Functional Logic Problem Solver

James Maher
B.A.(Mod.) Computer Science
Final Year Project April 2014
Supervisor: Hugh Gibbons

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

James Maher, 23rd April 2014
Contents

1. Chapter 1 – Introduction
2. Chapter 2 – Background
3. Chapter 3 – Solving Through Reduction
4. Chapter 4 – The Haskell Implementation
5. Chapter 5 – Reusable Rules
6. Chapter 6 – The Zebra Problem
7. Chapter 7 – Conclusions and Future Work
8. Bibliography
9. Appendices
   1. marriage_solver.hs
   2. getable_marriage_solver.hs
   3. zebra_solver.hs
Chapter 1

Introduction

Logic problems are a problem of finding different combinations of given elements. The user is given a set of clues and asked to find the set of combinations that are able to satisfy these clues by finding which elements go into each combination. This report will outline a method of doing this using the functional language Haskell.

These problems can be complex and, depending on the number of elements, can have many possibilities to be reduced to find the answer.

The aims of this project are to create a way of solving these problems that is scalable and reusable.

In this report a method of reduction will be adopted to solve these problems. It will be outlined with an example. A program will be devised to perform this program and from that program a more sustainable framework for solving these problems will be derived.

The report will deal with the motivations and design of the program design. The whole program, in its various versions will be included in the appendix.
Chapter 2

Background

Logic problems are usually solved using, single use, generate and test programs. These have the advantages of being relatively short programs and simple to implement. In a language like Prolog where the ability of backtracking, which is useful for generating combinations, is part of the language semantics they can be very easily implemented.

The problem with this method is that it does not scale well with the size of the problem. The more elements that are in the problem the number of combinations greatly increases. For larger problems these methods are not viable as a problem like the Zebra problem, Einstein's riddle, can take a matter of seconds to solve. This leads to the aim of using a more scalable algorithm to be used to solve the problems. Reduction is the method used in this project as it can deal with multiple combinations at once.

Finding combinations like this is a feature on constraint logic programming. It is used in many fields to generate possibilities in a given domain. It is used in timetabling systems and artificial intelligence to find applicable possibilities of actions or combinations that could be used. These are similar to the problems that are addressed in this report as they deal with possibilities within a tabular format and placing elements based where they do not violate rules.
Chapter 3

Solving Through Reduction

Reduction is a technique where all the permutations viewed as correct until they can be proven not to be. This has the advantage over traditional generate and test methods because each element removed removes a number of permutations. A generate and test method would have to create these separately. This has the distinct advantage of being faster and more scalable so it can be used again for larger problems.

The permutations can be viewed as a table where the number of columns is the number of combinations of elements required. The rows contain lists for each type of element. At the beginning the lists will contain all of their respective elements which would represent the maximum permutations.

Rules need to be defined from the problem's clues. These rules contain a relation between two elements. If the rule is false the elements can be checked for if they can be removed. Also if one of the elements is solved, the only element in its respective list, the other can be fixed in its position removing all other possibilities for it and in the list it has been fixed to.

The most fundamental reason behind this style of algorithm is that the rules can be used iteratively over the table until the table stops having elements removed. This leaves the elements in positions where they can be true. Elements should only be removed if they are known to be wrong and rules should only cause elements to be removed. Otherwise incorrect elements would be placed back into the table because they have to be wrong to have been removed in the first place.

Solving the Marriage Problem Through Reduction

The marriage problem in question is a small logic problem involving the day is which five men and women are married on separate days from Monday through Friday.

The types of elements are Women Men and Days. The lists in the table will consist of all elements of each of these types at the beginning.

Women = Anne, Cathy, Eve, Fran, Ida
Men = Paul, Rob, Stan, Vern, Wally
Days = Monday-Friday

The formal rules will be derived from these clues given by the problem.

Clues
1. Anne was married on Monday, but not to Wally.
2. Stan's wedding was on Wednesday. Rob was married on Friday, but not to Ida.
3. Vern (who married Fran) was married the day after Eve.
The Table

Since each row must have a unique element of any type one can be chosen to index the columns of the table. In this case the most useful is the days because it is the only type that is ordered. Now the order of the columns can imply the values for the days. This reduces the available permutations while not reducing the available combinations for the columns because each column can have at most one unique day.

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anne</td>
<td>Anne</td>
<td>Anne</td>
<td>Anne</td>
<td>Anne</td>
</tr>
<tr>
<td>Cathy</td>
<td>Cathy</td>
<td>Cathy</td>
<td>Cathy</td>
<td>Cathy</td>
</tr>
<tr>
<td>Eve</td>
<td>Eve</td>
<td>Eve</td>
<td>Eve</td>
<td>Eve</td>
</tr>
<tr>
<td>Fran</td>
<td>Fran</td>
<td>Fran</td>
<td>Fran</td>
<td>Fran</td>
</tr>
<tr>
<td>Ida</td>
<td>Ida</td>
<td>Ida</td>
<td>Ida</td>
<td>Ida</td>
</tr>
<tr>
<td>Paul</td>
<td>Paul</td>
<td>Paul</td>
<td>Paul</td>
<td>Paul</td>
</tr>
<tr>
<td>Rob</td>
<td>Rob</td>
<td>Rob</td>
<td>Rob</td>
<td>Rob</td>
</tr>
<tr>
<td>Stan</td>
<td>Stan</td>
<td>Stan</td>
<td>Stan</td>
<td>Stan</td>
</tr>
<tr>
<td>Vern</td>
<td>Vern</td>
<td>Vern</td>
<td>Vern</td>
<td>Vern</td>
</tr>
<tr>
<td>Wally</td>
<td>Wally</td>
<td>Wally</td>
<td>Wally</td>
<td>Wally</td>
</tr>
</tbody>
</table>

The Rules

Using the clues given in the problem the more precise and formal rules need to be defined to check the elements in the table. The rules need to able able to represent a relationship between two of the elements.

<table>
<thead>
<tr>
<th>(Anne, Monday)</th>
<th>(Anne, ~Wally)</th>
<th>(Stan, Wednesday)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Rob, Friday)</td>
<td>(Rob, ~Ida)</td>
<td>(Vern, Fran)</td>
</tr>
<tr>
<td></td>
<td>(Eve, Vern) *</td>
<td></td>
</tr>
</tbody>
</table>

Here there are the pairs of elements that have relations given in the clues. The tilde is used to show that the other element is related to the absence of the one with the tilde. There is a special case for (Eve, Vern) because that relation is dependent on their relations to another type. We will look at this later.

