.prems"(4,1) non_terminal m Some
            show ?thesis by (simp add:Let_def)
          next
            case False
            with "1.IH"(3)[OF _ _ m refl refl Some this refl
                             i_1_bound[OF Suc _ "1.prems"(2)]
                             tl_i_1_eq[OF Suc _ "1.prems"(3)]]
                 Suc "1.prems"(4,1) non_terminal m Some
            show ?thesis by (simp add:Let_def)
          qed
        qed
      qed
    qed
qed
```

The Isar scripts above are to prove the fact that, in any argument sequence outputted by `GS'_arg_seq`, *only* the last pair `is_terminal`. `is_terminal` is critical because of the simple fact that:

```
is_terminal N engagements prop_idxs ⟹
Gale_Shapley' N MPrefs WPrefs engagements prop_idxs = engagements,
```

and we are aiming for a proof of `GS'_arg_seq_computes_GS'`. In fact, the *only* part corresponds to the direction omitted as "…" in the scripts above.

The proof scripts showcase computational induction. First, a case analysis on the index `i` separates the base and recursive branches: if `i` is 0, since `i` also points at the last element in the sequence, the sequence must have unit length, which suggests that its *init_args* `is_terminal` via `GS'_arg_seq_length_gr_1`. In other words, we are in the base case. *init_args* is of course also what `i`, X, Y point to, so we have `is_terminal N X Y`, and are done.

If `i` is equal to `Suc i_1` (`i` is at least 1), then `length seq` is at least 2. Again by `GS'_arg_seq_length_gr_1` we must have `¬ is_terminal` *init_args*, so we are in one of the three recursive branches of `GS'`. Further branching on the values of `findFiance` and `prefers` determines which of the three branches we are in, after which we have fulfilled the Rec part (the branching predicate) of the induction hypothesis. We know from `length seq = Suc i` that `i` points to the last element in `seq`, which means it of course points to the last element in each of the 3 `seq_tl`'s also (the tail of `seq`). The induction hypotheses essentially say that the last element of each `seq_tl` `is_terminal` – then of course so is the last element of `seq`, and so we are done. Concretely, from P *args*
`⟦seq = GS'_arg_seq` *args*; `i < length seq`; `seq!i = (X, Y)`;
`length seq = Suc i⟧`,
given `i = Suc i_1` and one of the 3 Rec branching predicates to know which $args_1$ or `seq_tl` we're talking about, we can easily derive:
`⟦seq_tl = GS'_arg_seq` $args_1$; `i_1 < length seq_tl`; `seq_tl!i_1 = (X, Y)`;
`length seq_tl = Suc i_1⟧`,
thus fulfilling the P $args_1$ part of the induction hypothesis. Of course, we also need (`seq =` *init_args* `# seq_tl`) in the process, which is by Rec and definition of `GS'_arg_seq`.

5.1.2 Lemma: GS'_arg_seq_same_endpoint

The following additional lemma will lead to `GS'_arg_seq_computes_GS'`:

```
lemma GS'_arg_seq_same_endpoint:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
  i < length seq; seq!i = (X, Y)⟧
   ⟹ Gale_Shapley' N MPrefs WPrefs X Y
    = Gale_Shapley' N MPrefs WPrefs engagements prop_idxs"
proof (induction N MPrefs WPrefs engagements prop_idxs
       arbitrary:seq i rule:GS'_arg_seq.induct)
```

```
  …
qed
```

The lemma above states that starting the `Gale_Shapley'` computation from anywhere in the argument sequence will end up at the same result – that of the one started at $init\_args$. Walking down the same path will take you to the same endpoint no matter where along the path you started. The proof is by computational induction, where the induction hypothesis will give the result `GS' X Y = GS'` $args_1$, which is also equal to `GS'` $args$ since `GS'` is tail-recursive, completing the proof.

5.1.3 Proof

With the 2 lemmas, proving `GS'_arg_seq_computes_GS'` is straightforward. We aim to show that `X_last`, the first element of the last pair in the sequence `GS'_arg_seq` $args$, equals `Gale_Shapley'` $args$. We have from `last_eq_terminal` that `(X_last, Y_last) is_terminal`. This means `X_last = Gale_Shapley' N _ _ X_last Y_last`. From `same_endpoint`, this is also equal to `Gale_Shapley'` $args$, as required.

5.2 Lemma: GS'_arg_seq_step_1,2,3

Computational induction as demonstrated above is certainly powerful, but it can still be a limiting factor. The induction hypotheses enable assumptions about the sequence-tails, corresponding to `GS'_arg_seq` $args_1$, from which we derive facts about the whole sequence `GS'_arg_seq` $args$; however, this inductive step only allows us to discuss the transition between the first and second elements of the argument sequence, and not elsewhere.

Recall that the argument sequence is a trace of the computation done by `Gale_Shapley'`. Intuitively, we should be able to inspect any point, say midway in the argument sequence, and be able to deduce the subsequent argument-pair, without having to consult the rest of the sequence (e.g. the $init\_args$). Indeed, consider the following lemmas:

