Matching DSL
DSL Report Card

Prashant Kumar
School of EECS
Oregon State University

June 12, 2020

1 Introduction

Matching individuals to an "appropriate" choices has various applications:

- Matching medical residents to residency programs at various hospitals (NRMP programs in US and Canada).
- School Allocation in New Orleans.
- Stable Marriages.
- Matching room mates in dormitories.

There are a few libraries that provide a way to specify the matching problem. However, the problem specification requires technical understanding of the algorithmic aspects of the problem and is not declarative enough.

2 Users

We envisage the end user to be a person who is a domain expert and doesn't know the underlying technical underpinnings of the algorithm used for matching. However, since the DSL is a shallow one, the user should be well versed with the Haskell programming language and the type class mechanism. This might be a tall ask for the end users as the domain experts may not have such an advanced understanding of the Haskell which may hinder their ability to express themselves with ease in our DSL. Another potential user group we envisage is the Haskell programmer group, who may not understand the intricacies of the matching algorithm but might still need to use it. For this group, our solution is an ideal one.

3 Outcomes

Executing our matching DSL on a source and target sets should result in an association of every element of source to zero or more elements of the target set. This association is based on the choices of the individual elements of the source and the target set specified by the user. In addition to the association, we also plan to generate an "explanation" behind the rationale for the association.

Some useful static checks can be made inside the DSL. For example, the association of every element of the source set to every element of the target set has to be present. The absence of the association has to be mentioned explicitly. The DSL can statically ensure that none of these associations are missing and can raise appropriate warning if that is not the case.

4 Use Cases / Scenarios

A program written in the matching DSL is shown below. This program finds the stable marriage pairs between a group of men and women.

```haskell
data Man = Bob | Ben | Jack deriving (Eq,Show)

data Woman = Alice | Jane | Jill | Eli | Ann deriving (Eq,Show)

instance MatchSet Man where
    members = [Bob,Ben,Jack]

instance MatchSet Woman where
    members = [Alice,Jane,Jill,Eli,Ann]

-- ==============================================================================================

-- culturalSimi => cultural similarity,ageCompt => age comptability
data MChoice = M {culturalSimi::Level,ageCompt::Rating,mlooks :: Rating} deriving Show

-- politicsSimi => similarity of politics,intcompt => intellectual compatibiltiy
data WChoice = W {politicsSimi :: Bool,intcompt :: Rating,wlooks::Rating} deriving Show

-- ==============================================================================================

mch :: Level -> Rating -> Rating -> Maybe MChoice
mch l a m = return $ M {culturalSimi = l, ageCompt = a, mlooks = m}

mensCh = expressCh [Alice,Jane,Jill,Eli,Ann]

bobsChoice = mensCh [mch High 10 10,Nothing,mch High 8 10,mch VHigh 9 8, mch Med 10 10]
bensChoice = mensCh [mch High 8 6,mch VHigh 10 10,mch High 6 8,mch High 4 6,mch Med 7 6]
jacksChoice = mensCh [mch Low 7 8,mch Low 2 3,mch Med 6 8,mch VHigh 9 8,mch Med 9 10]

instance Relate Man Woman MChoice  where
    assignVal = \case {Bob -> bobsChoice;Ben -> bensChoice;Jack -> jacksChoice}

-- ==============================================================================================

wch :: Bool -> Rating -> Rating -> Maybe WChoice
wch b i w = return $ W {politicsSimi=b,intcompt=i,wlooks=w}

womensCh = expressCh [Bob,Ben,Jack]

alicesChoice= womensCh [wch False 7 7,wch True 10 10,wch True 8 7]
janesChoice = womensCh [wch True 10 7,wch True 10 10,wch True 8 7]
jillsChoice = womensCh [wch True 8 10,wch True 9 9,wch False 10 9]
```

```
elisChoice = womensCh [wch True 7 10,wch False 10 4,wch True 4 9]
annsChoice = womensCh [wch True 8 9, wch True 7 9,wch False 9 9]

instance Relate Woman Man WChoice  where
    assignVal = \case {Alice -> alicesChoice;Jane -> janesChoice;Jill -> jillsChoice;Eli -> elisChoice;Ann -> annsC

-- ================================================================================================

-- take a choice made by a man and convert it to a number using linear MADM
instance Evaluable MChoice where
    eval (M x y z) = madm [0.2,0.4,0.4] [evalL x,evalR y,evalR z]

instance Evaluable WChoice where
    eval (W x y z) = madm [0.3,0.3,0.4] [evalB x,evalR y,evalR z]


instance StableMarriage Man Woman MChoice WChoice
instance StableMarriage Woman Man WChoice MChoice
```