Pair Rule

The first six relations are that the two given elements are within the same combination and therefore column in the table. Firstly some of these relations contain days. Since days have already been fixed they can be fixed in the table directly.
It is known where Ann, Stan and Rob are on the table because they are paired with days and days have been used to index the table. This then means that all other elements of their respective types on their respective days can be removed. This means that these elements are the only one of their type in combination, this means that element can be considered solved because no element appears in more than one combination. Solved elements can be removed from all other combinations because of this property.

Since the pair relations containing the type used to index the table can be applied directly and leaving both elements solved on the first iteration there is no use to applying these rules iteratively. Instead they can be separated and applied once before the other pair rules. From now on rules like this will be categorised as direct rules.

The remaining pair rules will be used to examine the remaining elements in each combination and elements that are in contradiction to the rule can be removed. Before that is done a more formal definition of what makes a pair rule can be established.

A pair rule is true 'if and only if' both elements are in their respective lists for a combination. So first the values of the elements can be converted into boolean values by checking if they are elements in the combination, where true is an element and false is not. 'If and only if' can be broken down to two separate logical implications. For the pair (a, b) in the combination 'com' it would be;

1. (a in com) → (b in com)
   and
2. (b in com) → (a in com)

When a pair rule is false something needs to be removed so it is true again. Logical implication is only false if the first element is true and the second is false so, in the case of one, this means that a is in com but b is not. A way to fix this is to add b but we only want to remove elements because any element that has been removed already had violated a rule. So a is removed which leaves false implies false which is true. In the case of 2 being false, b is removed for the same reason.

This only works for pairs where both elements are positive. It is not possible to remove the negation of an element but rules that have negatives can be used if an element in the rule is a solved element. This allows the other element of the rule to be fixed because the other cannot appear anywhere else because of the definition of the problem. This is useful for both when one or both are true.

Where both elements of the rule are negative, this is equivalent to when both elements are positive.

Now these rules are like as follows:

<table>
<thead>
<tr>
<th>Direct</th>
<th>(Anne, Monday)</th>
<th>(Stan, Wednesday)</th>
<th>(Rob, Friday)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>(Anne, ~Wally)</td>
<td>(Rob, ~Ida)</td>
<td>(Vern, Fran)</td>
</tr>
</tbody>
</table>
Relative Rules

3. Vern (who married Fran) was married the day after Eve.

This rule, where Vern and Eve are concerned, provides an issue. Although the two elements are related, the relation is between two combinations. The relation between these two combinations is that one has a day after the other. This is why using a type that has an ordered property as the type initially indexed is useful. This rule is now quite clear in the table and could be checked by just checking the structure of the table. Apart from this offset the rule means something similar to the pair rule.

\[
1. \text{(Eve in com1)} \rightarrow \text{(Vern in com2)} \\
\text{and} \\
2. \text{(Vern in com2)} \rightarrow \text{(Eve in com1)} \\
\text{where (day of com2) – (day of com1) = 1}
\]

Here the where qualification encapsulates the difference between two days in the combinations in question. The subtraction is using the ordering of the days to enumerate them from 0 through 4 and means that the day of com2 is one after the day of com3. This is easy to see on the table because the days are used to index columns and this rule means that Vern is in the column after the column with Eve.

It is also possible to fix one of the elements in the relation if the other has been solved like in the pair rule.

Applying the Rules

Now that all the clues have been reduced into rules the rules can be applied to the table to solve for the answer to the problem. Starting with the table that has has the direct rules already applied, from above, the pair and relative rules can be applied repeatedly until the table stops losing elements. For this example an iteration will consist of the pair rules being applied together asynchrony followed by the relative rule and then the next iteration.

Since Anne and Rob are solved elements the rules containing them can be fixed onto the table. This will remove Wally from Monday and Ida from Friday respectively. The rule of pair(Vern, Fran) is positive on both sides so it can be checked using the implications 1 and 2 above to find if elements can be removed.

<table>
<thead>
<tr>
<th>pair(Vern,Fran)</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vern in com</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Fran in com</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

This tables shows, from the definition of the pair rule above, that Vern can be removed on Monday and Fran can be removed from Wednesday and Friday in the table. To view this more intuitively, it is because they both have to be on the same day but on these days the other element is
Applying these removals from the table, the table now looks like.
Now the relative rule will be applied. Since it contains no solved elements and both sides are positive it can be applied in the same style. The only reason that this is actually being separated is because of the conceptual difference between it and the pair rule. It can, in practice, be applied at the same time.

<table>
<thead>
<tr>
<th>Relative(Eve, Vern,1)</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eve in com1 → Vern in com2</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Vern in com2 → Eve in com1</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

In the above table, the first row takes the column's day as com1 and the second row takes the column as com2. This really means on each column Eve in com1 → Vern in com2 is checking forward and the other is checking backwards. This means that when com2 is Monday, com1 is a non-existent day and similarly for when com1 is Friday, com1 does not exist. For these non-existent days the search for an element will always be false because they cannot be there. This means that Eve on Friday and Vern on Monday will always be removed. The reason that the check for Vern on Monday is True is because it is already removed. Now the removals of Eve from Tuesday, Thursday and Friday and Vern from Tuesday can be applied to the table.

On Friday, Cathy and Monday, Paul have been solved. This is because no other element of their type is in the same combination. All other instances of Cathy and Paul can be removed from the table. Although Eve is intuitively solved it is not, in the current definition of solved, so no action will be taken. This action is propagating the solved elements of the table.
Now the second iteration can begin. The rules pair(Anne, ~ Wally) and pair(Rob, ~Ida) are True because Anne and Rob are solved and cannot be paired with Wally or Ida respectively. The rule pair (Vern, Fran) will be analysed again to find any more elements that can be removed.

<table>
<thead>
<tr>
<th>pair(Vern,Fran)</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vern in com → Fran in com</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Fran in com → Vern in com</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

The table shows that Fran can be removed from Tuesday so that will be applied. Since the relative rule is currently true for all values it will be skipped and the solved elements of Wally, Stan and, after the removal of Fran, Ida can be propagated as well.

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anne</td>
<td>Ida</td>
<td>Eve</td>
<td>Fran</td>
<td>Cathy</td>
</tr>
<tr>
<td>Paul</td>
<td>Wally</td>
<td>Stan</td>
<td>Vern</td>
<td>Rob</td>
</tr>
</tbody>
</table>

Now all elements of the table have been solved the table cannot reduce any further. The rules can be checked again to ensure that the table is correct but since elements are only removed when it is provable that they are not correct it is unlikely that it is not.
Chapter 4
The Haskell Implementation

The reduction method is easily implemented in a functional format. Once the table is implemented the rules can be viewed as functions where a table is given and the function returns a table that has had any elements that are in violation of the rule have been removed.