```
lemma GS'_arg_seq_step_1:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
```

```
  Suc i < length seq; seq!i = (X, Y); findFreeMan X = Some m;
  w = MPrefs!m!(Y!m); findFiance X w = None⟧
   ⟹ seq!Suc i = (X[m:=Some w], Y[m:=Suc(Y!m)])"
proof (induction N MPrefs WPrefs engagements prop_idxs
       arbitrary:seq i rule:GS'_arg_seq.induct)
  …
qed

lemma GS'_arg_seq_step_2:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
  Suc i < length seq; seq!i = (X, Y); findFreeMan X = Some m;
  w = MPrefs!m!(Y!m); findFiance X w = Some m';
  prefers w WPrefs m m'⟧
   ⟹ seq!Suc i = (X[m:=Some w, m':=None], Y[m:=Suc(Y!m)])"
proof (induction N MPrefs WPrefs engagements prop_idxs
       arbitrary:seq i rule:GS'_arg_seq.induct)
  …
qed

lemma GS'_arg_seq_step_3:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
  Suc i < length seq; seq!i = (X, Y); findFreeMan X = Some m;
  w = MPrefs!m!(Y!m); findFiance X w = Some m';
  ¬prefers w WPrefs m m'⟧
   ⟹ seq!Suc i = (X, Y[m:=Suc(Y!m)])"
proof (induction N MPrefs WPrefs engagements prop_idxs
       arbitrary:seq i rule:GS'_arg_seq.induct)
  …
qed
```

The step lemmas take a single step forward from `(X, Y)`, an arbitrary point in a given argument sequence. The only constraint is that they are not at the very end, where no step can be taken. Note that the assumptions do not include `¬is_terminal N engagements prop_idxs`, nor `¬is_terminal N X Y`. These are derivable from `Suc i < length seq`, which implies both that the sequence has length at least 2, and that `i X Y` do not point to the end. Then, `length_gr_1` and `last_eq_terminal` lead to the `non_terminal` branching predicates.

The step lemmas are proven nearly identically using computational induction. Case analysis on `i` is used as usual. The 0 case, i.e. taking the 1^{st}-2^{nd} step, follows directly from `GS'_arg_seq_0` (see annotations in 5.1.1). The `Suc` case takes a step that lies fully within the `seq_tl` (i.e. anywhere but the head – first element – of `seq`), allowing direct use of the induction hypothesis.

The above lemmas will prove useful in later proofs; in particular, they enable natural-number induction directly on `i` given any argument sequence.

6 is_matching

The `is_matching` result in the first part of the formal specification, that our implementation `Gale_Shapley` always outputs a matching (i.e. a bijection, or a full set of N monogamous marriages), `"(Gale_Shapley MPrefs WPrefs) <~~> map Some [0..<N]"`, is a giant leap. The proof scripts that directly contributed to this result made up 80% of the whole Isabelle theory file – that is to say, only 20% is relevant only to the second part of the specification, `stable`. Stability seems like a more substantial result at first glance. It is, indeed, but it turns out that many of the important lemmas leading to stability are needed for the `is_matching` result as well.

In any case, I first broke `is_matching` down into 3 parts.

6.1 Roadmap: is_matching_intro

```
abbreviation is_distinct where
"is_distinct engagements ≡
 ∀m1 < length engagements. ∀m2 < length engagements. m1 ≠ m2 ⟶
  engagements!m1 = None ∨ engagements!m1 ≠ engagements!m2"
```

This says that in a `(woman option) list`, an entry is either None or unique. In other words, no woman is married to more than one man simultaneously. Note that, additionally assuming that None's do not exist in the list implies that all entries in the list are distinct from each other. The Isabelle/HOL built-in predicate `distinct` holds for all lists `xs` where all elements are unique within the list. `is_distinct` is a modified variant to accommodate duplicate None's.

```
abbreviation is_bounded where
"is_bounded engagements ≡ ∀m < length engagements.
 engagements!m ≠ None ⟶ the (engagements!m) < length engagements"
```

Suppose the length of `engagements` is N. This says that all entries are either None or `Some w` where `w < N`. (All women are from `0` to `N-1`.)