The output of the program written in our matching DSL is shown below:

```
> showMatch(solveP :: Match Man Woman)
Jack:
                 Matched with [Ann]
                 Remaining capacity: 0

Ben:
                 Matched with [Jane]
                 Remaining capacity: 0

Bob:
                 Matched with [Alice]
                 Remaining capacity: 0

> showMatch(solveP :: Match Woman Man)
Eli:
                 Matched with []
                 Remaining capacity: 1

Ann:
                 Matched with [Jack]
                 Remaining capacity: 0

Alice:
                 Matched with [Bob]
                 Remaining capacity: 0

Jill:
                 Matched with []
                 Remaining capacity: 1

Jane:
                 Matched with [Ben]
                 Remaining capacity: 0
```

Figure 1: Example: NRMP Match

NRMP is used to match doctors to their residencies. An example of an NRMP problem is shown in Figure 1. The encoding of this problem in our DSL is shown below.

```
data Candidate = Arthur | Sunny | Joseph | Latha | Darrius deriving (Eq,Show)
data Hospital = City | Mercy | General deriving (Eq,Show)

instance MatchSet Candidate where
    members = [Arthur,Sunny,Joseph,Latha,Darrius]

instance MatchSet Hospital where
    members = [City,Mercy,General]
    capacity _ = 2

data CChoice = C {lctnPref::Rating,salary::Double,clgIntrst ::Level,specReput::Level}
data HChoice = H {examScore::Int, intervPerf::Level, prevRelExpr::Bool}

-- ==========================================================================================
cch :: Rating -> Double -> Level -> Level -> Maybe CChoice
cch l s c r = return $ C {lctnPref=l,salary=s,clgIntrst=c,specReput=r}

candidateCh = expressCh [City,Mercy,General]

arthursChoice = candidateCh [cch 5 90000 High VHigh,Nothing,Nothing]
sunnysChoice  = candidateCh [cch 6 90000 High Med,cch 6 80000 Med High,Nothing]
josephsChoice = candidateCh [cch 8 90000 High VHigh,cch 2 100000 Low Low,cch 5 90000 High High]
lathasChoice  = candidateCh [cch 7 100000 High VHigh,cch 8 120000 VHigh VHigh,cch 6 95000 High VHigh]
darriusChoice = candidateCh [cch 7 110000 VHigh High,cch 5 100000 High Med,cch 6 90000 VLow Med]


josephsChoice :: Hospital -> Maybe CChoice
josephsChoice = \case Mercy   -> return $ C {lctnPref = 2,salary = 100000,clgIntrst = Low,specReput = Low}
                      City    -> return $ C {lctnPref = 8,salary = 90000,clgIntrst = High,specReput = VHigh}
                      General -> return $ C {lctnPref = 5,salary = 90000,clgIntrst = High,specReput = High}

instance Relate Candidate Hospital CChoice where
    assignVal = \case {Arthur -> arthursChoice;Sunny -> sunnysChoice;Joseph -> josephsChoice;
                       Latha  -> lathasChoice;Darrius -> darriusChoice}

-- ==========================================================================================
```

