**Journal of Automated Reasoning manuscript No.**
(will be inserted by the editor)

# Gale-Shapley Verified

**Tobias Nipkow**

**Abstract** This paper presents a detailed verification of the Gale-Shapley algorithm for stable matching (or marriage). The verification proceeds by stepwise transformation of programs and proofs. The initial steps are on the level of imperative programs, ending in a linear time algorithm. An executable functional program is obtained in a last step. The emphasis is on the stepwise development of the algorithm and the required invariants.

## 1 Introduction

The Gale-Shapley algorithm [8] solves a matching problem: it finds a stable matching between two sets of elements given an ordering of preferences for each element. This work spawned a subfield of research, stable matching. The algorithm is of great practical importance: variations of it have been used for matching medical school students and hospital training programs since the 1950s [11] and are used today for matching clients and servers in Akamai's content delivery network [20]. The textbook by Kleinberg and Tardos [14] presents the problem and the algorithm as the first of five representative algorithm design problems. Shapley and Roth were awarded the 2012 Prize in Economic Sciences in Memory of Alfred Nobel "for the theory of stable allocations". Nevertheless, no formally verified correctness proof of the algorithm can be found in the literature (see Section 14).

This paper presents the first formally verified development of a linear-time executable implementation of the Gale-Shapley algorithm (using the proof assistant Isabelle [25,24]). The formalization is available online [23]. The development of the final algorithm is by stepwise transformation. By accident we discovered a small defect in a proof rule in a well-known program verification textbook [3,2] that had gone unnoticed for 30 years (see Section 9).

Technical University of Munich, Germany
https://www.proof.cit.tum.de/~nipkow

**Algorithm 0** The informal Gale-Shapley algorithm as presented by Gusfield and Irving

```
assign each element to be free;
while some a in A is free do
begin
  b := first B on a's list to whom a has not yet tried to be matched;
  if b is free then
    assign a and b to be matched
  else
    if b prefers a to its current match a' then
      assign a and b to be matched and a' to be free
    else
      b rejects the match with a and a remains free
end
```

## 2 Problem and Algorithm

We start with the informal presentation of the problem and algorithm by Gusfield and Irving in their well-known monograph [11]. However, our terminology avoids all reference to gender.[1]

There are two disjoint sets A and B with $n$ elements each. Each element has a strictly ordered preference list of all the members of the other set. Element $p$ prefers $q$ to $q'$ iff $q$ precedes $q'$ on $p$'s preference list. The problem is to find a stable matching, i.e. a subset of A×B, with the following properties:

- Every $a \in$ A is matched with exactly one $b \in$ B and vice versa.
- The matching is *stable*: there are no two elements from opposite sets who would rather be matched with each other than to the elements they are actually matched with.

The Gale-Shapley algorithm (Algorithm 0) is guaranteed to find such a stable matching in $O(n^2)$ iterations. Moreover, because the $a \in$ A get to choose, the resulting matching is A-optimal: there is no other stable matching between A and B with the given preferences where some $a \in$ A does better, i.e. is matched to a $b \in$ B that $a$ prefers to the computed match.

## 3 Isabelle Notation

Isabelle types are built from type variables, e.g. $'a$, and (postfix) type constructors, e.g. $'a\ list$. The infix function type arrow is $\Rightarrow$. The notation $t :: \tau$ means that term $t$ has type $\tau$. Isabelle (more precisely Isabelle/HOL, the logic we work in) provides types $'a\ set$ and $'a\ list$ of sets and lists of elements of type $'a$. They come with the following notations: $f$ ‘ $A$ (the image of set $A$ under $f$), function $set$ (conversion from lists to sets), $x \cdot xs$ (list with head $x$ and tail $xs$), $|xs|$ (length of list $xs$), $xs\ !\ i$ (the $i$th element of $xs$ starting at 0), and $xs[i := x]$ (list $xs$ where the $i$th

[1] Because Reviewer 2 of a draft version of this article was hurt by the gender-based terminology and made a plea not to reproduce and disseminate such outdated terminology, I have completely neutralized all reference to gender. I need to warn all sensitive persons that reading references [8,11,15,16,13] may hurt or offend them. Citing those sources does in no way constitute support for their outdated terminology.

element has been replaced by $x$). Throughout the paper, all numbers are of type *nat*, the type of natural numbers.

Data type *'a option* is also predefined:

**datatype** *'a option* = *None* | *Some 'a*

## 4 Hoare Logic and Stepwise Development

Most of the work in this paper is carried out on the level of imperative programs. The Hoare logic used for this purpose is based on work by Gordon [10] and was added to Isabelle by Nipkow in 1998. Possibly the first published usage was by Mehta and Nipkow [21]. Recently Guttmann and Nipkow added *variants*, i.e. expressions of type *nat* that should decrease with every iteration of a loop. Total correctness Hoare triples have the syntax $[P]\ c\ [Q]$ where $P$ and $Q$ are pre- and post-conditions and $c$ is the program. Loops must be annotated with invariants and variants like this:

*WHILE condition INV* {*invariant*} *VAR* {*variant*} *DO body OD*

The implementation of the Hoare logic comes with a verification condition generator.

We progress from the first to the last algorithm by stepwise transformation. In each step we restate the full modified algorithm, which is not an issue for algorithms of 15-20 lines. Readjusting the proofs to a modified algorithm is reasonably easy: the proofs of the verification conditions are all relatively short (about 60 lines per algorithm), they are structured and readable [26,27], and our transformation steps are small. Most importantly, the key steps in the proofs are formulated as separate lemmas that are reused multiple times. For example, we may have an algorithm with some set variable $S$, prove some lemma about $S$, refine the algorithm by replacing $S$ by a list variable $L$, and reuse the very same lemma with $S$ instantiated by *set L*.

Although this methodology is inspired by stepwise refinement, in particular data refinement, we avoid the word refinement because it suggests a modular development. Instead we speak of stepwise transformation or development and of data concretisation.

This methodology is not intended for the development of large algorithms but for small intricate ones. I can confidently say that it worked well for Gale-Shapley: I spent most of my time on the core proofs and very little on copy-paste-modify. Structured proofs helped to localize the effect of most changes. The problems of code duplication in programming do not arise because the theorem prover keeps you honest.

It should be mentioned that there is a refinement framework in Isabelle [18] and it would be interesting to see how the development of Gale-Shapley plays out in it.

## 5 Formalization of the Stable Matching Problem

We do not refer to the actual sets A and B and their elements directly but only to their indices $0, \ldots, n-1$. We use mnemonic names like $a$ and $b$ to indicate which

of the two sets these indices range over. We speak of $a$'s and $b$'s to mean sets of indices. By $\{<n\}$ we abbreviate the set of all $i < n$.