This chapter will look at a Haskell implementation for the marriage problem that was used as the example for the reduction method. It will involve creating a design for the table, defining rules to work over this data structure and a function that will drive the solving algorithm. The full implementation is available in appendix 1.

The Table

The table holds information on three categories in the Marriage puzzle; Men, Women and Days. In its most general form it would be a the full table showing all the options for all the positions for each combination. One type can be fixed and it's value can be inferred from the index within the table instead, as discussed in the previous chapter. This means the table can be a list of days using the index to evaluate from Monday through Friday.


type Day a b = ([a], [b])

The other types can be placed in a tuple that holds the lists of options still available for that particular day. This leaves defining the types Man and Woman for the data structure. These can be put in the program using empty data constructors. The typeclasses Eq and Show can be derived automatically by ghc so the types are equatable, can use the function (==), and can be printed.


data Man = Paul | Rob | Stan | Vern | Wally
deriving (Eq, Show)

data Woman = Anne | Cathy | Eve | Fran | Ida
deriving (Eq, Show)

The Rules

Now functions need to be defined so the rules can be implemented. They need to be of type [Day Man Woman] → [Day Man Woman] so they can be continually applied to the same table. This has the advantage of making the rules composable, so they can be put together into a larger function and applied to a table. Different rules will also need different arguments though because different rules need different information to check. This is easily dealt with by higher order functions in Haskell. If the rules are defined in such a way that the [Day] is the last argument all the other arguments can be passed and the remaining function can still be manipulated until it is applied to a table.
The Direct Rule

This is the version of the pair rule that has an element in a type that has been fixed, in this case the days. This allows these pairs to have the elements that are not days fixed in position in the [Day]. Since the days are represented as an index the function only needs the element in question and the index of the Day it is to be fixed. Two functions can be defined to perform this rule, one for each type. It needs to remove the other elements from the given index and remove any instance of the given element from the other indices. This will leave the only instance of the given element at the index defined in the rule.

\[
\text{direct1} :: \text{Eq a} \Rightarrow \text{a} \rightarrow \text{Int} \rightarrow [\text{Day a b}] \rightarrow [\text{Day a b}]
\]
\[
\text{direct1} \_ \_ \[] = []
\]
\[
\text{direct1} \text{man index ((manlist, wm):t)} = ((\text{if index == 0})
\]
\[
\text{then ([man], wm)}
\]
\[
\text{else (delete man manlist, wm))}
\]
\[
: \text{direct1 man (index - 1) t)
\]

It is simply done by pattern matching the existing lists and building a tuple with the new lists. It fixes the return if the index is equal to zero. The index is decrementing every time the function recurses. The function for direct2 does the same but for the other side of the tuple.

The Pair Rule

The pair rule has a problem one additional problem that the direct rules do not and that is the different positive and negative values that can be given to elements. This could be handled using different functions but instead a data type was used to hold this information.

\[
\text{data Val a = T a | F a}
\]

Instead of passing an element type to the pair function, like Man or Woman, a Val Man or Val Woman can be passed and a pattern match can discover if the element is positive or negative. Using this pattern match will allow for different definitions for the pair function that is based on the Val data constructors passed to it.

Now guards can be used to check one of the elements are solved in the tuple and if it is contained in the function's rule. If it is the other side can be fixed in position in the tuple function can stop.

\[
\text{pair va@(T a) vb@(T b) ((alist, blist):tail)} | \text{alist == [a]} = (([a], [b]):tail)
\]
\[
| \text{blist == [b]} = (([a],[b]):tail)
\]

If both elements are positive then the further tests can be performed to remove elements before recursing. If one is negative then it can recurse immediately to the next element of the list.

\[
|\text{otherwise} = \text{case (aval, bval) of}
\]
\[
(\text{True, True}) \rightarrow ((\text{alist, blist)}):\text{pair va vb tail})
\]
\[
(\text{True, False}) \rightarrow ((\text{delete a alist, blist)}):\text{pair va vb tail})
\]
\[
(\text{False, True}) \rightarrow ((\text{alist, delete b blist)}}):\text{pair va vb tail})
\]
\[
(\text{False, False}) \rightarrow ((\text{alist, blist)}):\text{pair va vb tail})
\]
\[
\text{where aval = elem a alist}
\]
\[
\text{bval = elem b blist}
\]
This case statement in the function clearly separates the different actions that are applied depending on the elements being in the day. Where the values are defined in the where statement. The results of the case statement are based on the actions that are defined for the implication rules for the pair rule in the previous chapter.

The Relative Rule

This rule is similar to the pair rule so it will be given arguments in Val a format even though for these problem they can be left unimplemented. This function cannot simply recurse like the previous two because of the offset element means that it needs to access more than one Day \( a \) \( b \) in the list. Also a way for ensuring that the rules is checked backwards and forwards needs to be applied.

The strategy used to check this rule is to check each individual instance in the one direction as the function goes though the list and where the relation is impossible, where one of the indices must be outside the list, the element is removed. These two actions combined will remove any elements that violate the rule.

To do these actions first a function will be needed to remove elements from a certain point in the list. This function is similar to the direct rule function except it does nothing until the index given is zero then it removes the given element. They are called \( \text{rm1} \) and \( \text{rm2} \), again one for each type.

The relative rule function is defined in two parts. There is the checking part which checks the different indices for the rule’s correctness and the part that removes the overlap where of the indices will be outside the list.

\[
\text{filter} \ (\text{not.checkRelation}) \ (\text{take} \ (\text{length} \ \text{week}) \ [0..])
\]

The checking part first filters a list of all the indices by whether the check is true or not. The check checks from the index of the Man type with the offset to the Woman type. The filter returns the list of the indices which the check failed and this is used to remove the elements of these relations from the list.

