# Part 1

—

# An Implementation of the Gale-Shapley Algorithm

CSCI 3110 Code

Fall 2015

## 1 Introduction

Here's the pseudo-code of the Gale-Shapley algorithm from class:

```
GaleShapley($M, W$)
  while there is an unmarried man $m$ do
    $m$ chooses the first woman $w$ on his preference list he has not proposed to yet and proposes to her
    if $w$ is unmarried or prefers $m$ over her current partner $m'$ then
      $w$ divorces $m'$
      $w$ marries $m$
```

Here we develop a full implementation of this algorithm.

## 2 The Algorithm

The input to the algorithm is represented as an input size $n$, a list of men, along with their preference lists, and a list of women, along with their preference lists:

```
data Instance = Instance {n     :: Int
                         ,men   :: [Man]
                         ,women :: [Woman]
                         }
```

Men and women are both two types of *Person*s. Every *Person* has a *name* of some type $a$ and is eligible to get married to *Person*s with names of some other type $b$. The ranking of potential partners is given as a simple list of type $[b]$:

```
data Person a b = Person {name     :: a
                         ,prefList :: [b]
                         }
```

Men are *Person*s with names of type *M* and preferring women, whose names are of type *W*. For women, the roles of *M* and *W* are reversed:

```
type Man   = Person M W
type Woman = Person W M
```

In our implementation, we assume the names of men and women are just integers. However, to allow the Haskell compiler to check that we're not accidentally adding a man to a man's preference list or a woman to a woman's preference list, we wrap these integers into types *M* and *W* that inherit the *Int* type's notions of equality, ordering, and ranges but are considered distinct types by the compiler:

```
newtype M = M Int deriving (Eq, Ord, Ix)
newtype W = W Int deriving (Eq, Ord, Ix)
```

This ensures that the compiler complains whenever we try to use a *W* where an *M* is expected and vice versa, but the internal representation of each type is just a plain integer.[1]

Now, the main function, *stableMatching*, is to produce a stable *Matching* from a given input *Instance*, where a *Matching* is just a list of *Marriage*s and a *Marriage* is simply a pair of a man and a woman (their names suffice):

```
stableMatching :: Instance → Matching

type Marriage = (M, W)
type Matching = [Marriage]
```

As we discussed in class, the ordered sequence of women in the preference list of each *Man* is perfectly adequate to allow the *Man* to propose to the women on his preference list in order. In order to decide whether a woman should accept or reject a proposal, she needs to know here current partner and she needs to have a constant-time ranking function that returns the position of a given man in her preference list. As discussed in class, this function can be implemented using a simple lookup in an inverted preference list, implemented as an *Array*. We pack this information about each *Woman* into a data structure of type *State* and store the states of all women in a state *Array* indexed by the names of all women:

```
data State  = State {partner :: Maybe Man
                    ,rank    :: Man → Int
                    }
```

[1]Haskell gives us three keywords to define new types: **data**, **newtype**, and **type**. **type** is similar to C's typedef; **type** $t_1 = t_2$ introduces $t_1$ as a type synonym for $t_2$. Either can be used where the other is expected. **data** defines a new data type similar to C's struct. The type is different from all other types and it has a distinct runtime representation. **newtype** is somewhere in the middle. We are only allowed to define structs with exactly one field using **newtype**. To the compiler the type is distinct from all other types as if we had defined it using **data**. The runtime representation of a type $t_1$ defined using **newtype** $t_1 = T\ t_2$ is exactly the same as that of $t_2$. Thus, **newtype** can be used help the type checker help us ensure write correct programs but without the runtime overhead of a **data** declaration.

```
type States = Array W State
```