We fix the following variables:

- The cardinality $n :: nat$
- The preference lists $P$, $Q :: nat\ list\ list$. For each $a \in \{<n\}$, $P\ !\ a$ is the preference list of $a$, i.e. a list of all $b$'s in decreasing order of preference. Dually for $Q$. We assume that the preference lists have the right lengths and are permutations of $\{<n\}$:

$$\left.\begin{array}{l} n = |P| = |Q| \\ a < n \longrightarrow |P\ !\ a| = n \wedge set\ (P\ !\ a) = \{<n\} \\ b < n \longrightarrow |Q\ !\ b| = n \wedge set\ (Q\ !\ b) = \{<n\} \end{array}\right\} Pref$$

These properties are used implicitly in proofs we discuss.

It is important to emphasize that although we model everything in terms of lists, in a last step these lists will be implemented as arrays to obtain constant time access to each element.

The notation $P \vdash x < y$ means that $x$ occurs before $y$ (meaning $x$ is preferred to $y$) in the preference list $P$:

$(P \vdash x < y) = (index\ P\ x < index\ P\ y)$
$index\ []\ \_ = 0$
$index\ (x \cdot xs)\ y = (\mathsf{if}\ x = y\ \mathsf{then}\ 0\ \mathsf{else}\ index\ xs\ y + 1)$

The algorithm tries to match each $a$ first with $P\ !\ a\ !\ 0$, then with $P\ !\ a\ !\ 1$ etc. Thus we do not record $a$'s current match $b = P\ !\ a\ !\ i$ but we record the $i$, and increment $i$ every time a match $(a, b)$ has to be undone. A list $A :: nat\ list$ (of length $n$) will record the index for the current match of each $a$. The actual match $b$ is $P\ !\ a\ !\ (A\ !\ a)$ and we define

$match\ A\ a = P\ !\ a\ !\ (A\ !\ a)$

Note that unless the algorithm has really matched $a$ and $b = match\ A\ a$, then $b$ is merely the current top choice of $a$.

In the sequel, $A$ will always represent such a list of indices into the preference lists. The predicate $wf\ A$ expresses that $A$ associates every $a < n$ with some $b < n$:

$wf\ A = (|A| = n \wedge set\ A \subseteq \{<n\})$

This means that $match\ A$ is a function from $\{<n\}$ to $\{<n\}$.

To improve readability we introduce the following suggestive notation:

$A\langle a\rangle = match\ A\ a$
$A\langle M\rangle = match\ A\ `\ M$

where $M :: nat\ set$.

### 5.1 Stable Matchings

We are looking at the special case of bipartite matching where every $a$ and $b$ are connected. A *matching* on $M \subseteq \{<n\}$ is an injective function from $M$ to $\{<n\}$.

$matching\ A\ M = (wf\ A \wedge inj\text{-}on\ (match\ A)\ M)$

The predicate $inj\text{-}on\ f\ S$ means that $f :: 'a \Rightarrow 'b$ is injective on $S :: 'a\ set$.

We want a *stable* matching, i.e. one where there are no "unstable" matches $(a, b)$ and $(a', b')$ such that $a$ prefers $b'$ to $b$ and $b'$ prefers $a$ to $a'$:

$stable\ A\ M = (\neg\ (\exists\, a{\in}M.\ \exists\, a'{\in}M.\ P\ !\ a \vdash A\langle a'\rangle < A\langle a\rangle \wedge Q\ !\ A\langle a'\rangle \vdash a < a'))$

We will not just show that the Gale-Shapley algorithm finds a stable matching but also that this matching is A-optimal, i.e. there is no stable matching where some $a$ can do better:

$optiA\ A =$
$(\nexists\, A'.\ matching\ A'\ \{{<}n\} \wedge stable\ A'\ \{{<}n\} \wedge (\exists\, a{<}n.\ P\ !\ a \vdash A'\langle a\rangle < A\langle a\rangle))$

The dual property is B-pessimality, i.e. no $b$ can do worse:

$pessiB\ A =$
$(\nexists\, A'.\ matching\ A'\ \{{<}n\} \wedge stable\ A'\ \{{<}n\}$
$\quad \wedge (\exists\, a{<}n.\ \exists\, a'{<}n.\ A\langle a\rangle = A'\langle a'\rangle \wedge Q\ !\ A\langle a\rangle \vdash a < a'))$

Unsurprisingly, it is easy to prove that any A-optimal $A$ is $b$-pessimal:

$optiA\ A \longrightarrow pessiB\ A$

### 5.2 Stepwise Development

The following points remain unchanged throughout the development process:

- The matchings are recorded as a variable $A$ as described above. How it is recorded if some $a$ has been matched to $A\langle a\rangle$ or not changes during the development process.
- The precondition always assumes $A = replicate\ n\ 0$, where $replicate\ m\ x$ is a list of $m$ $x$'s. That is, all $a$'s start at the beginning of their preference list. Making this assumption a precondition avoids a trivial initialization loop. We will frequently deal with initializations like this.
- The postcondition is always $matching\ A\ \{{<}n\} \wedge stable\ A\ \{{<}n\} \wedge optiA\ A$

## 6 Algorithm 1

Algorithm 1 follows the informal algorithm by Gusfield and Irving [11] quite closely. Variable $M$ records the set of $a$'s that have been matched. Initially no $a$ has been matched.

At the beginning of each iteration an unmatched $a$ is picked via Hilbert's choice operator: $SOME\ x.\ P$ is some $x$ that satisfies $P$. If there is no such $x$, we cannot deduce anything (nontrivial) about $SOME\ x.\ P$. However, we only use the choice operator in cases where there is a suitable $x$. If there are multiple $x$, $SOME\ x.\ P$ will return an arbitrary fixed one. Although the choice is deterministic, our proofs work by merely assuming that some suitable $x$ has been picked and thus the algorithm would work just as well with a nondeterministic choice. In the end, the exact nature of $SOME$ is irrelevant: only the first two programs use $SOME$, it is transformed away afterwards.

**Algorithm 1**

```
1  [M = ∅ ∧ A = replicate n 0]
2  WHILE M ≠ {<n} INV {invAM A M}  VAR {var A M}
3  DO a := (SOME a. a < n ∧ a ∉ M);
4      b := A⟨a⟩;
5      IF b ∉ A⟨M⟩ THEN M := M ∪ {a}
6      ELSE a' := (SOME a'. a' ∈ M ∧ A⟨a'⟩ = b);
7           IF Q ! A⟨a'⟩ ⊢ a < a'
8           THEN A := A[a' := A ! a' + 1]; M := M − {a'} ∪ {a}
9           ELSE A := A[a := A ! a + 1] FI
10     FI
11 OD
12 [matching A {<n} ∧ stable A {<n} ∧ optiA A]
```

The term $(SOME\ a'.\ a' \in M \wedge A\langle a'\rangle = b)$ expresses the inverse of *match A* applied to $b$.