```
lemma is_matching_intro:
```

```
  assumes noFree:"∀m < length engagements. engagements!m ≠ None"
      and "is_distinct engagements" and "is_bounded engagements"
    shows "engagements <~~> map Some [0 ..< length engagements]"
proof -
  …
qed
```

`is_matching_intro` introduces the 3 sufficient conditions for `is_matching`: `is_distinct`, `is_bounded`, and `noFree` (i.e. None `∉ set engagements`).

Core to its proof is the fact `card_subset_eq`, which states that for finite sets, $[\![A \subseteq B; \mathrm{card}\, A = \mathrm{card}\, B]\!] \Longrightarrow A = B$. Moreover, by combining `mset_eq_perm` (see 3.4) and `set_eq_iff_mset_eq_distinct`, we have that, for two lists, $[\![\mathrm{distinct}\, xs;\ \mathrm{distinct}\, ys; \mathrm{set}\, xs = \mathrm{set}\, ys]\!] \Longrightarrow xs < \sim\sim > ys$.

We will later see that, for `(Gale_Shapley MPrefs WPrefs)`, proving `is_distinct` is trivial (checking preservation of an invariant will suffice), but both `is_bounded` and `noFree` require heavy groundwork.

6.2 Lemma: GS'_arg_seq_all_distinct

```
lemma GS'_arg_seq_all_distinct:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
  is_distinct engagements; i < length seq; seq!i = (X, Y)⟧
   ⟹ is_distinct X"
proof (induction i arbitrary: X Y)
  …
qed
```

Though induction on `i` is used, computational induction is equally viable for this lemma. Induction on `i` proceeds in the standard way: for the base case `i = 0`, note that `X = engagements`, and we are done. The inductive step assumes the induction hypothesis:

```
⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
 is_distinct engagements; i < length seq; seq!i = (?X, ?Y)⟧
   ⟹ is_distinct ?X,
```

as well as the premises:

```
seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
is_distinct engagements; Suc i < length seq; seq!Suc i = (X, Y),
```

and we need to show `is_distinct X`. First name `seq!i = (X_prev, Y_prev)`. Then the induction hypothesis easily gives `is_distinct X_prev`. From this, the 3 step lemmas help derive X in terms of `X_prev` depending on the branch, e.g. `X = X_prev[m:=Some w]`. Lastly, performing the `is_distinct X_prev ⟹ is_distinct X` invariant-preservation check finishes the proof. This check, as done in Isar, is far from straightforward, but still not worth reporting in detail here.

It is worth commenting that if our goal was to solely prove `is_distinct (Gale_Shapley MPrefs WPrefs)`, `GS'_arg_seq` would not even be needed. By using computational induction via `rule: Gale_Shapley'.induct`, we can directly show the statement,

```
is_distinct engagements ⟹
is_distinct (Gale_Shapley' N MPrefs WPrefs engagements prop_idxs).
```

The remaining step, `is_distinct (replicate N None)`, is trivial. But this is a very weak result; a nearly identical proof but with alternative formulations will give a much stronger result, as shown above: that `is_distinct` holds throughout the argument sequence, not only at the end. This stronger version will prove useful.

6.3 Difficulty of noFree and is_bounded

Before I describe the groundwork that took place in order to prove the `noFree` and `is_bounded` results for `(Gale_Shapley MPrefs WPrefs)`, we should develop an intuition for why directly attacking the proofs is infeasible.

Consider `is_bounded`, and imagine formulating and proving it in a manner similar to the `GS'_arg_seq_all_distinct` result. If we assume as a premise that the initial `engagements is_bounded` (`is_bounded (replicate N None)` is indeed trivially true), both computational induction and natural number induction would require an invariant check similar to, as an example, `is_bounded X_prev`

⟹ is_bounded X_prev[m:=Some w]. This would succeed should we be able to show w < N, i.e. (MPrefs!m!(Y_prev!m)) < N. One might say this is easy by is_valid_pref_matrix N MPrefs, arguing thus that every entry in the 2D matrix is less than N. Indeed, this is what I will eventually do; however, the derivation has the prerequisite of checking that Y_prev!m ≡ prop_idxs!m ≡ the proposal index of an arbitrary, free man at an arbitrary time is always within-bounds of (MPrefs!m), i.e. strictly less than its length, N. This is certainly non-trivial! In the general case, it isn't even true: consider engagements=[None, Some 0] and prop_idxs=[2, 0]. From all we know so far, this could very well be a point in a valid trace, where the free man selected, 0, has already run out of proposals to make ($2 = N$, $2 \not< N$), such that prop_idxs!0 is out of bounds of (MPrefs!0). So just to prove is_bounded, we need to first answer the 3rd question raised in 3.1: how can we show that all free men must have proposals left to make? This is equivalent to showing the contrapositive that any man having already proposed to all the women must be married (and stay married). This is nontrivial. In fact, I will discover and show an even stronger result that, as soon as any man finishes all his proposals, that man, and every other man as well, must all be married. This fact, alongside a bit of arithmetic, also leads to the noFree result.

On the other hand, from what we know so far, noFree is certainly not a preserved invariant but rather an emerging property (in fact, since noFree implies is_terminal, observing GS'_arg_seq_last_eq_terminal, assuming noFree does become true at some point, it will do so precisely and only at the very end.) noFree is certainly false for (replicate N None). Hence, the invariant-preservation strategy breaks down.

6.4 Lemma: GS'_arg_seq_all_w_marry_better

That all women marry better, i.e. all women are continually, monotonically made (non-strictly) better off, is the single most important lemma formulated and proven in this formalization. Indeed, this is an intrinsic, characteristic property of the Gale-Shapley algorithm itself, and certainly a core, intentional decision made in its original design.

Let us first take another look at the pseudocode. Consider an arbitrary woman, **w0**. In each iteration of Gale-Shapley, observe that:

```
If w₀ = w, the woman the free man m proposes to:
    If w₀ was unmarried, she is married to m and better off.