Since the check only starts at the Man elements the Woman elements that are at indices where the Man element would be outside the list are left behind by the check. This is solved by the function \( \text{overlapRemove} \) that composes a list of \( \text{rm2} \) functions so that these leftover elements are removed and applies the resulting function to the list.

\[
\text{overlapRemove} \ \text{offset} \ \text{week} \\
\quad | \text{offset} > 0 = \text{compose} \ (\text{map} \ (\lambda x \rightarrow \text{rm1} \ a \ x) \ [0..(\text{offset}-1)]) \ \text{week} \\
\quad | \text{offset} < 0 = \text{compose} \ (\text{map} \ (\lambda x \rightarrow \text{rm2} \ b \ x) \\
\quad \quad \quad [((\text{length} \ \text{week}) – 1)..((\text{length} \ \text{week})-1-\text{offset})]) \ \text{week}
\]

The compose function takes a list of functions and makes them into one using the composition operator. This function is useful for grouping functions to be applied together. Especially when a map is used to build the functions.
The Solve Function

The solve function is used to control the application of the different functions to the table. It is passed three functions that are applied to a [Day a b] and return a [Day a b]. What this is actually doing is separating the different types of functions and passing them to the solve. Since the functions can be composed, the rules of the same type can be composed together then passed to the solver.

```
directs = direct2 Anne 0 . direct1 Stan 2 . direct1 Rob 4
pairs = pair (F Wally) (T Anne) . pair (T Vern) (T Fran) . pair (T Rob) (F Ida)
relatives = relative (T Vern) (T Eve) (-1)
```

```
solveMarriage = solve directs pairs relatives theweek
```

The solver applies the directs rules to the full list then iterates the pairs and the relatives until they stop to reduce the table. This will reduce the problem to a unique answer.

```
solve directs pairs relatives week
    = let afterDirects = directs week in iterate afterDirects
        where iterate week
            = let after = propagateSolved $ relatives $ pairs week
                in if after == week
                    then after
                    else iterate after
```

The solve function uses a function called propagateSolved. This function is used to find any solved elements in the list and remove them from any of the day where they are not the only element of that type. It is actually defined from two functions.

```
propagateSolved x = propagate (solved x) x
```

The solved function returns a Day Man Woman that contains the elements that have been solved and propagate removes them from where they are not the only element.

With all these elements the solver can find the answer to the Marriage Problem and it shows a structure which these types of logic problems can be solved using reduction.
Chapter 5
Reusable Rules

The major limitation in the previous solution is that the rules are not reusable. Every time there is a new puzzle these will need to be redefined to be used if the puzzle has more than three types of elements. This is because of the pattern matching taking place within the rule functions. These pattern matches need to know the type of the structure being matched to work.

If the pattern matching is moved out of the rule functions the code will become much more reusable and actually much cleaner because of the more general type definitions used in the rule functions.

To achieve this typeclasses can be used to create functions that can have different instances for different types and these functions can be used in the rules so the rules can be defined for different instances of these.

These new features are in the program in appendix 2, implemented for the marriage problem.

The Typeclass

In the previous solution the main use of the tuple pattern was to extract and change the lists of elements within the tuples of the overall list. Functions can be defined to perform these operations within a typeclass.

\[
\text{type } \text{Container} = ([\text{Man}], [\text{Woman}])
\]

\[
\text{class } \text{Contained a where}
\]

\[
\text{get} :: \text{Container} \rightarrow [a]
\]

\[
\text{insert} :: [a] \rightarrow \text{Container} \rightarrow \text{Container}
\]

Now the definitions of the rule functions can be changed to no longer include any tuple patterns because they no longer contain how many elements are in the tuple. Instead the functions get and insert can be used to perform these operations where the type in question has an instance of the typeclass Contained for it.

The definition on the Container type hides the internal types from the type checker. This is important because it means that the type of the container will not effect the order of the arguments being pass to the rule function.

<table>
<thead>
<tr>
<th>Previous</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>if index == 0</td>
<td>if index == 0</td>
</tr>
<tr>
<td>then ([man], wm)</td>
<td>then insert [el] cont</td>
</tr>
<tr>
<td>else (delete man manlist, wm)</td>
<td>else remove el con</td>
</tr>
</tbody>
</table>

These changes will make the code within rules look more like this, where the container is pattern matched out at the beginning and used in insert and get functions to provide the functionality that once needed the type of the tuple to be known.
The example shows a segment from the direct function where changes to the current tuple are done. In the previous version the tuple is built from the from the information pattern matched out. It needs to know both lists and then modifies them and builds a new tuple in the function. The new version still does this but it is done by the functions insert and remove, remove uses get and insert to remove a single element from a container, the instance of which is decided from the type of 'el' which is passed to the function. The lack of the pattern matching also means that the new code is correct for all the types in the container that have instances of Contained, so there is no need for more versions of the direct rule for different types. This abstraction follows into the pair and relative functions as now the order in which the types of elements are passed is no longer important because the pattern match is performed in an outside function.

For this to work instances for these need to be defined and they are simple because they are only pattern matches.

\[
\text{instance Contained Woman where}
\]
\[
\begin{align*}
\text{get (_, women) &= women} \\
\text{insert womenList (men, _) &= (men, womenList)}
\end{align*}
\]

Here is the instance for Woman. The instance for Man is the same but the opposite sides of the tuple are used.

**Other Functions**

Unfortunately insert and get are not the only functions that need to use the pattern of the tuple. There are other utility functions that need to be defined to be used on the Container type. These functions are to do with propagating solved elements in the list. The pattern matches are needed here because all the types of elements need to be checked.

**Solved**

The function solved is defined to return only the lists of one element in a container. This will return any solved elements in one container. This is used with the findSolved function that recurses down the list and unions the results of the solved function for each Container in the list. This will then return a Container with all the solved elements.

**Propagate**

\[
\text{propagate (aList, bList) = compose (concat [map safeRemove aList, map safeRemove bList])}
\]

The propagate function also needs to know all the types in the Container so it can create a function that will remove the solved elements for each type. This is done by mapping a function called safeRemove, which removes an element from a Container if its no the only one, to the list of solved elements of a particular type. This creates a list of functions to safeRemove these elements. This is done for every type and then composed together, because they will all be of type \([\text{Container}] \to [\text{Container}]\), and then can be applied to the overall list. This is an example of the usefulness of composability of functions because it allows elements of different types to be removed together.
Direct

Apart from the change mentioned earlier, where there is no longer a need for different instances of different types, the direct function was also changed to use a Val a value passed to it. It will remove the element at the given index if it is a F constructed Val a. This just replaced the rm1 and rm2 functions from earlier.
Chapter 6

The Zebra Problem

This full program for this chapter is included as appendix 3.

This version of the Zebra Problem is taken from [rosettacode.org/wiki/Zebra_puzzle](http://rosettacode.org/wiki/Zebra_puzzle).

Now that the rules can be used on different size Containers it can be used for different puzzles. This chapter will add to the code to implement a solver for the Zebra Problem, or Einstein’s Riddle, using the reduction technique and the structures that are used in the previous examples. Here is one of the variations of the problem.