In each iteration, one of three possible basic actions is performed, where $a$ is unmatched and $b = A\langle a\rangle$:

- **match** (line 5): $b$ was unmatched; now $a$ is matched (to $b$).
- **swap** (line 8): $b$ was matched to some $a'$ but $b$ prefers $a$ to $a'$; now $a'$ is unmatched and moves to the next element on its preference list and $a$ is matched (to $b$)
- **next** (line 9): $b$ was matched to some $a'$ and $b$ prefers $a'$ to $a$; now $a$ moves to the next element on its preference list.

We will now detail the correctness proof, first of the invariant, then of the variant.

### 6.1 The Invariant

$$invAM\ A\ M = (matching\ A\ M \wedge M \subseteq \{<n\} \wedge pref\text{-}match\ A\ M \wedge optiA\ A)$$

$$pref\text{-}match\ A\ M =$$
$$(\forall\, a{<}n.\ \forall\, b{<}n.\ P\ !\ a \vdash b < A\langle a\rangle \longrightarrow (\exists\, a'{\in}M.\ b = A\langle a'\rangle \wedge Q\ !\ b \vdash a' < a))$$

The predicate *pref-match* says that if $a$ prefers $b$ to its match $A\langle a\rangle$ then $b$ is matched to some $a'$ who $b$ prefers to $a$. This is the invariant way of expressing this more operational or temporal property used by Knuth [16] (adapted):

Point 2. If $a$ prefers $b$ to its match $b'$, it means that $b$ has rejected $a$ for another.

An alternative formulation of *pref-match* is based directly on the indices below $A\ !\ a$:

$$pref\text{-}match'\ A\ M =$$
$$(\forall\, a{<}n.\ \forall\, b{\in}preferred\ A\ a.\ \exists\, a'{\in}M.\ b = A\langle a'\rangle \wedge Q\ !\ b \vdash a' < a)$$
$$preferred\ A\ a = (!)\ (P\ !\ a)\ `\{<\ A\ !\ a\}$$

where (!) is the prefix version of the infix !. Both predicates are equivalent

$$wf\ A \longrightarrow pref\text{-}match'\ A\ M = pref\text{-}match\ A\ M$$

and we use whichever is more convenient in a given situation.

The invariant can be seen as a generalization of the postcondition. The missing link is this lemma from which it follows directly that the invariant together with $M = \{<n\}$ implies the postcondition:

**Lemma 1** *matching* $A$ $\{<n\} \wedge$ *pref-match* $A$ $\{<n\} \longrightarrow$ *stable* $A$ $\{<n\}$

*Proof* By contradiction. Assume there are $a_1$, $a_2 < n$, $a_1 \neq a_2$ such that $P\ !\ a_1 \vdash A\langle a_2\rangle < A\langle a_1\rangle$ and $Q\ !\ A\langle a_2\rangle \vdash a_1 < a_2$. Assumption *pref-match* $A$ $\{<n\}$ implies that there is an $a'$ such that $A\langle a_2\rangle = A\langle a'\rangle$ and $Q\ !\ A\langle a_2\rangle \vdash a' < a_1$. Injectivity of *match* $A$ on $M$ implies $a_2 = a'$ and thus we have both $Q\ !\ A\langle a_2\rangle \vdash a_1 < a_2$ and $Q\ !\ A\langle a_2\rangle \vdash a_2 < a_1$, a contradiction. □

The precondition $M = \emptyset \wedge A =$ *replicate* $n$ 0 is easily seen to establish the invariant.

### 6.2 Preservation of the Invariant

We need to show that *invAM* is preserved by the basic actions **match**, **swap** and **next**. For **match** this is easy and we concentrate on **swap** and **next**. We present the proofs bottom up, starting with the key supporting lemmas.

We begin by showing that *wf* $A$ is preserved. Both **swap** and **next** increment some $A\ !\ a$ where $a < n$ and $A\langle a\rangle \in A\langle M\rangle$ (**swap**: $a'$ instead of $a$). We need to show that the result is still $< n$. This is the corresponding lemma:

**Lemma 2** *wf* $A \wedge M \subset \{<n\} \wedge$ *preferred* $A$ $a \subseteq A\langle M\rangle \wedge a < n \wedge A\langle a\rangle \in A\langle M\rangle$
$\longrightarrow A\ !\ a + 1 < n$

*Proof* By contradiction. If $A\ !\ a + 1 = n$, then (!) $(P\ !\ a)$ ‘ $\{<n\} \subseteq A\langle M\rangle$ (by assumptions). Moreover (!) $(P\ !\ a)$ ‘ $\{<n\} = \{<n\}$ because $|P\ !\ a| = n \wedge$ *set* $(P\ !\ a) = \{<n\}$. Thus $\{<n\} \subseteq A\langle M\rangle$ and hence $n \leq |M|$. This is a contradiction because $M \subset \{<n\}$ implies $|M| < n$. □

The following lemma (proof omitted) shows that *optiA* is preserved by **swap** and **next**:

**Lemma 3**
*wf* $A \wedge a < n \wedge a' < n \wedge A\langle a'\rangle = A\langle a\rangle \wedge Q\ !\ A\langle a'\rangle \vdash a' < a \wedge$ *optiA* $A$
$\longrightarrow$ *optiA* $(A[a := A\ !\ a + 1])$

The following three lemmas (of which the first one is straightforward) express that *invAM* is preserved by the three basic actions **match**, **swap** and **next**.

**Lemma 4**
*invAM* $A$ $M \wedge a < n \wedge a \notin M \wedge A\langle a\rangle \notin A\langle M\rangle \longrightarrow$ *invAM* $A$ $(M \cup \{a\})$

**Lemma 5**
*invAM* $A$ $M \wedge a < n \wedge a \notin M \wedge a' \in M \wedge A\langle a'\rangle = A\langle a\rangle \wedge Q\ !\ A\langle a'\rangle \vdash a < a'$
$\longrightarrow$ *invAM* $(A[a' := A\ !\ a' + 1])$ $(M - \{a'\} \cup \{a\})$

*Proof* Preservation of *wf* $A$ follows from Lemma 2, preservation of injectivity is straightforward and preservation of *optiA* follows from Lemma 3. It remains to prove *pref-match'* $A'$ $M'$ where $A' = A[a := A\ !\ a + 1]$ and $M' = M - \{a'\} \cup \{a\}$:

$\forall\, x{<}n.\ \forall\, b{\in}\mathit{preferred}\ A'\ x.\ \exists\, y{\in}M'.\ b = A\langle y\rangle \wedge Q\ !\ b \vdash y < x$

where we have already replaced $A'\langle y\rangle$ by $A\langle y\rangle$ because $y \in M'$ implies $y \neq a'$. Now we distinguish if $x = a'$ or not.