    If w₀ was married to m', she either becomes married to m whom she
    prefers over m', and is better off; or she stays married to m'
    and is as well off as before.
If w₀ ≠ w, w₀'s situation stays unaffected.
```

One of the chief consequences of this observation is that, if at any point an arbitrary woman **w** becomes/is married, she will stay married thereafter (though not necessarily to the same man).

To collect and formulate the above observations, consider the following:

```
fun married_better::
"woman ⇒ pref_matrix ⇒ matching ⇒ matching ⇒ bool" where
"married_better w WPrefs eng_1 eng_2 =
(case findFiance eng_1 w of None ⇒ True | Some m_1 ⇒ (
   case findFiance eng_2 w of None ⇒ False | Some m_2 ⇒ (
        m_1 = m_2 ∨ prefers w WPrefs m_2 m_1)))"
```

`married_better` defines a relation between any 2 `engagements` states with respect to a given woman `w`. It is true if and only if `w` is (non-strictly) better off in `eng_2` relative to `eng_1`. The relation is clearly transitive and reflexive, and can be easily proven to be so.

```
lemma GS'_arg_seq_all_w_marry_better:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs;
  is_distinct engagements; i < length seq; seq!i = (X_pre, Y_pre);
  j < length seq; seq!j = (X_post, Y_post);
  i ≤ j⟧ ⟹ married_better w WPrefs X_pre X_post"
proof (induction "j - i" arbitrary:j X_post Y_post)
  …
qed
```

The lemma is formulated to hold for any 2 X's in the argument sequence. Since `married_better` is transitive, to prove the lemma, we need only show `married_better` for 2 consecutive X's. This is done using the same reasoning as discussed above in the observations, involving a case analysis on whether `w` is the receiver of free man `m`'s proposal.

Because examining consecutive X's will suffice, a proof by computational induction will also work for `all_w_marry_better`. Nevertheless, induction on `j - i` is most intuitive. In the base case (`j = i`), `X_pre` and `X_post` are the same, so we are done by reflexivity. In the inductive step, the induction hypothesis gives that, for `X_post_pre` the X immediately before `X_post`, `married_better w WPrefs X_pre X_post_pre`. We then use the step lemmas to prove `married_better w WPrefs X_post_pre X_post` by deriving the form of `X_post` in terms of `X_post_pre`. Transitivity completes the proof.

`is_distinct engagements` is added as a premise, from which `all_distinct`, proven earlier, gives that any X in the sequence `is_distinct`. This enables the `X!m = Some w ⟹ findFiance X w = Some m` derivation, which isn't possible otherwise since `w` could have 2 different fiancés.

6.5 Miscellaneous Lemmas: Sanity Checks

This section lists a linearly progressing set of self-explanatory lemmas, where the later ones may depend on the earlier ones, with brief notes on their proofs.

```
lemma GS'_arg_seq_prev_prop_idx_same_or_1_less:
  … (has been stated just before title of 5.1)
proof -
  …
qed
```

Proof is by the step lemmas and contradiction. A case analysis on whether `m` and `the (findFreeMan X_prev)` are equal is used.

```
lemma GS'_arg_seq_exists_prev_prop_idx:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                                  engagements (replicate N 0)"
      and "i < length seq" and "seq!i = (X, Y)"
      and "k = Suc k_1" and "m < N" and "Y!m = k"
    shows "∃j X_prev Y_prev. j < i ∧ seq!j = (X_prev, Y_prev)
```

```
                         ∧ Y_prev!m = k_1 ∧ findFreeMan X_prev = Some m"
proof (rule ccontr)
  …
qed
```

`ccontr` stands for classical contradiction. If we assume for a contradiction that the goal is not true, then using `prev_prop_idx_same_or_1_less`, from `i`, we can move `j` backwards one step at a time (by induction on `i-1-j`), and derive that `Y!m` must equal k all the way to the very beginning. `(replicate N 0)!m = 0`, yet k is non-zero, and so we are done.

This lemma essentially states that, if currently in the trace, `m` will next propose to the woman indexed k in his preferences, we must be able to locate the earlier point in time where he proposed to the woman indexed `k-1`.