1. There are five houses.
2. The English man lives in the red house.
3. The Swede has a dog.
4. The Dane drinks tea.
5. The green house is immediately to the left of the white house.
6. They drink coffee in the green house.
7. The man who smokes Pall Mall has birds.
8. In the yellow house they smoke Dunhill.
9. In the middle house they drink milk.
10. The Norwegian lives in the first house.
11. The man who smokes Blend lives in the house next to the house with cats.
12. In a house next to the house where they have a horse, they smoke Dunhill.
13. The man who smokes Blue Master drinks beer.
14. The German smokes Prince.
15. The Norwegian lives next to the blue house.
16. They drink water in a house next to the house where they smoke Blend.

**The Elements and Types**

First to build the table in order to solve the problem using the reduction method the different elements and types of elements must be extracted from the clues. They are, in Haskell;

```haskell
data Colour = Red | Green | Blue | Yellow | White

data Man = Eng | Swe | Dan | Nor | Ger

data Pet = Dog | Birds | Cats | Horse | Zebra

data Drink = Coffee | Tea | Milk | Beer | Water

data Smoke = PallMall | Dunhill | Blend | BlueMaster | Prince
```

The houses of the problem will make natural containers.

```haskell
type Container = ([Colour],[Man],[Pet],[Drink],[Smoke])
```

Now all the data needed to build the table has been deduced from the clues the typeclasses for the types in the Container must be instantiated and the functions that need to pattern match the Container can be defined. This is a tedious process but because of its mechanical and predictable nature it can be automated.
**Defining Rules**

Now the rules of the problem can be deduced from the problem. This should be done with the knowledge that some rules, whether they are needed or not, are already implemented.

| The English man lives in the red house. | pair Eng Red |
| The Swede has a dog. | pair Swe Dog |
| The Dane drinks tea. | pair Dan Tea |
| The green house is immediately to the left of the white house. | relative Green White 1 |
| They drink coffee in the green house. | pair Coffee Green |
| The man who smokes Pall Mall has birds. | pair PallMall Birds |
| In the yellow house they smoke Dunhill. | pair Yellow Dunhill |
| In the middle house they drink milk. | direct Milk 2 |
| The Norwegian lives in the first house. | direct Nor 0 |
| The man who smokes Blend lives in the house next to the house with cats. | nextTo Blend Cats |
| In a house next to the house where they have a horse, they smoke Dunhill. | nextTo Horse Dunhill |
| The man who smokes Blue Master drinks beer. | pair BlueMaster Beer |
| The German smokes Prince. | pair Ger Prince |
| The Norwegian lives next to the blue house. | nextTo Nor Blue |
| They drink water in a house next to the house where they smoke Blend. | nextTo Water Blend |

There is a rule that is in this problem that was not in the last. The nextTo rule means that the two elements are beside each other but does not specify a direction. This is a new rule and will have to be implemented with the type of [Container] → [Container] so it can be composed with the other rules.

**NextTo**

The nextTo rule needs to be implemented to solve this problem. This involves finding out when elements can be removed because of the rule. Again the rule is similar to the pair rule but has a major difference because there is a choice for one of the elements when checking. In the noextTo function there are local functions that are in a where that can check a given element to the left and right and one, using the prior two, a function that checks both sides.

The function nextCheck does not take any arguments. Instead it works in the namespace of the function as it is declared in a where statement. It returns a tuple where the left holds a boolean that shows if b is present either side. If it is false a can be removed from this Container. Similarly the other side of the tuple for the removal of b. This function is used in the case that neither element is fixed.
nextCheck = (leftCheck b || rightCheck b, leftCheck a || rightCheck a)
leftCheck :: (Contained el, Eq el) => el -> Bool
leftCheck | index > 0 = \ el -> elem el $ get prev
| otherwise = \ el -> False
rightCheck :: (Contained el, Eq el) => el -> Bool
rightCheck | index < (length week) - 1 = \ el -> elem el $ get next
| otherwise = \ el -> False

The function is split into worker and header functions. The header function initiates
the index value for the worker function, which is incremented on recursion. This is used to set the
Container values next and prev by indexing into the whole of the list of Containers.

Guards separate functionality for when one of the elements is solved and when neither are
solved. This is the same as the other rules but in the cases where the element is solved this rule is
more complex. This is due to the element of choice that the rule has. To fix the other element if one
is fixed the other has to have only one place that it can go. Otherwise it cannot be fixed yet.

**Solving**

All the rules for the problem now have functions that can implement them on the table.
Since the new one now has a function. Now the puzzle can be solved in the same way as before.

The solve function has been slightly adjusted because there is no important reason to
distinguish between the pair, relative and nextTo rules in the solver. This is because they are all the
iterated rules and are only used together anyway.
Chapter 7

Conclusions

The project was able to achieve many of it's aims. A framework was created, for these types of problems, that was scalable and reusable. The resulting programs of the project are able to find the answers for these problems, with unique answers, in a reasonable time. This is the most important aim as the nature of these problems scale exponentially in size as more elements can mean that there is a lot more combinations available.

The use of reduction as a methodology to solve these problems, instead of generate and test, allows the problem to be solved quicker and in a much more scalable way. This is because reduction takes impossible elements out instead of testing combinations. This is a more efficient way of reducing the possible combinations left because for each element removed from a location there are a group of combinations that have that element there.

This methodology mapped well to the use of a functional language because of the rules' similarity to having functions working over the table. Haskell's guards make the code more human literate by easily grouping different cases of the rules and providing the actions for those cases reading from left to right. This is a huge advantage for maintaining the code especially because of the usually unclear nature of programming logical edge cases in general.

The rules are generalised with the use of typeclasses that have instances for the types of the elements that are in the general containers. With the types of the containers not used directly in the rule functions they can be reused for different problems where the containers can be different sizes.

Future Work

Although the aims were met there are a few areas that could be improved to improve the system. One obvious omission is the lack of support for multiple answers which is important if this methodology is to be used for something more substantial in constraint logic. The rules are reusable but the way it is done is not very good. The typeclasses are automatable but does have to be defined and compiled for every problem. Depending on the domain this functionality could be sufficient but regardless it can be improved on.

That inability to deal with multi answer problems is important because if the system was to be used in something like artificial intelligence it would have to show multiple answers because there could be multiple solutions. This functionality would be reasonably simple as the current solve reduces the table as far as it can. From the resulting table more tables could be generated by simply fixing an element in different positions or fixing the elements of a rule where that rule is possible. These tables could than be solved by the existing solve but with functionality for failure added. It would be necessary to deal with failure of solves because not all elements would be removed if it was certain any more because of the relatively arbitrary permutation outside of solve.