A woman's partner is of type *Maybe Man*, that is, it can have values *Nothing*, if she does not have a partner, or *Just m*, if *m* is her partner. To initialize the state of the algorithm, we can now populate the *States* array with *State* records that store *Nothing* as their *partner*s and whose rank functions are constructed by calling a function *makeRank* on each *Woman*'s preference list:

```
initState      :: Instance → States
initState inst = array (W 1, W (n inst))                                   ∘
                   map (λw → (name w, State Nothing (makeRank (n inst) (prefList w)))) $
                   women inst
```

*makeRank* constructs an inverted preference list from a given preference list by combining every entry in the preference list with its position in the preference list: *zip* (*prefList w*) [1 . .]. Then we construct the inverted preference list, an array with index range (*M* 1,*M n*), by storing *j* in position *M i* of the array, for every pair (*M i,j*) in the resulting list. The function returned by *makeRank* is then simply implemented as a lookup in this inverted preference list:

```
makeRank      :: Int → [M] → Man → Int
makeRank n ms = rank
  where rank m      = rankArray ! name m
        rankArray = array (M 1, M n) (zip ms [1 . .])
```

The core of our algorithm is a function *proposals*, which turns the initial *States* constructed by *initState* into the final *States* where every *State* stores the corresponding woman's *partner* in the final solution, using the proposals made by a sequence of men:

```
proposals :: [Man] → States → States
```

Given the final *States*, we need to extract the final *Matching* from these *States* using the following function:

```
getMatching :: States → Matching
```

Given these three functions, we can now specify our *stableMatching* function as:

```
stableMatching inst = getMatching (proposals (men inst) (initState inst))
```

The *getMatching* function is easy enough to implement. All we have to do is extract the partner of every woman and produce a list of these pairs. *assocs* applied to an *Array* returns the list of (*index*, *value*) pairs of the *Array*. In our case, the indices are of type *W* and the values are of type *State*. To obtain a valid *Marriage* from such a pair of type (*W*, *State*), we need to extract the *name* of the *partner* stored in the *State* record and reverse the order of *Man* and *Woman* because, in our *Marriage* type, the *Man*'s name comes first. This gives the following implementation:

$$getMatching = map\ (\lambda(w,s) \to (name \circ fromJust \circ partner\ \$\ s, w)) \circ assocs$$

where

$$\begin{aligned}
&fromJust && :: Maybe\ a \to a \\
&fromJust\ Nothing && = error\ \texttt{"Undefined for Nothing"} \\
&fromJust\ (Just\ x) && = x
\end{aligned}$$

is a pretty dangerous function in general because it may crash our program. Here, it is fine to use it because we know that at the end of the algorithm, the *partner* of every woman is *Just* a man.

The *proposals* function is where we have to leave the realm of pure functions and dive into using our strict state monad *ST s*: Our algorithm needs to update the partners of women as they change after proposals are made. Since we also want constant-time access to the woman the current man proposes to, we need to store the states of the women in an array and purely functional arrays are immutable, so each update would take $O(n)$ time. Our strategy is to turn our *States* array into a mutable array, whose array cells can be updated in constant time. We start by defining a type synonym similar to *States* to refer to our mutable state within the stateful computation:

$$\textbf{type}\ MStates\ s = STArray\ s\ W\ State$$

The *proposals* function now runs a stateful computation *proposalsM* wrapped in a pair of *thaw*/*freeze* calls to convert the initial state into a mutable state and the final state back into an immutable state:

$$\begin{aligned}
proposals\ ms\ st = runST\ \$\ \textbf{do}\ & mst \leftarrow thaw\ st \\
& proposalsM\ mst\ ms\ Nothing \\
& freeze\ mst
\end{aligned}$$

$$proposalsM :: MStates\ s \to [Man] \to Maybe\ Man \to ST\ s\ ()$$

The arguments to *proposalsM* are the current states of the women, a list of unmarried men who are currently waiting to propose, and *Maybe* a *Man* who is unmarried and has been selected to make the next proposal. Initially, we haven't chosen any *Man* to make the next proposal, so we pass all unmarried men in as the original list *ms* and set the currently proposing man to *Nothing*. Now, how should *proposalsM* behave?