```
lemma GS'_arg_seq_all_prev_prop_idxs_exist:
"⟦seq = GS'_arg_seq N MPrefs WPrefs engagements (replicate N 0);
  i < length seq; seq!i = (X, prop_idxs); m < N; prop_idxs!m = k;
  prop_idx < k⟧ ⟹ ∃j X_prev Y_prev. j < i ∧ seq!j = (X_prev, Y_prev)
                 ∧ Y_prev!m = prop_idx ∧ findFreeMan X_prev = Some m"
proof (induction "k-1 - prop_idx" arbitrary: prop_idx)
  …
qed
```

This lemma states that not only are we able to find the moment where `m` proposed to the woman indexed `k-1` in his list, we are able to find all the other earlier-made proposals as well. The proof is by induction and `exists_prev_prop_idx`.

```
lemma GS'_arg_seq_married_once_proposed_to:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs engagements prop_idxs"
      and "is_distinct engagements" and "Suc i < length seq"
      and "seq!i = (X, Y)" and "seq!Suc i = (X_next, Y_next)"
      and "findFreeMan X = Some m" and "w = MPrefs!m!(Y!m)"
    shows "∃m'. findFiance X_next w = Some m' ∧ (m' = m
                              ∨ m' ≠ m ∧ ¬ prefers w WPrefs m m')"
proof (cases "findFiance X w")
  …
qed
```

This lemma states that, after m proposes to w, w must be married to either m or someone she prefers over m. The proof is by the step lemmas.

I am now able to state and prove the key result that leads to noFree and is_bounded.

6.6 Lemma: any_man_done_proposing_means_done

```
lemma GS'_arg_seq_any_man_done_proposing_means_done:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                                (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs" and "i < length seq"
      and "seq!i = (engagements, prop_idxs)" and "m < N"
      and "prop_idxs!m = N"
    shows "findFreeMan engagements = None"
```

This is the result mentioned in 6.3. Once any man runs out of proposals to make, we must be done. This is because, at this point, the man has already proposed to all the women. For each woman, once anyone proposes to her, she becomes married due to married_once_proposed_to, and stays married, due to all_w_marry_better. Hence, at this point, all N women must be married. Of course, thus, all the men, including this man, are married as well. No man is unmarried: Q.E.D. We thus additionally have is_terminal, and must be looking at the very end of the argument sequence.

To summarize: as soon as any man has ran out of proposals, Gale_Shapley' terminates. This is why GS' during computation will never encounter a free man m with no proposals left.

The Isar proof for any_man_done_proposing_means_done, essentially a translation of the argument presented above to machine-checkable form, is attached below.

```
proof -
  let ?Some_Ns = "map Some [0 ..< N]"
  have distinct:"is_distinct (replicate N None)" by simp
```

```
  have "∀prop_idx < length (MPrefs!m).
                  findFiance engagements (MPrefs!m!prop_idx) ≠ None"
    apply (rule)
  proof
    fix prop_idx
    let ?w = "MPrefs!m!prop_idx"
    assume "prop_idx < length (MPrefs!m)"
    also have "... = N" …
    finally have "prop_idx < N" .
    from GS'_arg_seq_all_prev_prop_idxs_exist[OF assms(1,3-6) this]
    obtain j X_prev Y_prev X' Y' where "j < i"
      and "seq!j = (X_prev, Y_prev)" and X_Y:"seq!Suc j = (X', Y')"
      and "findFreeMan X_prev = Some m" and "Y_prev!m = prop_idx" …
    from GS'_arg_seq_married_once_proposed_to[OF assms(1) distinct
           less_trans_Suc[OF this(1) assms(3)] this(2-4)] this(5)
    have "findFiance X' ?w ≠ None" by fastforce
    moreover have "married_better ?w WPrefs X' engagements"
      using GS'_arg_seq_all_w_marry_better[OF assms(1) distinct
             less_trans_Suc[OF `j<i` assms(3)] X_Y assms(3,4)
             Suc_leI[OF `j<i`]] by fast
    ultimately show "findFiance engagements ?w ≠ None" …
  qed
  hence "∀w ∈ set [0 ..< N]. findFiance engagements w ≠ None" …
  hence "set ?Some_Ns ⊆ set engagements" …
  moreover have "card (set ?Some_Ns) = N"
  proof -
    have "distinct xs ⟹ distinct (map Some xs)" for xs
      apply (induction xs)
      by auto
    hence "distinct ?Some_Ns" by simp
    from distinct_card[OF this] show ?thesis by simp
  qed
  moreover have "card (set engagements) ≤ N"
            and "finite (set engagements)" …
  ultimately have "set ?Some_Ns = set engagements"
    by (metis card_seteq)
  with … show ?thesis by auto
qed
```

The scripts above differ from the version in the source theory file by only a few … omissions made for readability reasons. It should be fairly readable even to those unfamiliar with Isar, and demonstrates Isar's usability and readability.

The brief sentences in the paragraphs preceding the scripts already capture the core of the reasoning behind any_man_done_proposing_means_done. But formulating it and proving it fully requires a lot more work, such as what was done in 6.5, involving lemmas obviously true but not necessarily easy to prove, and tidying up loose ends. Work like this constitutes the bulk of the temporal cost of any formalization.

6.7 Corollaries

From any_man_done_proposing_means_done, 3 important results follow immediately.