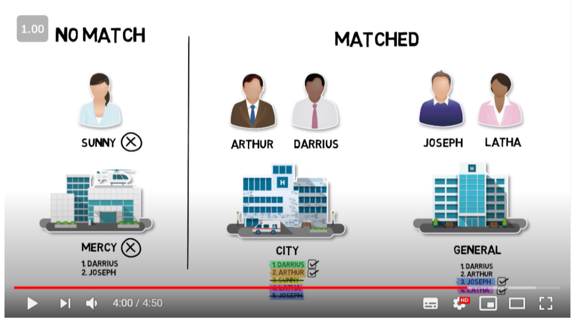

Figure 2: Stable NRMP Match

```
hch :: Int -> Level -> Bool -> Maybe HChoice
hch e i p = return $ H {examScore = e,intervPerf=i,prevRelExpr=p}

hospitalCh = expressCh [Arthur,Sunny,Joseph,Latha,Darrius]

mercysChoice = hospitalCh [Nothing,Nothing,hch 700 Med True,Nothing,hch 770 High True]



citysChoice = hospitalCh [hch 790 Med True,hch 750 Med True,hch 690 Low False,hch 750 Low True,hch 800 VHigh True]
generalsChoice = hospitalCh [hch 790 Med True,Nothing,hch 780 VLow False,hch 690 VLow False,hch 770 VHigh True]



instance Relate Hospital Candidate HChoice where
    assignVal = \case {Mercy -> mercysChoice; City -> citysChoice;General -> generalsChoice}

instance Evaluable CChoice where
    eval (C l s c r) = madm [0.2,0.3,0.3,0.2] [evalR l,g s,evalL c,evalL r]
         where
            g = \v -> v/120000.0

instance Evaluable HChoice where
    eval (H m i p) = madm [0.4,0.3,0.3] [g m,evalL i,evalB p]
         where
            g = \v -> (fromIntegral v)/800.0


instance StableMarriage Candidate Hospital CChoice HChoice
instance StableMarriage Hospital Candidate HChoice CChoice
```

The output of the NRMP problem in shown in Figure 3 which matches the output of our DSL.

```
> showMatch(solveP :: Match Candidate Hospital)
Sunny:
                 Matched with []
                 Remaining capacity: 1

Darrius:
                 Matched with [City]
```

```
                 Remaining capacity: 0

Latha:
                 Matched with [General]
                 Remaining capacity: 0

Joseph:
                 Matched with [General]
                 Remaining capacity: 0

Arthur:
                 Matched with [City]
                 Remaining capacity: 0


> showMatch(solveP :: Match Hospital Candidate)
General:
                 Matched with [Latha,Joseph]
                 Remaining capacity: 0

Mercy:
                 Matched with []
                 Remaining capacity: 2

City:
                 Matched with [Arthur,Darrius]
                 Remaining capacity: 0
```

Stable roommate is a matching problem where the source as well as the target set is the same. An encoding of a stable roommate problem in our framework is shown below:

```
import DataType

data Student = Charlie | Peter | Elise | Paul | Kelly | Sam deriving (Eq,Show)

instance MatchSet Student where
    members = [Charlie,Peter,Elise,Paul,Kelly,Sam]

data SChoice = S {sameCountry::Bool,sameSpec::Bool,tasteCompt::Level,habitsCompt::Level}

sch :: Bool -> Bool -> Level -> Level -> Maybe SChoice
sch c s i h = return $ S {sameCountry = c,sameSpec = s,tasteCompt = i,habitsCompt = h}

studCh = expressCh [Charlie,Peter,Elise,Paul,Kelly,Sam]

charlieChoice = studCh [Nothing,sch True True VHigh VHigh,sch False True Low VLow,
                        sch True True High High,sch True False Low Med,sch True True High Med]

petersChoice= studCh [sch False False Med Low,Nothing,sch True True High Med,
                      sch True False Med Med,sch True True High High,sch True False Med High]

eliseChoice = studCh [sch False True Low Low,sch True True High High,Nothing,
                      sch True False VLow Low,sch False True Med Med,sch True True Med Low]

paulsChoice = studCh [sch True True Med Med,sch False True Med Med,sch True True High High,
```