If there is no unmarried *Man*, then it should exit without any further modifications to the state; all men are married, we are done.

If we have selected a *Man* to make the next proposal (the third argument is *Just m*), we should let him make his next proposal. This proposal itself is implemented by a *proposeM* computation:

$$proposeM :: MStates\ s \to Man \to ST\ s\ (Maybe\ Man)$$

The return value of type *Maybe Man* is used to return *Maybe* a *Man* who is unmarried as a result of this proposal. Thus, if *m*'s proposal is accepted and the woman wasn't married before, *m* is now married and no new unmarried man has been created; the computation returns *Nothing*. If *m*'s proposal is rejected, then *m* is still unmarried; we return *Just m*. Finally, if *m*'s proposal is accepted but the woman was married to a man $m'$ before, $m'$ is now unmarried, so we return *Just* $m'$. To continue the proposal process, *proposalsM* takes this return value and calls itself recursively on the current state, the current list of unmarried men, and the return value of *proposeM*.

Finally, if there are unmarried men but none has been picked for the next proposal, we should pick the next unmarried *Man* from our list and make him the next *Man* to propose. These three cases translate into the following computation:

```
proposalsM _  [ ]      Nothing  = return ()
proposalsM st (m : ms) Nothing  = proposalsM st ms (Just m)
proposalsM st ms       (Just m) = proposeM st m ≫= proposalsM st ms
```

Before describing the *proposeM* computation, let us specify the logic of a proposal, which can be expressed as a pure function:

```
propose :: Man → State → (Maybe Man, State)
propose m w | w `likesBetter` m = (partner w, w {partner = m′})
            | otherwise         = (m′, w)
  where m′ = Just $ m {prefList = tail (prefList m)}
```

This code is pretty self-explanatory, but let's discuss it anyway. Given a *Man m* and a woman with *State w*, a proposal by *m* results in an update of *w* and *Maybe* a new unmarried *Man*. Hence, the inputs of the function are *m* and *w* and the result is a pair of type (*Maybe Man*, *State*). We make use of a predicate *w ‘likesBetter‘ m*, which is *True* if *w* is unmarried or likes *m* better than her current *partner*. If *w ‘likesBetter‘ m*, then her current *partner*, who may be *Nothing*, becomes unmarried, and *w*'s new *partner* becomes *m* with *w* removed from his preference list because he has just proposed to her. This updated *m*, wrapped in *Just*, is the $m'$ constructed in the **where** clause. If *w* does not like *m* better than her current *partner*, then *w* does not change and *m* remains unmarried, so we return the pair $(m', w)$ because we should still remove *w* from *m*'s preference list as *m* has just proposed to her and should not propose to her again.

The predicate *w ‘likesBetter‘ m* should return *True* if *w*'s current *partner* is *Nothing* (*w* is unmarried) or *m*'s *rank* in *w*'s preference list is less than her current *partner*'s *rank*:

```
likesBetter     :: State → Man → Bool
likesBetter w m = maybe True
                        (λp → rank w m < rank w p)
                        (partner w)
```

Finally, to implement a proposal by the currently active *Man m*, all we have to do is to take the first *Woman* on his preference list, lookup her state in the current *MStates* array, pass the *Man* and the woman's state to our *propose* function, write the updated state back to the *MStates* array, and return the *Maybe* unmarried *Man* the *proposal* produced. This can be expressed in terms of a *modifyArray* computation that takes care of reading and writing the *MStates* array:

```
proposeM st m = modifyArray st (head ∘ prefList $ m) (propose m)

modifyArray       :: Ix a ⇒ STArray s a b → a → (b → (c, b)) → ST s c
modifyArray a i f = do (r, x) ← f <$> readArray a i
                       writeArray a i x
                       return r
```

And that’s it. The algorithm is finished.