```
lemma GS'_arg_seq_prop_idx_bound:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                                (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs" and "i < length seq"
      and "seq!i = (engagements, prop_idxs)" and "m < N"
    shows "prop_idxs!m ≤ N"
proof (rule ccontr)
  …
qed
```

The proof is by contradiction. Should any prop_idxs!m be strictly greater than N, by all_prev_prop_idxs_exist, we can locate the earlier point in the trace where it was N, and m was free and chosen. This is an obvious contradiction, as we know that m, along with all other men, must be married at this point.

```
lemma GS'_arg_seq_prop_idx_bound_non_terminal:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                                (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs" and "i < length seq"
      and "seq!i = (engagements, prop_idxs)" and "m < N"
```

```
    and "¬is_terminal N engagements prop_idxs"
  shows "prop_idxs!m < N"
proof (rule ccontr)
  …
qed
```

From the earlier corollary, additionally excluding the very end of the argument sequence from consideration, we are able to show that any `prop_idx` must be strictly less than N, and thus within-bounds of any man's preference list. This is because `any_man_done_proposing_means_done` says that a `prop_idx` can only be N at the very end of the argument sequence. `is_bounded` should quickly follow, as we have proven that the statement `w = MPrefs!m!(prop_idxs!m)` is always well-behaved w.r.t. index bounds, so showing `w < N` is now straightforward from the format of `MPrefs`.

```
lemma GS'_arg_seq_N_imp_prev_bump:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                                (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs" and "i < length seq"
      and "seq!i = (engagements, prop_idxs)" and "m < N"
      and "prop_idxs!m = N"
    shows "∃i_1 N_1 X_prev Y_prev. i = Suc i_1 ∧ N = Suc N_1 ∧
                                     seq!i_1 = (X_prev, Y_prev) ∧
                         Y_prev!m = N_1 ∧ findFreeMan X_prev = Some m"
proof -
  …
qed
```

This lemma says that, if `prop_idxs!m = N`, not only must we be at the end of the argument sequence, in the second-to-last argument pair `(X_prev, Y_prev)`, we must have `Y_prev!m = N - 1`, and `findFreeMan X_prev = Some m`. The proof is simple: from `prev_prop_idx_same_or_1_less`, we just need to rule out the possibility that `Y_prev!m = N`, and will have the result needed. This is indeed impossible: we cannot have 2 distinct ends-of-sequence.

6.8 noFree

Recall that, from `last_eq_terminal` and `GS'_arg_seq_computes_GS'`, we already know that the X at the end of the argument sequence both `is_terminal` and is equal to `(Gale_Shapley MPrefs WPrefs)`. `is_terminal` asserts at least one of three predicates: that X Y differ in length, that Y's sum is at least N*N, or that `findFreeMan X = None`. We want to show the last predicate for `noFree`, and should thus rule out the first 2. This amounts to proving that the 2 input-sanitization early-exit checks added on top of the standard pseudocode are indeed extraneous, i.e. that they will always be unnecessary for well-formatted inputs (see end of 3.2). In other words, ruling out the first 2 predicates amounts to saying, if `Gale_Shapley'`, called by `Gale_Shapley` on well-formatted inputs, terminates, it must have terminated due to the 3rd condition, i.e. the standard Gale-Shapley terminal condition that all the men are married (a.k.a. `noFree`).

Ruling out the first is trivial, so the key lies in ruling out the second. This is the "bit of arithmetic" mentioned earlier in 6.3.

```
theorem GS'_arg_seq_never_reaches_NxN:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                              (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs" and "N ≥ 2"
      and "i < length seq" and "seq!i = (engagements, prop_idxs)"
    shows "sum_list prop_idxs < N * N"
proof (rule ccontr)
  …
qed
```

Note how the lemma holds throughout the argument sequence, not just the end.

The proof is by contradiction. If we assert the second predicate, `sum_list Y ≥ N * N`, then we must have at least one `m` such that `Y!m ≥ N`, otherwise `sum_list Y ≤ N * (N - 1)`. We thus have `Y!m = N` from `prop_idx_bound`. We know also that no other `m' ≠ m` will satisfy `Y!m' = N`, because a contradiction would arise due to `N_imp_prev_bump`, where `findFreeMan X_prev` is both `m` and `m'`. Hence, with `prop_idx_bound`, there is precisely one N in Y, and every other `prop_idx` in Y is strictly less than N. This means we can bound `sum_list Y` to `sum_list Y ≤ N * (N - 1) + 1`. Given $N \geq 2$, we know this bound is strictly less than `N * N`, ruling out the second predicate. Q.E.D.

6.9 Lemma: GS'_arg_seq_all_bounded

With the `prop_idx_bound` lemmas, I can finally formulate and prove `is_bounded` using invariant-preservation, like `GS'_arg_seq_all_distinct`. There is a bit of fiddling in the proof to show `w < N` not worth reporting here; the core reasoning has already been discussed in 6.3 and 6.7.