The container could be improved with the use of an iterable data structure and using generalised algebraic datatypes. The container cannot be implemented in a Prelude Haskell list because the elements within it would not all be the same type. It is conceivable that a list could be
created that would hold the types of the elements in it that would be used as the container. This means that the container could be changed in the program. It would involve a lot more effort what is the current typeclass Contained if not a complete overhaul of that area of the system. This was briefly attempted but my knowledge of Haskell's language extensions is not sufficient to create a data structure that can encompass this functionality.
Appendices
import Data.List

compose :: [a -> a] -> a -> a
compose fs = foldl (flip ()) id fs --From haskell.org/haskellwiki/Compose

data Man = Paul | Rob | Stan | Vern | Wally
    deriving (Eq, Show, Enum, Bounded)

data Woman = Anne | Cathy | Eve | Fran | Ida
    deriving (Eq, Show, Enum, Bounded)

type Day a b = ([a], [b])

showweek' :: (Show a, Show b) => [Day a b] -> String
showweek' [] = []
showweek' ((alist, blist):tail) = (show alist) ++ (show blist) ++ "\n" ++ (showweek' tail)

showweek :: (Show a, Show b) => [Day a b] -> IO()
showweek = putStrLn . showweek'

men = [minBound ..] :: [Man]
women = [minBound ..] :: [Woman]
theweek = take 5 $ repeat (men, women)

direct1 :: Eq a => a -> Int -> [Day a b] -> [Day a b]
direct1 _ [] = []
direct1 _ (index ((manlist, wm):t)) = ((if index == 0
    then ([man], wm)
    else (delete man manlist, wm))
    : direct1 man (index - 1) t)

direct2 :: Eq b => b -> Int -> [Day a b] -> [Day a b]
direct2 _ [] = []
direct2 _ (index ((mn, wm):t)) = ((if index == 0
    then (mn, [woman])
    else (mn, delete woman wm))
    : direct2 _ (index - 1) t)

rm1 :: Eq a => a -> Int -> [Day a b] -> [Day a b]
rm1 _ [] = []
rm1 _ (index ((manlist, wm):t)) = ((if index == 0
    then (delete man manlist, wm)
    else (manlist, wm))
    : rm1 _ (index - 1) t)

rm2 :: Eq b => b -> Int -> [Day a b] -> [Day a b]
rm2 _ [] = []
rm2 _ (index ((mn, wm):t)) = ((if index == 0
    then (mn, delete woman wm)
    else (mn, wm))
    : rm2 _ (index - 1) t)

data Val a = T a | F a
    deriving (Show)

pair :: (Eq a, Eq b) => Val a -> Val b -> [Day a b] -> [Day a b]
pair _ [] = []
pair va@(T a) vb@(T b) ((alist, blist):tail) | alist == [a] = (((a),[b]):tail)
propagate [] = []
propagate rm@(rmalist, rmblist) ((alist, blist):tail) = length alist == 1 && length blist == 1 = ((alist, blist): propagate rm tail)

| list == [b] = ([], [b]):tail |
| otherwise = case (aval, bval) of |
| (True, True) -> ([], [b]):pair va vb tail |
| (True, False) -> ([], [b]):pair va vb tail |
| (False, True) -> ([], [b]):pair va vb tail |
| (False, False) -> ([], [b]):pair va vb tail |

where aval = elem a alist |
| bval = elem b blist |
pair va@(T a) vb@(F b) ((alist, blist):tail) = a == [b] = ([], [b]):pair va vb tail |
pair va@(T a) vb@(F b) ((alist, blist):tail) = a == [b] = ([], [b]):pair va vb tail |
pair va@(F a) vb@(T b) ((alist, blist):tail) = a == [b] = ([], [b]):pair va vb tail |
pair va@(F a) vb@(T b) ((alist, blist):tail) = a == [b] = ([], [b]):pair va vb tail |

solved [] = [[]] |
solved ((alist, blist):tail) = length alist == 1 && length blist == 1 = (alist ++ al, blist ++ bl) |

| length alist == 1 = (alist ++ al, blist ++ bl) |
| length blist == 1 = (alist, blist ++ bl) |
| otherwise = (al, bl) |

where (al, bl) = solved tail |

marriage_solver.hs
propagateSolved x = propagate (solved x) x

repeatedApply funct days = let newdays = funct days
  in if newdays == days
    then propagateSolved newdays
    else repeatedApply funct $ propagateSolved newdays

--Remains a single table.
relative _ _ [] = []
relative (T a) (T b) offset week = overlapRemove offset $ removeFalseRelation (filter (not. checkRelation) (take (length week) [0..]))

  where
    checkRelation index | index + offset >= 0 &&
      index + offset < length
      week = case week !=
        index of

      (alist, _) -> if elem a alist

        then case week != (index + offset) of

          (_ , blist) -> if elem b blist

            then True

            else False

        else False

  removeFalseRelation [] = week
  removeFalseRelation (index:tail) = rm1 a
  index $ rm2 b (index + offset) $ removeFalseRelation tail
  overlapRemove offset week | offset > 0 =
    compose (map (\x -> rm1 a x) [0..(offset-1)])
    week
  | offset < 0 =
    compose (map (\x ->
      rm2 b x) [(length
      week) - 1]..((
      length week)-1-
      offset)])
    week
  | otherwise = week

directs = direct2 Anne 0 . direct1 Stan 2 . direct1 Rob 4
pairs = pair (F Wally) (T Anne) . pair (T Vern) (T Fran) . pair (T Rob) (F Ida)
relatives = relative (T Vern) (T Eve) (-1)
solve directs pairs relatives week = let afterDirects = directs week
  in iterate afterDirects
  where iterate week = let after = propagateSolved $ relatives $ pairs week
    in if after == week
      then after
      else iterate after
solveMarriage = solve directs pairs relatives theweek
import Data.List hiding (insert)

compose :: [a -> a] -> a -> a
compose fs = foldl (flip ()) id fs --From haskell.org/haskellwiki/Compose

data Man = Paul | Rob | Stan | Vern | Wally
  deriving (Eq, Show, Enum, Bounded)

data Woman = Anne | Cathy | Eve | Fran | Ida
  deriving (Eq, Show, Enum, Bounded)

type Container = ([Man], [Woman])
emptyCont = ([],[])

solved :: Container -> Container
solved (a, b) = let x = ifSingle a
               in let y = ifSingle b
                    in (x, y)
               where
                  ifSingle [el] = [el]
                  ifSingle _ = []

contUnion (a1, b1) (a2, b2) = (a1++a2, b1++b2)

propagate :: Container -> [Container] -> [Container]
propagate (aList, bList) = compose (concat [map safeRemove aList, map safeRemove bList])

