CS 421: Programming Languages

Final Project

Zijian Jing

zijianj2@illinois.edu

1. Overview

In this project, our goal is to use Haskell to solve a very popular puzzle – *Einstein's riddle*. As illustrated in *Escape from Zurg* (Erwig, 2004), the powerful type system, higher-order functions, and lazy evaluation provided by Haskell made it a good tool to solve search problems. We successfully solved the puzzle in under 200 lines of code and demonstrated how powerful Haskell is in logic programming.

2. Implementation

According to https://udel.edu/~os/riddle.html, *Einstein's riddle* reads as the follows:

The Situation

1. There are 5 houses in five different colors.
2. In each house lives a person with a different nationality.
3. These five owners drink a certain type of beverage, smoke a certain brand of cigar and keep a certain pet.
4. No owners have the same pet, smoke the same brand of cigar or drink the same beverage.

The question is: *Who owns the fish?*

Hints

1. the Brit lives in the red house
2. the Swede keeps dogs as pets
3. the Dane drinks tea
4. the green house is **on the left** of the white house
5. the green house's owner drinks coffee
6. the person who smokes Pall Mall rears birds
7. the owner of the yellow house smokes Dunhill
8. the man living in the center house drinks milk
9. the Norwegian lives in the first house
10. the man who smokes blends lives **next to** the one who keeps cats
11. the man who keeps horses lives **next to** the man who smokes Dunhill
12. the owner who smokes BlueMaster drinks beer
13. the German smokes Prince
14. the Norwegian lives **next to** the blue house
15. the man who smokes blend has a **neighbor** who drinks water

Based on the situation and the *hints* given, our code not only gives an answer to *Who owns the fish?* but also provides a complete mapping of location, color, nationality, beverage, smoke, and pet.

To set up the problem, first we declare six data types.

Fig. 1 Data type declaration

```
data Location    = One | Two | Three | Four | Five                deriving (Eq,Enum,Show)
data Color       = Red | White | Green | Yellow | Blue            deriving (Eq,Enum,Show)
data Nationality = Brit | Swede | Dane | Norwegian | German       deriving (Eq,Enum,Show)
data Beverage    = Tea | Coffee | Milk | Beer | Water             deriving (Eq,Enum,Show)
data Smoke       = Pallmall | Dunhill | Bluemaster | Prince | Blend deriving (Eq,Enum,Show)
data Pet         = Dog | Bird | Cat | Horse | Fish                deriving (Eq,Enum,Show)
```

These six types include all the key entities in this riddle. Note that there are five possible values each, which is necessary for mutual exclusivity as stated in *the situation*. Here `deriving (Enum)` is helpful for us to enumerate through all possibilities later.

Next, we define `Profile` that represents one possible combination.

```
type Profile = (Location, Color, Nationality, Beverage, Smoke, Pet)
```

We also want a list of all possible candidates.

```
type Candidates = [Profile]
```

Here we will start with a list of all combinations with replacement, but any violations of the *hints* will be filtered out. Specifically, except for *hints* 4, 10, 11, 14 and 15 that are about proximity, all other *hints* could be evaluated on individual candidates. To give an example using *hint* 1, if any candidate profile has only one of the two – `Color == Red` and `Nationality == Brit` – but not both, then due to mutual exclusivity, it is in violation of the *hints* and needs to be filtered out. The detailed implementation of *hint* 1 looks like the below.

Fig. 2 No violation of *hint* 1

```
-- h1: the Brit lives in the red house
h1 :: Profile -> Bool
h1 (_, Red, Brit, _, _, _) = True
h1 (_, Red, _,    _, _, _) = False
h1 (_, _,   Brit, _, _, _) = False
h1 _ = True
```

We could also see how pattern matching is being useful in Haskell above. Using higher-order function `filter` and list comprehension, we are able to create a filtered list of candidates below with no violations of non-proximity *hints*.

Fig. 3 Candidates with no violation of non-proximity *hints*

```
-- candidates filtered with all non-proximity rules
candidates :: Candidates
candidates = filter (combine [h1,h2,h3,h5,h6,h7,h8,h9,h12,h13])
             [(l, c, n, b, s, p) |
               c <- [Red ..],
               n <- [Brit ..],
               b <- [Tea ..],
               s <- [Pallmall ..],
               p <- [Dog ..],
               l <- [One ..]
             ]
```

Here, `combine` is defined below with recursion, and it basically is saying that candidate profiles should not violate any *hints*.