```
lemma GS'_arg_seq_all_bounded:
"⟦seq = GS'_arg_seq N MPrefs WPrefs
                       (replicate N None) (replicate N 0);
  is_valid_pref_matrix N MPrefs; i < length seq;
  seq!i = (engagements, prop_idxs)⟧ ⟹ is_bounded engagements"
proof (induction i arbitrary:engagements prop_idxs)
  …
qed
```

What is worth mentioning is that, while we have made frequent use of the step lemmas already, `all_bounded` is what demonstrates the *need* for them, as the alternative of using computational induction no longer works here. Computational induction works best when used on a property that holds throughout the set S containing all sequences of the form `GS'_arg_seq` $args$, i.e. the image of the function `GS'_arg_seq`. With certain tricks we can still use computational induction to prove properties only holding for a subset of S. Failure is certain, however, when attempting computational induction on a property that holds for one single member of S only, or too narrow a subset of S. We have in the various underlying lemmas hardcoded premises like `prop_idxs = (replicate N 0)`, ending up with a form of `all_bounded` that holds only for the standard sequence initiated by a call to `Gale_Shapley`, starting with `0`'s and `None`'s. It would be difficult, if not impossible, and certainly unnecessary, to try to define a nontrivial subset of S where `all_bounded` holds throughout, for the sake of sticking with computational induction. Concretely, as an example, if we include in $\mathrm{P}\, args$ the premise `prop_idxs = (replicate N 0)`, we will certainly fail to show it for $\mathrm{P}\, args_1$: `next_prop_idxs ≠ (replicate N 0)`. In any case, we have at this point completed all 3 parts required by `is_matching_intro`.

7 Stability

Only one additional lemma is needed before a direct attack on `stable` is possible.

7.1 Lemma: GS'_arg_seq_be_brave

```
lemma GS'_arg_seq_be_brave:
"⟦seq = GS'_arg_seq N MPrefs WPrefs (replicate N None) prop_idxs;
  i < length seq; seq!i = (X, Y); m < N;
  ∄j X' Y'. j < i ∧ seq!j = (X', Y') ∧ findFreeMan X' = Some m ∧
                                        MPrefs!m!(Y'!m) = w⟧
    ⟹ X!m ≠ Some w"
proof (induction i arbitrary:X Y)
  …
qed
```

`be_brave` says that, should man `m` ever wish to be with woman `w`, he must have proposed to her first. The lemma is formulated using the contrapositive. This is another example where a constraint on the argument sequence is necessary, i.e. `initial engagements = (replicate N None)`. Without this, trivial counterexamples follow. The proof is by induction and the step lemmas, where in the inductive step, I check that, given that `m` is not with `w` at the moment (by induction hypothesis), and that `m` is also not currently proposing to `w` (by premise), `m` still won't be with `w` at the end of this iteration.

7.2 Theorem: GS'_arg_seq_all_stable

Please refer to 3.4 for the definition of `unstable`.

```
theorem GS'_arg_seq_all_stable:
  assumes "seq = GS'_arg_seq N MPrefs WPrefs
                              (replicate N None) (replicate N 0)"
      and "is_valid_pref_matrix N MPrefs"
      and "is_valid_pref_matrix N WPrefs"
      and "i < length seq" and "seq!i = (X, Y)"
```