findSolved [] = emptyCont
findSolved (cont:tail) = solved `contUnion` (findSolved tail)

propagateSolved x = propagate (findSolved x) x

class Contained a where
get :: Container -> [a]
safeRemove :: (Contained a, Eq a) => a -> [Container] -> [Container]
instance Contained Man where
  get (men, _) = men
  insert manList (_, women) = (manList, women)
instance Contained Woman where
  get (_, women) = women
  insert womenList (men, _) = (men, womenList)

remove :: (Contained a, Eq a) => a -> Container -> Container
remove el cont = insert (delete el (get cont)) cont

safeRemove :: (Contained a, Eq a) => a -> [Container] -> [Container]
safeRemove _ [] = []
safeRemove a (cont:tail) | get cont == [a] = (cont: safeRemove a tail)
  | otherwise = (remove a cont: safeRemove a tail)

men = [minBound ..] :: [Man]
women = [minBound ..] :: [Woman]
theWeek = take 5 $ repeat (men, women)

direct :: (Contained a, Eq a) => (Val a) -> Int -> [Container] -> [Container]
direct _ _ [] = []
direct (T el) index (cont:t) = ((if index == 0
          then insert [el] cont
          else cont) ++ t)
getable_marriage_solver.hs

define (F el) index (cont:t) = (if index == 0
    then remove el cont
    else cont
    : direct (F el) (index - 1) t)

data Val a = T a | F a

deriving (Show)

pair :: (Contained a, Eq a, Contained b, Eq b)
    => Val a -> Val b -> [Container] -> [Container]
pair _ _ [] = []
pair va@(T a) vb@(T b) (cont:tail) | get cont == [a] = (insert [b] cont:tail)
    | get cont == [b] = (insert [a] cont:tail)
    | otherwise = case (val, val) of
        (True, True) => (cont:pair va vb tail)
        (True, False) => (remove a cont:pair va vb tail)
        (False, True) => (remove b cont:pair va vb tail)
        (False, False) => ((remove a $ remove b cont):pair va vb tail)
    where val = elem a $ get cont
        bval = elem b $ get cont

pair va@(T a) vb@(F b) (cont:tail) | get cont == [a] = (remove b cont:pair va vb tail)
    | get cont == [b] = (remove a cont:pair va vb tail)
    | otherwise = (cont: pair va vb tail)

pair va@(F a) vb@(T b) (cont:tail) | get cont == [a] = (remove b cont:pair va vb tail)
    | get cont == [b] = (remove a cont:pair va vb tail)
    | otherwise = (cont: pair va vb tail)

pair va@(F a) vb@(F b) (cont:tail) | notInCont a = (remove b cont:tail)
    | notInCont b = (remove a cont:tail)
    | otherwise = (cont: pair va vb tail)
    where notInCont el = not $ elem el $ get cont

--Remains a single table.
relative :: (Contained a, Eq a, Contained b, Eq b)
    => Val a -> Val b -> Int -> [Container] -> [Container]
relative _ _ [] = []
relative va@(T a) vb@(T b) offset week
    | offset == 0 = pair va vb week
    | offset > 0 = let notOverlap = applyFunctionList (map \x -> direct (F b) x) [0..(offset-1)]
        in relCheck notOverlap

    where
        applyFunctionList [] a = a
        applyFunctionList (f:tail) a = f (applyFunctionList tail a)
        relCheck [] = []
        relCheck (cont:tail) | length tail < offset = (remove a cont:tail)
            | get cont == [a] = (cont
then if bval <= aval

    then (cont: relCheck tail)

    else (cont:(relCheck $ direct (F b) (offset-1) tail))

else (remove a cont:relCheck tail)

where aval = elem a $ get cont

    bval = elem b $ get (tail !! (offset - 1))

directs = direct (T Anne) 0 . direct (T Stan) 2 . direct (T Rob) 4
pairs = pair (F Wally) (T Anne) . pair (T Vern) (T Fran) . pair (T Rob) (F Ida)
relatives = relative (T Eve) (T Vern) 1

solve directs pairs relatives week = let afterDirects = directs week
    in iterate afterDirects

where iterate week = let after = propagateSolved $ relatives $ pairs week
    in if after == week
        then after
        else iterate after

solveMarriage = solve directs pairs relatives theweek
import Data.List hiding (insert)

compose :: [a -> a] -> a -> a
compose fs = foldl (flip ()) id fs --From haskell.org/haskellwiki/Compose

ppList :: Show a => [a] -> IO()
ppList = putStrLn . ppList'
ppList' [] = []
ppList' (head:tail) = (show head) ++ "\n" ++ (ppList' tail)

data Colour = Red | Green | Blue | Yellow | White
  deriving (Eq, Show, Enum, Bounded)

data Man = Eng | Swe | Dan | Nor | Ger
  deriving (Eq, Show, Enum, Bounded)

data Pet = Dog | Birds | Cats | Horse | Zebra
  deriving (Eq, Show, Enum, Bounded)

data Drink = Coffee | Tea | Milk | Beer | Water
  deriving (Eq, Show, Enum, Bounded)

data Smoke = PallMall | Dunhill | Blend | BlueMaster | Prince
  deriving (Eq, Show, Enum, Bounded)

type Container = (Container, Container, Container, Container, Container)
emptyCont = (emptyCont, emptyCont, emptyCont, emptyCont, emptyCont)

colours = [minBound..] :: Colour
men = [minBound..] :: Man
pets = [minBound..] :: Pet
drinks = [minBound..] :: Drink
smokes = [minBound..] :: Smoke
theStreet = take 5 $ repeat (colours,men,pets,drinks,smokes)

solved :: Container -> Container
solved (a,b,c,d,e) = let v = ifSingle a
  in let w = ifSingle b
    in let x = ifSingle c
      in let y = ifSingle d
        in let z = ifSingle e
            in (\v w x y z -> (v,w,x,y,z)) v w x y z
    where
      ifSingle [el] = [el]
      ifSingle _ = []

contUnion :: Container -> Container -> Container
contUnion (a,b,c,d,e) (w,v,x,y,z) = (a++w,b++v,c++x,d++y,e++z)

findSolved [] = emptyCont
findSolved (cont:tail) = solved cont `contUnion` findSolved tail

propagate :: Container -> [Container] -> [Container]
propagate (aList, bList, cList, dList, eList) = compose (concat [map safeRemove aList, map safeRemove bList, map safeRemove cList, map safeRemove dList, map safeRemove eList])