If $x \neq a'$ then $\mathit{preferred}\ A'\ x = \mathit{preferred}\ A\ x$ and we have to show

$\forall\, b{\in}\mathit{preferred}\ A\ x.\ \exists\, y{\in}M'.\ b = A\langle y\rangle \wedge Q\ !\ b \vdash y < x.$

If $b \in \mathit{preferred}\ A\ x$, $\mathit{pref\_match}\ A\ M$ yields a witness $y \in M$. It remains to show that there is also a witness $y' \in M'$. This follows in the critical case $y = a'$ because $y' = a$ does the job: $Q\ !\ A\langle a'\rangle \vdash a' < x$ and $Q\ !\ A\langle a'\rangle \vdash a < a'$ imply $Q\ !\ A\langle a\rangle \vdash a < x$ because $A\langle a'\rangle = A\langle a\rangle$.

If $x = a'$ then $b = \mathit{preferred}\ A'\ a' = \mathit{preferred}\ A\ a' \cup \{A\langle a'\rangle\}$. If $b \in \mathit{preferred}\ A\ a'$ the claim follows from $\mathit{pref\_match}\ A\ M$. The fact that any witness $y \in M$ is also in $M'$ follows because $y = a'$ would imply $Q\ !\ A\langle a'\rangle \vdash a' < a'$, a contradiction. If $b = A\langle a'\rangle$ then $y = a$ is a suitable witness. □

**Lemma 6** $\mathit{invAM}\ A\ M \wedge a < n \wedge a \notin M \wedge a' \in M \wedge A\langle a'\rangle = A\langle a\rangle \wedge \neg\, Q\ !\ A\langle a'\rangle \vdash a < a' \longrightarrow \mathit{invAM}\ (A[a := A\ !\ a + 1])\ M$

The proof is similar to the one of the preceding lemma, but simpler.

## 6.3 The Variants

Our variants (see Section 4) are of the form $ub - \mathit{count}$ where *count* counts the number of iterations and $ub$ is some upper bound. It will follow trivially from our definitions of *count* that it increases with every iteration. Thus $ub - \mathit{count}$ decreases if the invariant and the loop condition imply $\mathit{count} < ub$; the latter is required because subtraction on *nat* stops at 0.

The term $ub$ is clearly an upper bound of the number of iterations. If the loop body executes in constant time, we can conclude that the loop has complexity $O(ub)$.

### *6.3.1 A Simple Variant*

Examining the loop body of Algorithm 1 we see that with each iteration either $|M|$ increases (action **match**) or it stays the same and one $A\ !\ a$ (where $a < n$) increases (actions **swap** and **next**). Thus *count* is $(\sum_{a\,<\,n} A\ !\ a) + |M|$. Because $|M|$ is bounded by $n$ and we will prove that every $A\ !\ a$ (where $a < n$) is bounded by $n - 1$, there is an obvious upper bound of $n * (n - 1) + n = n^2$. A possible variant is given by the following function of $A$ and $M$:

$\mathit{var}_0\ A\ M = n^2 - ((\sum_{a\,<\,n} A\ !\ a) + |M|)$

The following easy properties show that $\mathit{var}_0$ is decreased by all three basic actions:

$\mathit{wf}\ A \wedge M \subseteq \{{<}n\} \wedge a < n \wedge a \notin M \longrightarrow \mathit{var}_0\ A\ (M \cup \{a\}) < \mathit{var}_0\ A\ M$

$\mathit{wf}\ A \wedge M \subseteq \{{<}n\} \wedge M \neq \{{<}n\} \wedge a' < n$
$\longrightarrow \mathit{var}_0\ (A[a' := A\ !\ a' + 1])\ M < \mathit{var}_0\ A\ M$

This is the variant that Hamid and Castleberry [13] work with, except that they do not have $M$ but increment $A\ !\ a$ where we add $a$ to $M$. However, we can do better.

*6.3.2 The Precise Variant*

Knuth [16] improves the $n^2$ bound to $n^2 - n + 1$ based on this exercise:

Prove that at most one $a$ obtains its last choice with the fundamental algorithm.

Gusfield and Irving [11] argue that this bound follows because "the algorithm terminates when the last $b$ is matched for the first time". We will now give a formal proof that is more in line with Knuth's text and does not require the temporal "first" and "last".

Knuth's exercise amounts to the following proposition: there is at most one unmatched $a$ that is down to its last preference.

**Corollary 1**
$M \subseteq \{<n\} \wedge$ *inj-on* $(match\ A)\ M \wedge (\forall\, a{<}n.\ preferred\ A\ a \subseteq A\langle M\rangle)$
$\longrightarrow (\exists_{\leq 1}\ a.\ a < n \wedge a \notin M \wedge A\ !\ a + 1 = n)$

This is a corollary to the following lemma: if an unmatched $a$ has arrived at the end of its preference list, then all other $a$'s are matched.

**Lemma 7** $M \subseteq \{<n\} \wedge$ *inj-on* $(match\ A)\ M \wedge$ *preferred* $A\ a \subseteq A\langle M\rangle \wedge a < n$
$\wedge\ a \notin M \wedge A\ !\ a + 1 = n \longrightarrow \{a\} \cup M = \{<n\}$

*Proof* From *Pref* it follows that $(!)\ (P\ !\ a)$ ' $\{<n\} = \{<n\}$ and thus $(!)\ (P\ !\ a)$ is injective on $\{<n\}$ and thus in particular on $\{<A\ !\ a\}$ (because $A\ !\ a < n$). Hence $|preferred\ A\ a| + 1 = n$ (1). Assumption $preferred\ A\ a \subseteq A\langle M\rangle$ implies $|preferred\ A\ a| \leq |M|$ (2). From $M \subseteq \{<n\}$ and $a < n \wedge a \notin M$ it follows that $|M| < n$ (3). Combining (1), (2) and (3) yields $|M| + 1 = n$ and thus $\{a\} \cup M = \{<n\}$. □

Now we can prove the key upper bound for $\sum_{a\ <\ n} A\ !\ a$:

**Lemma 8** $matching\ A\ M \wedge M \subset \{<n\} \wedge (\forall\, a{<}n.\ preferred\ A\ a \subseteq A\langle M\rangle)$
$\longrightarrow (\sum_{a\ <\ n} A\ !\ a) \leq (n-1)^2$

*Proof* From Lemma 2 we have $\forall\, a{\in}M.\ A\ !\ a + 1 < n$. We distinguish two cases. If there is an $a' < n$ such that $a' \notin M$ and $A\ !\ a' + 1 = n$ then, because there is at most one such $a'$ (by Corollary 1), it follows that $\forall\, a{<}n.\ a \neq a' \longrightarrow A\ !\ a \leq n - 2$ and thus $(\sum_{a\ <\ n} A\ !\ a) \leq (n-1) * (n-2) + (n-1) = (n-1)^2$. If there is no such $a'$, then $\forall\, a{<}n.\ A\ !\ a + 1 < n$ and thus $(\sum_{a\ <\ n} A\ !\ a) \leq n * (n-2) \leq (n-1)^2$. □