```
    shows "¬ unstable MPrefs WPrefs X"
proof
  assume "unstable MPrefs WPrefs X"
  …
  … show False …
qed
```

Note that this refutes the existence of unstable pairs throughout the argument sequence, not just at the end. The proof, unlike most of the earlier non-trivial results, no longer makes use of any form of induction. It is instead a dense set of logical deduction and case analysis steps based on the vast set of lemmas and results introduced earlier. It is best summarized instead of reproduced here as Isar scripts.

We assume that an unstable pair exists for a contradiction. This means that we have married pairs (`m1`, `w1`) and (`m2`, `w2`), where `w1` and `m2` both prefer each other over their current partners.

From `be_brave` and the fact that `m2` is currently married to `w2`, we know that `m2` must have proposed to `w2` earlier.

In addition, because `m2` prefers `w1` over `w2`, by `all_prev_prop_idxs_exist`, we can locate the even earlier point where `m2` proposed to `w1` first. By `married_once_proposed_to`, immediately after this point, `w1` must be married to either `m2` or someone she prefers over `m2`.

Yet, `w1` is currently married to `m1`. We have assumed that `w1` prefers `m2` over `m1`. This means that `w1` is currently worse off than the earlier point where she received `m2`'s proposal, and thus contradicts `all_w_marry_better`. Q.E.D.

The Isar proof involves quite a bit more fiddling.

In any case, from `all_stable`, [`stable`]() follows immediately.

8 Related Work

Hamid and Castleberry [3] presented an implementation of Gale-Shapley in the Coq proof assistant. The standard tail-recursion paradigm is used for the iterative-functional translation, same as this formalization. They used an additional redundant check before selecting the man `m` for the iteration: to check not only that he is free, but also that he still has proposals left to make. This amounts to sidestepping all the difficulty encountered when directly attempting `is_bounded` or `noFree` in this formalization, i.e. the need to bound `prop_idx!m`.

More generally, they liberally added extraneous if-checks for cases that will never occur in actual execution in order to sidestep the difficulty of formally proving that the corresponding predicate is indeed impossible.

This formalization avoids such extraneous checks wherever possible. Where required, they are used, but also accompanied by a proof that they are certainly redundant, e.g. `never_reaches_NxN`.

Furthermore, Hamid and Castleberry's work completely missed core results such as `all_w_marry_better`, and failed to prove both termination (bijection, all-married) and correctness (stability).

9 Conclusion

The outcome of this formalization is a single, fully machine-checked, Isabelle/HOL .thy theory file `Gale_Shapley.thy` consisting of around 1400 lines of definitions and Isar proof scripts, available at:

https://github.com/20051615/Gale-Shapley-formalization/blob/master/Gale_Shapley.thy

The key, highest-level outcomes from the theory file correspond to the specification: a definition of the function `Gale_Shapley::pref_matrix ⇒ pref_matrix ⇒ matching`, and proven facts concerning its output `(Gale_Shapley MPrefs WPrefs)` for arbitrary, well-formatted inputs `MPrefs WPrefs`: that it is a permutation of `[Some 0, Some 1, …, Some N-1]`, and that it is `¬ unstable`. This is the first formalization of the Gale-Shapley algorithm that includes complete, machine-checked proofs of both termination and correctness that I know of.

The formalization is a formal, maximally rigorous explanation of why the Gale-Shapley algorithm, in its original form and design, is both terminating (marries everyone) and correct (outputs a stable result). For termination, at first glance, the possibility of divorcing `m'` from his partner as well as the possibility of making near-zero progress in the case that `m` is rejected raise concerns. However, since each woman is continually made better off by design, the occurrence of any proposal guarantees that the receiver remains married thereafter, and so iterations of proposals do approach an all-married, final, terminating state. It is not possible for an unmarried man to not have any proposals left to make, because at this point the women must all have received at least one proposal. Stability comes from the design choice of men proposing strictly in order of preference, and women continually “upgrading” whenever possible and never the reverse. As such, we have answered the questions raised in 3.1.

Chief among the non-obvious properties of the Gale-Shapley algorithm found during this formalization is that, though the proposals take place chaotically, with no ordering constraints apart from requiring that the proposer is unmarried and proposes in order of his preferences, Gale-Shapley terminates precisely after each and every woman has received at least one proposal. Indeed, this may occur before any man ends up proposing to everyone.

10 Bibliography

[1] David Gale and Lloyd Shapley. 1962. College Admissions and the Stability of Marriage. *The American Mathematical Monthly, 69*(1), 9-15. doi:10.2307/2312726

[2] Alvin Elliot Roth and Marilda Sotomayor. 1990. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Econometric Society Monographs. Cambridge University Press.

[3] Nadeem Abdul Hamid and Caleb Castleberry. 2010. Formally certified stable marriages. In *Proceedings of the 48th Annual Southeast Regional Conference* (*ACM SE '10*). Association for Computing Machinery, New York, NY, USA, Article 34, 1–6. DOI:https://doi.org/10.1145/1900008.1900056

11 Appendix

11.1 Isabelle/HOL Reference

`xs!i`

the `i`-th element of list `xs`. Does not simplify if `i` out-of-bounds.

`term::type`

type annotation.

`Suc n`

equals `n+1`. Natural numbers in Isabelle/HOL are constructed in the canonical way using the recursive datatype, `nat = 0 | Suc nat`.

`sum-list xs`

computes the sum of elements in `xs`

`xs[i:=x]`

a copy of list `xs`, all the same except at location indexed `i`, now updated to hold `x`. Simply `xs` if `i` out-of-bounds.

`replicate n term`

a length-n list, `[term, term, …, term]`

`set xs`

the list-to-set conversion function

`Ball (set xs) predicate`

`True` if and only if $\forall x \in \mathrm{set}\, xs\,.\,(\mathrm{predicate}\; x)$

`[0 ..< N]`

`[0, 1, …, N-1]`

`the (Some x)`

= `x`. Does not simplify if `x = None`