propagateSolved x = propagate (findSolved x) x
```haskell
class Contained a where
get :: Container -> [a]
insert :: [a] -> Container -> Container

instance Contained Colour where
get (colors, _, _, _, _, _) = colors
insert colorIns (_,man,pet,drink,smoke) = (colorIns,man,pet,drink,smoke)

instance Contained Man where
get (_,mans, _, _, _, _) = mans
insert manIns (color,_,pet,drink,smoke) = (color,manIns,pet,drink,smoke)

instance Contained Pet where
get (_,_,_,pets, _, _) = pets
insert petIns (color,man,_,drink,smoke) = (color,man,petIns,drink,smoke)

instance Contained Drink where
get (_,_,_,_,drinks, _) = drinks
insert drinkIns (color,man,_,_,smoke) = (color,man,pet,drinks,smoke)

instance Contained Smoke where
get (_, _, _, _, _,smokes) = smokes
insert smokeIns (color,man,pet,drink,_,) = (color,man,pet,drink,smokeIns)

remove :: (Contained a, Eq a) => a -> Container -> Container
remove el cont = insert (delete el (get cont)) cont

safeRemove :: (Contained a, Eq a) => a -> [Container] -> [Container]
safeRemove _ [] = []
safeRemove a (cont:tail) | get cont == [a] = (cont: safeRemove a tail)
| otherwise = (remove a cont: safeRemove a tail)

direct :: (Contained a, Eq a) => (Val a) -> Int -> [Container] -> [Container]
direct _ _ [] = []
direct (T el) index (cont:t) = ((if index == 0
  then insert [el] cont
  else remove el cont)
  : direct (T el) (index - 1) t)

direct (F el) index (cont:t) = ((if index == 0
  then remove el cont
  else cont)
  : direct (F el) (index - 1) t)

data Val a = T a | F a
  deriving (Show)

class pair :: (Contained a, Eq a, Contained b, Eq b)
              => Val a -> Val b -> [Container] -> [Container]
class pair _ _ [] = []
pair va@(T a) vb@(T b) (cont:tail) \ get cont == [a] = (insert [b] cont:tail) -- Check for b
  | get cont == [b] = (insert [a] cont:tail) -- Check for a
  | otherwise = if aval <= bval
      then if bval <= aval
      then (cont:pair va vb tail)
      else (remove b cont: pair va vb tail)
    else (remove a cont:pair va vb tail)
      where aval = elem a $
get cont
  bval = elem b $ get cont

pair va@(T a) vb@(F b) (cont:tail) | get cont == [a] = (remove b cont: pair va vb tail)
  | get cont == [b] = (remove a cont: pair va vb tail)
  | otherwise = (cont: pair va vb tail)

pair va@(F a) vb@(T b) (cont:tail) | get cont == [a] = (remove b cont: pair va vb tail)
  | get cont == [b] = (remove a cont: pair va vb tail)
  | otherwise = (cont: pair va vb tail)

pair va@(F a) vb@(F b) (cont:tail) | notInCont a = (remove b cont:tail)
  | notInCont b = (remove a cont:tail)
  | otherwise = (cont: pair va vb tail)
    where notInCont el = not $ elem el $ get cont

--Remains a single table.
relative :: (Contained a, Eq a, Contained b, Eq b)
  => Val a -> Val b -> Int -> [Container] -> [Container]
relative _ _ _ [] = []
relative va@(T a) vb@(T b) offset week | offset == 0 = pair va vb week
  | offset > 0 = let notOverlap = applyFunctionList (map (\x -> direct (F b) x) [0..(offset-1)]) week
           in relCheck notOverlap
             | applyFunctionList [] a = a
             | applyFunctionList (f:tail) a = f (applyFunctionList tail a)
             | relCheck [] = []
             | relCheck (cont:tail) | length tail < offset = (remove a cont:tail)
             | get cont == [a] = (cont:direct (T b) (offset-1) tail)
             | otherwise = if
            aval <= bval

then if bval <= aval

  then (cont: relCheck tail)

  else (cont:(relCheck $ direct (F b) (offset-1) tail))

else (remove a cont:relCheck tail)

  where   aval = elem a $ get cont

          bval = elem b $ get (tail !! (offset - 1))

nextTo :: (Contained a, Eq a, Contained b, Eq b)
  => Val a -> Val b -> [Container] -> [Container]
nextTo va vb week = nextTo' va vb week 0
nextTo' va@(T a) vb@(T b) week index | index == length week = week
  | (get cont) == [a] = case (leftCheck b, rightCheck b) of
      (True,True) ->
      nextTo' va vb week
zebra_solver.hs

| (get cont) == [b] = case (leftCheck a, rightCheck a) of
|   (True,True) -> nextTo' va vb week (index + 1)
|   (True,False) -> nextTo' va vb (direct (T b) (index -1) week) (index + 1)
|   (False,True) -> nextTo' va vb (direct (T b) (index +1) week) (index + 1)
|   otherwise = case nextCheck of
|     (True,True) -> nextTo' va vb week (index + 1)
|     (True,False) -> nextTo' va vb (direct (F b) index week) (index + 1)
|     (False,True) -> nextTo' va vb (direct (F a) index week) (index + 1)
|     otherwise = nextTo' va vb (direct (F b) index $ direct (F a) index week) (index + 1)

where
  cont = week !! index
  next = week !! (index + 1)
  prev = week !! (index - 1)
  nextCheck = (leftCheck b || rightCheck b, leftCheck a || rightCheck a)
  leftCheck :: (Contained el, Eq el) => el -> Bool
  leftCheck | index > 0 = \ el -> elem el $ get prev
              | otherwise = \ el -> False
  rightCheck :: (Contained el, Eq el) => el -> Bool
  rightCheck | index < (length week) - 1 = \ el -> elem el $ get next
              | otherwise = \ el -> False

solve directs repeats week = let afterDirects = directs week
  in iterate afterDirects
  where iterate week = let after = propagateSolved $ repeats week
                     in if after == week
directs = direct (T Milk) 2 . direct (T Nor) 0
pairs = pair (T Eng) (T Red) . pair (T Swe) (T Dog) . pair (T Dan) (T Tea) . pair (T Coffee) (T Green) . pair (T PallMall) (T Birds) . pair (T Yellow) (T Dunhill) . pair (T BlueMaster) (T Beer) . pair (T Ger) (T Prince)
relatives = relative (T Green) (T White) 1 . nextTo (T Blend) (T Cats) . nextTo (T Horse) (T Dunhill) . nextTo (T Nor) (T Blue) . nextTo (T Water) (T Blend)
solveZebra = solve directs (relatives.pairs) theStreet