The assumptions of Lemma 8 imply $|M| < n$ and hence

$(\sum_{a\ <\ n} A\ !\ a) + |M| < n^2 - n + 1$

Thus the following definition of *var* makes sense:

$var\ A\ M = n^2 - n + 1 - ((\sum_{a\ <\ n} A\ !\ a) + |M|)$

The following easy properties (except that one has to be careful about subtraction) show that *var* is decreased by all three basic actions. The invariant together with $M \neq \{<n\}$ implies the assumptions of Lemma 8 which imply

$a \notin M \longrightarrow var\ A\ (M \cup \{a\}) < var\ A\ M$
$a < n \longrightarrow var\ (A[a := A\ !\ a + 1])\ M < var\ A\ M$

## 7 Algorithm 2

Algorithm 2 is the result of a data concretisation step: the set $M$ of matched $a$'s is replaced by a list $as$ of unmatched $a$'s. Thus the abstraction function $\alpha$ from lists to sets is $\alpha\ as = \{<n\} - set\ as$. (Note that formally it is only an abstraction function if the *SOME*-choice is nondeterministic.) The program treats the list $as$ like a stack: functions $hd$ and $tl$ return the head and the tail of the stack. In addition to the invariant $invAM\ A\ (\{<n\} - set\ as)$ we also need the well-formedness of $as$:

$invas\ as = (set\ as \subseteq \{<n\} \wedge distinct\ as)$

Thus the complete invariant is $invAM\ A\ (\{<n\} - set\ as) \wedge invas\ as$.

**Algorithm 2**

```
1  [as = [0..<n] ∧ A = replicate n 0]
2  WHILE as ≠ [] INV {invAM A ({<n} − set as) ∧ invas as}
3  VAR {var A ({<n} − set as)}
4  DO a := hd as;
5      b := A⟨a⟩;
6      IF b ∉ A⟨{<n} − set as⟩ THEN as := tl as
7      ELSE a' := (SOME a'. a' ∈ {<n} − set as ∧ A⟨a'⟩ = b);
8           IF Q ! A⟨a'⟩ ⊢ a < a'
9           THEN A := A[a' := A ! a' + 1]; as := a' · tl as
10          ELSE A := A[a := A ! a + 1] FI
11     FI
12 OD
13 [matching A {<n} ∧ stable A {<n} ∧ optiA A]
```

To exemplify our stepwise development approach we consider preservation of the invariant $invAM\ A\ M$ by the basic action **match** (line 6) where $M$ abbreviates $\{<n\} - set\ as$. That is, we assume $invAM\ A\ M \wedge invas\ as$, $as \neq []$, $a = hd\ as$ and $A\langle a\rangle \notin A\langle\{<n\} - set\ as\rangle$. Thus $as = a \cdot as'$ for some $as'$, $M \cup \{a\} = \{<n\} - set\ as'$ and $a < n \wedge a \notin M$ (using $invas\ as$). Now Lemma 4 applies and we conclude $invAM\ A\ (M \cup \{a\})$ which implies $invAM\ A\ (\{<n\} - set\ as')$, which is what we actually need to show. In summary, the translation between the two state spaces requires some bridging properties, but then we can apply the abstract lemmas.

From now on, we do not present correctness lemmas or proofs anymore but only annotated programs because the annotations are the key. Of course we still present all variants, invariants and non-trivial auxiliary definitions.

## 8 Algorithm 3

This data concretisation step addresses the issue that the algorithm needs to find out if the prospective match of some $a$ is already matched and to whom. Algo-

**Algorithm 3**

$[as = [0..<n] \wedge A = replicate\ n\ 0 \wedge B = (\lambda\text{-}.\ None)]$
$WHILE\ as \neq []$
$INV\ \{invAM\ A\ (\{<n\} - set\ as) \wedge invAB\ A\ B\ (\{<n\} - set\ as) \wedge invas\ as\}$
$VAR\ \{var\ A\ (\{<n\} - set\ as)\}$
$DO\ a := hd\ as;$
  $b := A\langle a\rangle;$
  $IF\ B\ b = None\ THEN\ B := B(b \mapsto a);\ as := tl\ as$
  $ELSE\ a' := the\ (B\ b);$
    $IF\ Q\ !\ A\langle a'\rangle \vdash a < a'$
    $THEN\ B := B(b \mapsto a);\ A := A[a' := A\ !\ a' + 1];\ as := a' \cdot tl\ as$
    $ELSE\ A := A[a := A\ !\ a + 1]\ FI$
  $FI$
$OD$
$[matching\ A\ \{<n\} \wedge stable\ A\ \{<n\} \wedge optiA\ A]$

rithm 3 records the inverse of *match* as a variable $B :: nat \Rightarrow nat\ option$. Eventually $B$ will be implemented by arrays. We call a function $m :: 'a \Rightarrow 'b\ option$ a *map*. Maps come with an update notation:

$m(a \mapsto b) = (\lambda x.\ \mathsf{if}\ x = a\ \mathsf{then}\ Some\ b\ \mathsf{else}\ m\ x)$

Function $the :: 'a\ option \Rightarrow 'a$ inverts *Some*: $the\ (Some\ x) = x$. The notation $[m..<n]$ denotes the list $[m, m+1, ..., n-1]$.

The new variable $B$ requires its own invariant:

$invAB\ A\ B\ M = (ran\ B = M \wedge (\forall\ b\ a.\ B\ b = Some\ a \longrightarrow A\langle a\rangle = b))$

where $ran\ m = \{b \mid \exists\ a.\ m\ a = Some\ b\}$. In a nutshell, $B$ is the inverse of *match*. Preservation of $invAB\ A\ B\ (\{<n\} - set\ as)$ by all three basic actions is a one-liner.

## 9 Algorithm 4

In this step we eliminate the list *as*. The resulting algorithm is (more or less) the one that Knuth [16] analyzes. The main idea: with each basic action, either the top of *as* changes or it is popped. Thus we don't need to record all of *as* but only how far we have popped it. The new variable $ai :: nat$ does just that. It starts with 0 and is incremented after each **match** step. This can also be viewed as a data concretisation: $a$ and $ai$ represent $a \cdot [ai + 1..<n]$.

Instead of one we now have two loops. In the inner loop, **swap** and **match** actions are performed, followed by a single **match** action. That is, $a$ is initialized with $ai$ and the inner loop tries to find an unmatched $b$ for $a$, possibly unmatching some $a'$ in the process.

The invariants $invAM$ and $invAB$ are unchanged, and the set $M$ of matched $a$'s is simply $\{<ai\}$. In the inner loop, $M = \{<ai + 1\} - \{a\}$ because we are looking for a match for $a$.