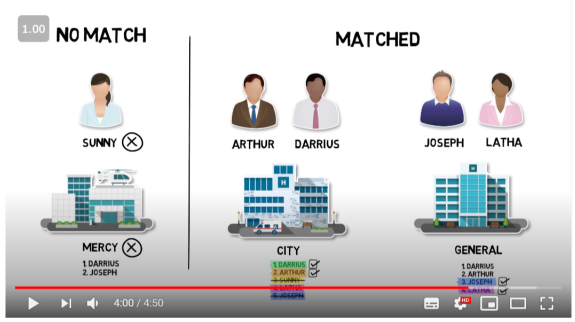

Figure 3: Stable NRMP Match

```
                      Nothing,sch False False Med Low,sch True True Low Med]

kellysChoice= studCh [sch True True Med High,sch True True High VHigh,sch False True Med Med,
                      sch False True Med Low,Nothing,sch True True Low Med]

samsChoice  = studCh [sch True True High High,sch False False Med Low,sch False True Low Low,
                      sch False True High Med,sch False True Med Med,Nothing]

instance Relate Student Student SChoice where
    assignVal = \case {Charlie -> charlieChoice;Peter -> petersChoice;Elise -> eliseChoice;
                       Paul -> paulsChoice;Kelly -> kellysChoice;Sam -> samsChoice}

instance Evaluable SChoice where
    eval (S x y z w) = madm [0.25,0.25,0.25,0.25] [evalB x,evalB y,evalL z,evalL w]

instance StableRoommate Student SChoice
```

5 Basic Objects

The structure of the DSL in the meta language Haskell is shown below. `MatchSet` is a type class which models the source and target sets of the things being matched. Every element of the source and the target set is associated with its capacity. The `members` function list out all the members of the source and the target set. `constraint` function lists out all the constraints that exist between the elements of the source or the target set. `Relate a b c` is a type class which matches the soruce set `a` with the target set `b` using a type `c`. `assignVal` is a function that assigns an element of type `a`, with an element of type `b` and assigns a value of type `Maybe c`. `Evaluable` type class takes a data type and evaluates it to a number. Finally, the `StableMarraige a b c d` type class associates set `a` with `b` using `c`, and set `b` with `a` using `d`.

```
-- ==========================================================================================
class (MatchSet a,MatchSet b,Evaluable c) => Relate a b c | a b -> c where
    assignVal :: a -> b -> Maybe c
    getRanks ::  Rank a b


class MatchSet a where
    members :: [a]
```

```haskell

    capacity :: a -> Int
    capacity _ = 1

--type Rank = Int
type Rank a b =  [(a,[b],TCapacity)]

class (MatchSet a,MatchSet b,Evaluable c) => Relate a b c | a b -> c where
     -- Nothing is used to represe  nt the no assignment case for an element of 'a'
    assignVal :: a -> b -> Maybe c

    getRanks ::  Rank a b
    getRanks = [(p,f [(q,(evalM.assignVal p) q) | q <- members], capacity p) | p <- members]
        where
            f = map fst.reverse.sortBy (compare `on` snd).filter (\(x,y) -> y /= Nothing)

class Evaluable a where
    eval :: a -> Double

    evalM :: Maybe a -> Maybe Double
    evalM Nothing = Nothing
    evalM (Just x) = Just $ eval x
-- ======================================================================================
type Match a b = [(a,[b],Int)]
type RankP a b =  (Rank a b,Rank b a)

class  (Relate a b c,Relate b a d,Eq b,Eq a) => StableMarriage a b c d | a b -> c d where
    solveP :: Match a b
    solveP = map (\(p,(_,r,_,t)) -> (p,r,t)) ls
        where
          ls = galeShapley (f x) (f y) (f x)
          (x,y) = (getRanks,getRanks)
          f = map (\(a,b,c) -> (a,(b,[],c,c)))

class Relate a a b => StableRoommate a b | a -> b where
    solveS :: Match a b
    solveS = undefined

-- ======================================================================================
```

6 Operators and Combinators

One of the issues with the DSL was the verbosity in expressing the preferences of source elements for the target elements. For example, consider the marriage example in Section **??**. A naive representation of the preference would look like the following:

```haskell
instance MatchSet Candidate where
    members = [Arthur,Sunny,Joseph,Latha,Darrius]

instance MatchSet Hospital where
    members = [City,Mercy,General]
    capacity _ = 2