Fig. 4 Definition of `combine`

```
-- combine conditions
combine :: [a -> Bool] -> a -> Bool
combine (f:fs) xs = (f xs) && (combine fs xs)
combine []     xs = True
```

Next, we use the list of candidates to construct a solution of length five, representing five mutually exclusive household profiles. We define type `Solution` as follows:

```
type Solution = [Profile]
```

We also define a solution space that serves as a container for all possible solutions.

```
type Space = [Solution]
```

Note that one might wonder why we are filtering on candidates first here, but not creating the space of all possible mutually exclusive solutions first. The reason is efficiency. Given that 10 out of 15 *hints* could be evaluated on individual candidates, filtering on candidates first could significantly reduce our search space. In fact, if we are creating a space of all possible mutually exclusive solutions first, there could be $(5!)^5$ or ~2.49e+10 possibilities for us to check.

On the other hand, if we start with candidates, there are $5^6 = 15625$ candidates in total, and with filtering we are left with only 133 candidates. Specifically, there are 21 candidates with `location == One`, 33 candidates with `location == Two`, 13 candidates with `location == Three`, 33 candidates with `location == Four`, and 33 candidates with `location == Five`. Size of the total search space becomes 21*33*13*33*33 or ~9.81e+06 which is much more manageable.

The next step is to ensure mutual exclusivity, and search for solutions that satisfy the five proximity *hints* (i.e., 4, 10, 11, 14, and 15) in the search space, as shown below.

Fig. 5 Final solution space

```
-- solution space
space :: Space
space = filter (combine [h4,h10,h11,h14,h15,mutex])
        [[pf1, pf2, pf3, pf4, pf5] |
          pf1 <- filter r1 candidates,
          pf2 <- filter r2 candidates,
          pf3 <- filter r3 candidates,
          pf4 <- filter r4 candidates,
          pf5 <- filter r5 candidates]
```

First, we put candidates to the right location. For instance, candidates with `location == One` will be `pf1` in the solution list.

```
[pf1, pf2, pf3, pf4, pf5]
```

Specifically, it is done through

```
pf1 <- filter r1 candidates
```

with `r1` defined below.

Fig. 6 Location rule `r1`

```
-- location rules
r1 :: Profile -> Bool
r1 (One, _, _, _, _, _) = True
r1 _ = False
```

Next, we work on the proximity *hints*. Now that we have the candidates in the right location of the solution list, we could simply use recursion to check whether a proximity *hint* is met or not. Use `h4` as an example below.

Fig. 7 Proximity *hint* `h4`

```
-- h4: the green house is on the left of the white house (proximity)
h4 :: Solution -> Bool
h4 ((l1, c1, n1, b1, s1, p1):(l2, c2, n2, b2, s2, p2):xs) = (c1 == Green && c2 == White) || (h4 ((l2, c2, n2, b2, s2, p2):xs))
h4 [x] = False
```

The last step is to ensure mutual exclusivity through `mutex`. To do this, first we define a helper function `conf` which checks if two candidate profiles are in conflict. The idea is simple, if any entity of one profile is the same as the entity of the other profile, they are in conflict.

Fig. 8 Helper function `conf`

```
-- check if two profiles are in conflict
conf :: Profile -> Profile -> Bool
conf (l1, c1, n1, b1, s1, p1) (l2, c2, n2, b2, s2, p2) = (l1 == l2) || (c1 == c2) || (n1 == n2) || (b1 == b2) || (s1 == s2) || (p1 == p2)
```

Now we define `mutex` as follows.

Fig. 9 Mutual exclusivity rule `mutex`

```
-- mutually exclusive
mutex :: Solution -> Bool
mutex (x:xs) = (not (any (conf x) xs)) && (mutex xs)
mutex []     = True
```

Now that all criteria are met, we just need to take 1 solution from the solution space.

```
solution = take 1 space
```

3. Result and Conclusion

Below is one solution of the riddle, from which we know that *German owns the fish*.

Table. 1 Riddle solution

Location	Color	Nationality	Beverage	Smote	Pet
One	Yellow	Norwegian	Water	Dunhill	Cat
Two	Blue	Dane	Tea	Blend	Horse
Three	Red	Brit	Milk	Pall Mall	Bird
Four	Green	**German**	Coffee	Prince	**Fish**
Five	White	Swede	Beer	Blue Master	Dog

In this project, we solved *Einstein's riddle* using Haskell. We used higher-order functions, recursions, and pattern matches multiple times in searching for the solution and found these characteristics very helpful, efficient, and intuitive. The type declarations make the project structured and easy to debug. With its specific characteristics, Haskell is indeed a powerful tool in solving search problems and puzzles in general.

4. Code

The code of this project could be found at the below link.

https://github.com/zjing20/su21-cs421-project/blob/main/src/Einstein.hs

References

Erwig, M. (2004). Escape from Zurg: An Exercise in Logic Programming. *Journal of Functional Programming*, *14*(3), 253-261.

Einstein's riddle. Einstein's five-houses riddle. (n.d.). https://udel.edu/~os/riddle.html.