The outer variant is $n - ai$. Note that the syntax permits us to remember that value of the outer variant in an auxiliary variable, here $z$. The point is that we need to show that the outer variant is not incremented by the inner loop. Hence we remember its value in $z$ and add the invariant $z = n - ai$. Although Isabelle's Hoare logic formalization goes back more than 20 years, it was only

**Algorithm 4**

$[ai = 0 \wedge A = replicate\ n\ 0 \wedge B = (\lambda\text{-}.\ None)]$
$WHILE\ ai < n\ INV\ \{invAM\ A\ \{<ai\} \wedge invAB\ A\ B\ \{<ai\} \wedge ai \leq n\}$
$VAR\ \{z = n - ai\}$
$DO\ a := ai;$
  $WHILE\ B\ (A\langle a\rangle) \neq None$
  $INV\ \{invAM\ A\ (\{<ai + 1\} - \{a\}) \wedge invAB\ A\ B\ (\{<ai + 1\} - \{a\})$
    $\wedge\ a \leq ai \wedge ai < n \wedge z = n - ai\}$
  $VAR\ \{var_2\ A\}$
  $DO\ a' := the\ (B\ (A\langle a\rangle));$
    $IF\ Q\ !\ A\langle a'\rangle \vdash a < a'$
    $THEN\ B := B(A\langle a\rangle \mapsto a);\ A := A[a' := A\ !\ a' + 1];\ a := a'$
    $ELSE\ A := A[a := A\ !\ a + 1]\ FI$
  $OD;$
  $B := B(A\langle a\rangle \mapsto a);\ ai := ai + 1$
$OD$
$[matching\ A\ \{<n\} \wedge stable\ A\ \{<n\} \wedge optiA\ A]$

recently extended with variants for total correctness (by Walter Guttmann). In the process of verifying the Gale-Shapely algorithm Nipkow noticed that invariants need to refer to variants and generalized Guttmann's extension. He also noticed that the account in the textbook by De Boer *et al.* [2] (which has not changed from the "first edition" [3]) is incomplete, which the authors confirmed (private communication). Their definition of valid proof outlines (programs annotated with (in)variants) [2, Definition 3.8] does not allow inner invariants to refer to outer variants: replacing $S^{**}$ by $S$ in the side condition for $z$ fixes the problem.

Finally we consider the inner variant:

$$var_2\ A = (n - 1)^2 - (\sum_{a\ <\ n} A\ !\ a)$$

To show that $var_2\ A$ decreases when $A\ !\ a$ (or $A\ !\ a'$) are incremented we consider one loop iteration with initial value $A$ and final value $A'$. Note that because the invariant again holds for $A'$, Lemma 8 implies that $(\sum_{a\ <\ n} A'\ !\ a) \leq (n - 1)^2$.

$$\begin{aligned}
&var_2\ A' = (n - 1)^2 - (\textstyle\sum_{a\ <\ n} A'\ !\ a)\\
&= (n - 1)^2 - ((\textstyle\sum_{a\ <\ n} A\ !\ a) + 1)\\
&< (n - 1)^2 - (\textstyle\sum_{a\ <\ n} A\ !\ a) \qquad \text{because } (\textstyle\sum_{a\ <\ n} A'\ !\ a) \leq (n - 1)^2\\
&= var_2\ A
\end{aligned}$$

The initial value of $var_2\ A$ is $(n - 1)^2$. Because the outer loop does not modify $A$, $(n - 1)^2$ is an upper bound on the total number of iterations of the inner loop, i.e. the number of **swap** and **next** actions. To this we must add the exactly $n$ **match** actions of the outer loop to arrive at an upper bound of $n^2 - n + 1$ actions, just like before.

## 10 Algorithm 5

In this step we implement $B :: nat \Rightarrow nat\ option$ by two lists (think arrays): $N :: bool\ list$ records which $b$'s have been matched with some $a$ and $B :: nat\ list$ says which $a$. This is expressed by the following abstraction function:

$\alpha\ B\ N = (\lambda b.\ \mathit{if}\ b < n \wedge N\ !\ b\ \mathit{then}\ \mathit{Some}\ (B\ !\ b)\ \mathit{else}\ \mathit{None})$

**Algorithm 5**

$[ai = 0 \wedge A = replicate\ n\ 0 \wedge |B| = n \wedge N = replicate\ n\ False]$
$WHILE\ ai < n\ INV\ \{invar1\ A\ B\ N\ ai\}\ \ VAR\ \{z = n - ai\}$
$DO\ a := ai;$
  $WHILE\ N\ !\ A\langle a\rangle\ INV\ \{invar2\ A\ B\ N\ ai\ a \wedge z = n - ai\}\ \ VAR\ \{var_2\ A\}$
  $DO\ a' := B\ !\ A\langle a\rangle;$
    $IF\ Q\ !\ A\langle a'\rangle \vdash a < a'$
    $THEN\ B := B[A\langle a\rangle := a];\ A := A[a' := A\ !\ a' + 1];\ a := a'$
    $ELSE\ A := A[a := A\ !\ a + 1]\ FI$
  $OD;$
  $B := B[A\langle a\rangle := a];\ N := N[A\langle a\rangle := True];\ ai := ai + 1$
$OD$
$[matching\ A\ \{<n\} \wedge stable\ A\ \{<n\} \wedge optiA\ A]$

At this point it is helpful to introduce names for the two invariants:

$invar1\ A\ B\ N\ ai =$
$(invAM\ A\ \{<ai\} \wedge invAB\ A\ (\alpha\ B\ N)\ \{<ai\} \wedge |B| = n \wedge |N| = n \wedge ai \leq n)$
$invar2\ A\ B\ N\ ai\ a =$
$(invAM\ A\ (\{<ai + 1\} - \{a\}) \wedge invAB\ A\ (\alpha\ B\ N)\ (\{<ai + 1\} - \{a\})$
$\wedge |B| = n \wedge |N| = n \wedge a \leq ai \wedge ai < n)$

## 11 Algorithm 6

In this step we implement the inefficient test $Q\ !\ A\langle a'\rangle \vdash a < a'$. Instead of finding the index of $a$ and $a'$ in the list $Q\ !\ A\langle a'\rangle$ we replace $Q$ by a list of lists that map $a$'s to their index, i.e. their rank in the preference lists. From $Q$ we construct the new data structure $R :: nat\ list\ list$ as $R = map\ ranking\ Q$ where

$ranking\ P = rk\text{-}of\text{-}pref\ 0\ (replicate\ |P|\ 0)\ P$

$rk\text{-}of\text{-}pref\ r\ rs\ (n \cdot ns) = (rk\text{-}of\text{-}pref\ (r + 1)\ rs\ ns)[n := r]$
$rk\text{-}of\text{-}pref\ r\ rs\ [] = rs$

If the list update operation is constant-time (which it will be with arrays), *ranking* is a linear-time algorithm and thus $R$ can be computed in time $O(n^2)$. A more intuitive but less efficient definition of *ranking* is

$ranking\ P = map\ (index\ P)\ [0..<|P|]$

The two definitions coincide if $set\ P = \{<|P|\}$.