```

```
alicesChoice :: Man -> Maybe WChoice
alicesChoice = \case Bob  -> return $ W {politicsSimi = False,intcompt = 7,wlooks = 7}
                     Ben  -> return $ W {politicsSimi = True,intcompt = 10,wlooks = 10}
                     Jack -> return $ W {politicsSimi = True,intcompt = 8,wlooks = 7}

janesChoice :: Man -> Maybe WChoice
janesChoice = \case Bob  -> return $ W {politicsSimi = True,intcompt = 10,wlooks = 7}
                    Ben  -> return $ W {politicsSimi = False,intcompt = 5,wlooks = 8}
                    Jack -> return $ W {politicsSimi = False,intcompt = 9,wlooks = 10}

elisChoice :: Man -> Maybe WChoice
elisChoice = \case Bob  -> return $ W {politicsSimi = True,intcompt = 7,wlooks = 10}
                   Ben  -> return $ W {politicsSimi = False,intcompt = 10,wlooks = 4}
                   Jack -> return $ W {politicsSimi = True,intcompt = 4,wlooks = 9}


annsChoice :: Man -> Maybe WChoice
annsChoice = \case Bob  -> return $ W {politicsSimi = True,intcompt = 8,wlooks = 9}
                   Ben  -> return $ W {politicsSimi = True,intcompt = 7,wlooks = 9}
                   Jack -> return $ W {politicsSimi = False,intcompt = 9,wlooks = 9}

instance Relate Woman Man WChoice  where
    assignVal = \case {Alice -> alicesChoice;Jane -> janesChoice;Jill -> jillsChoice;Eli -> elisChoice;Ann -> annsC
```

The verbosity of the representation might be detrimental to the use of the DSL. Therefore, we define the combinator `expressCh`, as shown below, to make the representation succinct.

```
expressCh :: (MatchSet b,Evaluable c,Eq b) => [b] -> [Maybe c] -> (b -> Maybe c)
expressCh bs cs b = cs !! i
    where i = (fromJust.elemIndex b) bs
```

The succinct representation is shown below:

```
wch :: Bool -> Rating -> Rating -> Maybe WChoice
wch b i w = return $ W {politicsSimi=b,intcompt=i,wlooks=w}

womensCh = expressCh [Bob,Ben,Jack]

alicesChoice= womensCh [wch False 7 7,wch True 10 10,wch True 8 7]
janesChoice = womensCh [wch True 10 7,wch True 10 10,wch True 8 7]
jillsChoice = womensCh [wch True 8 10,wch True 9 9,wch False 10 9]
elisChoice = womensCh [wch True 7 10,wch False 10 4,wch True 4 9]
annsChoice = womensCh [wch True 8 9, wch True 7 9,wch False 9 9]

instance Relate Woman Man WChoice  where
    assignVal = \case {Alice -> alicesChoice;Jane -> janesChoice;Jill -> jillsChoice;Eli -> elisChoice;Ann -> annsC
```

7 Implementation Strategy

The class system of Haskell is useful in modelling the stable matching problems in a way that is closer to the domain. Having access to such advanced features of Haskell was useful in implementing the DSL.

```
## 3 men, 2 women, given preferences:
 s.prefs <- matrix(c(1,2, 1,2, 1,2), 2,3)
 c.prefs <- matrix(c(1,2,3, 1,2,3), 3,2)
 hri(s.prefs=s.prefs, c.prefs=c.prefs)

## 3 men, 2 women, given preferences:
 s.prefs <- matrix(c("x","y", "x","y", "x","y"), 2,3)
 colnames(s.prefs) <- c("A","B","C")
 c.prefs <- matrix(c("A","B","C", "A","B","C"), 3,2)
 colnames(c.prefs) <- c("x","y")
 hri(s.prefs=s.prefs, c.prefs=c.prefs)
```

Figure 4: R Code for Specifying Stable-Marriage Problem

Deep embedding would have been useful in providing a succinct syntax for the language which seems a bit verbose and unnatural at the moment.

8 Related DSLs

There is a R library[1] which can be used to model matching algorithms. A code fragment from an **R** library shown in Figure 4. However, the major difference the R package and our DSL is that the R package directly takes ranks as input whereas our DSL take abstract criterion of the end user, and then evaluates them to get the rank. This abstraction makes our DSL more end user friendly.

References

1. Klein, T. (2015). "Analysis of Stable Matchings in R: Package matchingMarkets" (PDF). Vignette to R Package MatchingMarkets.