In Algorithm 6, the only operations used are arithmetic, list indexing (!) and pointwise list update $xs[m := x]$. If we implement the latter with constant-time operations on arrays (as we will, in the next section), each assignment and each test takes constant time. Thus the overall execution time of the algorithm is proportional to the number of executed tests and assignments. Clearly the outer loop, without the inner one, takes time $O(n)$. As analyzed in the previous section, the inner loop body is executed at most $(n - 1)^2$ times. Thus the overall complexity

**Algorithm 6**

```
R = map ranking Q ⟶
[ai = 0 ∧ A = replicate n 0 ∧ |B| = n ∧ N = replicate n False]
WHILE ai < n INV {invar1 A B N ai}  VAR {z = n − ai}
DO a := ai;
    b := A⟨a⟩;
    WHILE N ! b INV {invar2 A B N ai a ∧ b = A⟨a⟩ ∧ z = n − ai}
    VAR {var2 A}
    DO a' := B ! b;
        r := R ! A⟨a'⟩;
        IF r ! a < r ! a'
        THEN B := B[b := a]; A := A[a' := A ! a' + 1]; a := a'
        ELSE A := A[a := A ! a + 1] FI;
        b := A⟨a⟩
    OD;
    B := B[b := a]; N := N[b := True]; ai := ai + 1
OD
[matching A {<n} ∧ stable A {<n} ∧ optiA A]
```

of the algorithm is $O(n) + O(n^2) = O(n^2)$. Because the input, $P$ and $R$, is also of size $O(n^2)$ we have a linear-time algorithm.

## 12 Algorithm 7

In a final step (on the imperative level) we implement lists by arrays. The basis is Lammich's and Lochbihler's Collections library [17,19] that offers imperative implementations of arrays with a purely functional, list-like interface specification. The basic idea is due to Baker [4,5] and guarantees constant time access to arrays provided they are used in a linear manner (i.e. no access to old versions), which our arrays obviously are, because the programming language is imperative.

Algorithm 7 is not displayed because it is Algorithm 6 with *xs* ! *i* replaced by *array-get xs i*, *xs*[*i* := *x*] replaced by *array-set xs i x* and *replicate n x* replaced by *new-array x n*, where *array-get* (below: *xs* !! *i*), *array-set* (below: *xs*[*i* ::= *x*]) and *new-array* (below: *array x n*) are defined in the Collections library. Correctness of this data concretisation step is proved via the abstraction function *list* :: *'a array* ⇒ *'a list* and refinement lemmas like *list* (*a*[*i* ::= *x*]) = (*list a*)[*i* := *x*].

This is our final imperative algorithm. It has linear complexity, as explained in the previous section. Although the programming language has a semantics that can in principle be executed, Isabelle provides no support for that. Therefore we now recast the last imperative algorithm as recursive functions in Isabelle's logic, which can be executed in Isabelle [1] or exported to standard functional languages [12].

## 13 Algorithm 8, Functional Implementation

We translate the imperative code directly into two functions *gs* and *gs-inner* (Algorithm 8) using the combinator

*while* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a*) ⇒ *'a* ⇒ *'a*

**Algorithm 8** Functional Implementation

```
gs n P R =
while (λ(A, B, N, ai). ai < n)
 (λ(A, B, N, ai).
    let (A, B, a, b) = gs-inner P R N (A, B, ai, P !! ai !! (A !! ai))
    in (A, B[b ::= a], N[b ::= True], ai + 1))
 (array 0 n, array 0 n, array False n, 0)

gs-inner P R N =
while (λ(A, B, a, b). N !! b)
 (λ(A, B, a, b).
    let a' = B !! b; r = R !! (P !! a' !! (A !! a'));
        (A, B, a) =
          if r !! a < r !! a' then (A[a' ::= A !! a' + 1], B[b ::= a], a')
          else (A[a ::= A !! a + 1], B, a)
    in (A, B, a, P !! a !! (A !! a)))
```

from the Isabelle library [22]. It comes with the recursion equation

*while b c s* = (*if b s then while b c* (*c s*) *else s*)

for execution and a Hoare-like proof rule (not shown) for total correctness involving a wellfounded relation on the state space. With its help and the lemmas used in the proof of Algorithm 7 we can show the main correctness theorem: *gs* computes a stable matching that is A-optimal:

*gs n* (*pref-array P*) (*rank-array Q*) = (*A*, *B*, *N*, *ai*)
$\longrightarrow$ *matching* (*list A*) {<*n*} $\wedge$ *stable* (*list A*) {<*n*} $\wedge$ *optiA* (*list A*)

where *pref-array* converts $P$ from lists to arrays and *rank-array* converts $P$ into $R$ in array-form, i.e. *rank-array* behaves like *ranking*, but on arrays. Both *pref-array* and *rank-array* (which are straightforward and not shown) are linear-time functions. Thus the conversion from lists to arrays does not influence the time and space complexity of the algorithm. The complexity is still $O(n^2)$ because all basic operations are constant-time and the wellfounded relations used in the total-correctness proofs are defined directly in terms of the variants of the imperative Algorithms 6 (and hence 7).

So far we have worked in the context of the assumptions *Pref* on $P$ and $Q$ stated at the beginning of Section 5. In a final step, to obtain unconditional code equations for the implementation, we move out of that context. The top-level Gale-Shapley function checks well-formedness of the input explicitly by calling predicate *Pref* before calling *gs*:

```
Gale-Shapley P Q =
(if Pref P Q then Some (fst (gs |P| (pref-array P) (rank-array Q)))
 else None)
```

where *fst* selects the first component of a tuple. Function *Pref* is executable but not linear-time because it operates on lists. It would be simple to convert it to a linear-time function on arrays, but because it is just boiler-plate and not part of the actual Gale-Shapley algorithm we ignore that.

The correctness theorem for *Gale-Shapley* follows directly from the one for *gs*:

$$\begin{aligned}&\mathit{Pref}\ P\ Q \wedge n = |P| \\ \longrightarrow\ &(\exists A.\ \mathit{Gale\text{-}Shapley}\ P\ Q = \mathit{Some}\ A \wedge \mathit{matching}\ P\ (\mathit{list}\ A)\ \{<n\} \\ &\quad \wedge\ \mathit{stable}\ P\ Q\ (\mathit{list}\ A)\ \{<n\} \wedge \mathit{optiA}\ P\ Q\ (\mathit{list}\ A))\end{aligned}$$

## 14 Related Work

Most proofs about stable matching algorithms, starting with Gale and Shapley, omit formal treatments of the requisite assertions. However, there are noteworthy exceptions.

Hamid and Castleberry [13] were the first to subject the Gale-Shapley algorithm to a proof assistant treatment (in Coq). They present an implementation (and termination proof) of the Gale-Shapley algorithm and an executable checker for stability but no proof that the algorithm always returns a stable matching. They do not comment on the complexity of their algorithm, but it is not linear, not just because they do not refine it down to arrays, but also because of other inefficiencies. Nor do they consider optimality.

Gammie [9] mechanizes (in Isabelle) proofs of several results from the matching-with-contracts literature, which generalize those of the classical stable marriage scenarios. Along the way he also develops executable algorithms for computing optimal stable matches. The complexity of these algorithms is not analyzed (and not clear even to the author, but not linear). The focus is on game theoretic issues, not algorithm development.

Probably the first reasonably precise analysis of the algorithm is by Knuth [15, 16, Lecture 2]. His starting point is akin to our Algorithm 4, except that at this point he is not precise about the representation of data structures and the operations on them. Moreover, his assertions are a mixture of purely state-based ones and temporal ones (e.g. "has rejected") and the proof is not expressed in some fixed program logic. In a later chapter [15, 16, Lecture 6] he shows an array-based implementation and relates it informally to the algorithm from Lecture 2.

Bijlsma [6], in Dijkstra's tradition and notation [7], derives in a completely formal (but not machine-checked) manner an algorithm very close to our Algorithm 7. The main difference is that he starts from a specification and we start from an algorithm. Thus his and our development steps are largely incomparable. He does not consider optimality.

## 15 Conclusion and Further Work

We have seen a step by step development of an efficient implementation of the Gale-Shapley algorithm. It is desirable to cover more of the algorithmic content of the stable matching area. A good starting point are the further problems covered by Gusfield and Irving [11], e.g. the hospitals/residents problem (where $m$ doctors are matched with $n$ hospitals with a fixed capacity) and the stable roommates problem (where $2n$ people are matched with one another into pairs). A second avenue for further work is the development of efficient code from the abstract fixpoint-based algorithm for matching-with-contracts that was formalized by Gammie [9].

**Acknowledgements** I want to thank Reviewer 1 for very perceptive comments, in particular a simplification of Lemma 2, and Katharina Kreuzer for proofreading.

## References

1. Aehlig, K., Haftmann, F., Nipkow, T.: A compiled implementation of normalization by evaluation. Journal of Functional Programming **22**(1), 9–30 (2012)
2. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer (2009). URL `https://doi.org/10.1007/978-1-84882-745-5`
3. Apt, K.R., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts and Monographs in Computer Science. Springer (1991). URL `https://doi.org/10.1007/978-1-4757-4376-0`
4. Baker, H.G.: Shallow binding in LISP 1.5. Commun. ACM **21**(7), 565–569 (1978). URL `https://doi.org/10.1145/359545.359566`
5. Baker, H.G.: Shallow binding makes functional arrays fast. ACM SIGPLAN Notices **26**(8), 145–147 (1991). URL `https://doi.org/10.1145/122598.122614`
6. Bijlsma, A.: Formal derivation of a stable marriage algorithm. In: W. Feijen, A. van Gastreren (eds.) C.S. Scholten dedicata: van oude machines en nieuwe rekenwijzen. Academic Service Schoonhoven (1991). URL `https://dspace.library.uu.nl/handle/1874/19385`
7. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
8. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. American Mathematical Monthly **69**(1), 9–15 (1962). URL `https://doi.org/10.2307/2312726`
9. Gammie, P.: Stable matching. Archive of Formal Proofs (2016). `https://isa-afp.org/entries/Stable_Matching.html`, Formal proof development
10. Gordon, M.C.: Mechanizing programming logics in higher order logic. In: G. Birtwistle, P. Subrahmanyam (eds.) Current Trends in Hardware Verification and Automated Theorem Proving. Springer (1989)
11. Gusfield, D., Irving, R.W.: The Stable marriage problem - structure and algorithms. Foundations of computing series. MIT Press (1989)
12. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: M. Blume, N. Kobayashi, G. Vidal (eds.) Functional and Logic Programming, 10th International Symposium, FLOPS 2010, *Lecture Notes in Computer Science*, vol. 6009, pp. 103–117. Springer (2010). URL `https://doi.org/10.1007/978-3-642-12251-4_9`
13. Hamid, N.A., Castleberry, C.: Formally certified stable marriages. In: Proceedings of the 48th Annual Southeast Regional Conference, ACM SE '10. ACM (2010). URL `https://doi.org/10.1145/1900008.1900056`
14. Kleinberg, J., Tardos, E.: Algorithm Design. Addison-Wesley (2006)
15. Knuth, D.E.: Mariages Stables et leurs relations avec d'autres problèmes combinatoires. Les Presses de l'Université de Montréal (1976)
16. Knuth, D.E.: Stable Marriage and its Relation to Other Combinatorial Problems. American Mathematical Society (1997). Translation of [15]
17. Lammich, P.: Collections framework. Archive of Formal Proofs (2009). `https://isa-afp.org/entries/Collections.html`, Formal proof development
18. Lammich, P.: Refinement to imperative/hol. In: C. Urban, X. Zhang (eds.) Interactive Theorem Proving - 6th International Conference, ITP 2015, *LNCS*, vol. 9236, pp. 253–269. Springer (2015). URL `https://doi.org/10.1007/978-3-319-22102-1_17`
19. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: M. Kaufmann, L.C. Paulson (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, *LNCS*, vol. 6172, pp. 339–354. Springer (2010). URL `https://doi.org/10.1007/978-3-642-14052-5_24`
20. Maggs, B.M., Sitaraman, R.K.: Algorithmic nuggets in content delivery. SIGCOMM Comput. Commun. Rev. **45**(3), 52–66 (2015). URL `https://doi.org/10.1145/2805789.2805800`
21. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: F. Baader (ed.) Automated Deduction — CADE-19, *LNCS*, vol. 2741, pp. 121–135. Springer (2003)
22. Nipkow, T.: Verified efficient enumeration of plane graphs modulo isomorphism. In: M.C.J.D. van Eekelen, H. Geuvers, J. Schmaltz, F. Wiedijk (eds.) Interactive Theorem Proving, ITP 2011, *Lecture Notes in Computer Science*, vol. 6898, pp. 281–296. Springer (2011). URL `https://doi.org/10.1007/978-3-642-22863-6_21`

23. Nipkow, T.: Gale-Shapley algorithm. Archive of Formal Proofs (2021). https://isa-afp.org/entries/Gale_Shapley.html, Formal proof development
24. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014). http://concrete-semantics.org
25. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
26. Wenzel, M.: Isar — a generic interpretative approach to readable formal proof documents. In: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Thery (eds.) Theorem Proving in Higher Order Logics, TPHOLs'99, *LNCS*, vol. 1690, pp. 167–183. Springer (1999)
27. Wenzel, M.: Isabelle/isar — a versatile environment for human-readable formal proof documents. Ph.D. thesis, Institut für Informatik, Technische Universität München (2002). Http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss2002020117092