CITS3242 Project Report – Part 1

Author: Ash Tyndall, 20915779. Partner: Nic Barbaro, 20502247.

Our solution to part 1 employs a variety of F#'s functional features to allow successful experiment sufficiency testing with few caveats.

Program Structure

Our program is structured with three core functions, `suffices`, `unify` and `reduceTransitive`, along with many smaller helper functions.

Suffices

The `suffices` function forms the front-end of the solution, as it is the external function that part 2 of the project will call to determine the exact circumstances that the queued experiments can suffice for other experiments. The function uses recursion to construct its arguments, and implements the maximum rule depth to prevent infinite recursion.

The first primary step the function goes through is calculating the `evaluated` list comprehension. This list comprehension is a `rule list option` that both evaluates the provided `unit->rule` rules list and removes any non-transitive cases from the sufficiencies list (more discussion later). It does this by constructing a set of all variables seen in the sufficiencies and not in the main rule and passing that set to `reduceTransitive`. An option is used in the case that the transitive reduction fails due to a hanging variable; the deletion of a `None` in the list will cause an graceful termination.

The second primary step involves running each evaluated rule against the `checkRule` function. This function uses the `unify` function to determine if a mapping is possible between the rule and the given experiment, as well as recursively calling `suffices` to the sub-sufficiencies that may exist. A fold with `&&` is used to ensure that all of these sub-sufficiencies are true.

Finally, after performing these two steps, a fold with `||` is used to ensure that all of the `checkRule` executions are true.

Unify

The `unify` function forms the heart of the solution, as it is the function responsible for attempting to create a set of variable substitutions that make two experiments equivalent ("unification"). Much like `suffices`, it is implemented as a recursive function, although it much more heavily uses the F# pattern matching techniques.

At its core, `unify` reduces the problem of mapping two experiments to a small about of base cases that assume that all variables are found on the right-hand side of both experiments. Following extensive thinking and testing, we determined that a map would be most suited to storing the unification result, and a set of cases where recursion or adding to the map was required.

To determine of the unification as somehow failed, we return a `map option` and use a specially written helper function `mMerge` to determine if two unification approaches are incompatible, and to

propagate `None` results up the function tree when necessary. We consider two unifications incompatible when passed to `mMerge` if they both attempt to map the same key to a different value.

Reduce Transitive

The `reduceTransitive` is one of the more interesting functions of the codebase, and is used to handle the case in which sub-sufficiencies have an unknown that connections any number of experiments together. This implementation represents a misunderstanding of the problem (see Caveats section), however it performs its intended function well.

This function accepts a list of sub-sufficies and a set of variables that are considered to be "unknown" (U) and need to be removed from the set. Initial design of this function went in several directions, but our ultimate approach is based on directed graph theory and recursion.

The function attempts to reduce the sub-sufficiencies into a non-transitive version by considering each pair of variables (a, b) to both logically mean $a \Rightarrow b$ and to form the edge (a, b) on a acyclic graph representing these implications of which a and b are nodes.

In the case that (a, b) where $a \wedge b \in U$, we can say that $a = b$ and replace any instances of b with a in the sufficiencies and remove b from U. We accomplish this with a simple helper function that replaces all references to a given variable with another in the sufficiencies.

In the case that $(a, b), (b, c)$ where $b \in U$, we can say that $a = c$ and replace those two tuples with (a, c). We discovered a particularly interesting way to perform this function over the entire sufficiency list simultaneously removing one variable: If we had a list of all variables (represented as nodes) that had edges into a given unknown variable, and the same list for edges that left that unknown variable's node, an equivalent set of (v, e) edges that maintained that same implications without the unknown variable was equal to the cross-product of those two lists.

Interesting Features

A common convention we adopted in our code when recursion with mandatory state parameters was required (as in two of the core functions) was overriding the local scope with a curried version of the recursive function with the mandatory parameters already placed within in. This reduced code clutter and increased code readability dramatically.

An interesting discovery made was that the && and || operations are actually functions that could be passed into the `Seq.fold` function. This allowed succinct collapsing of list comprehensions to one value in several places.

Also, because we were detailing with option types frequently, we wrote several helper functions and an operator (see option helpers section) to assist us with handling the Some/None values in an appropriate and succinct manner.

Program Performance and Caveats

Our final solution to part 1 passed all but one of the original tests included. This was due to a misinterpretation of the case of unknown variables in sub-sufficiencies. We had initially assumed that in the case of an unknown, logic could be used to turn it into some set of sub-sufficiencies

comprised entirely of known values. In the case of a hanging unknown variable (i.e. it is only found on one side of expressions), we would return false in `suffices`. This assumption was false, and due to time constraints, could not be rectified before project submission.

However we do think that our approach to resolution does represent a functional approach to a similarly interesting problem, thus we have left that code in our solution and included a slightly modified set of test cases to test it against.

In all cases, our code fails gracefully, using the appropriate option types when needed.

Program Development

Owing to the unique logic required in the problem, and the newness of functional programming, the steps we followed in developing our solution were mainly pair-based discussion of the problem and paper based walk-through of potential solutions.

For all three of the core problems, we initially had difficulty determining what recursion cases were needed, and what had to be done in those cases. To solve this issue, we sat down with a pen and paper together and attempted to firstly determine what the terminating case was, and then begin to build the program up from that.

Reflection

I found this project challenging, but not insurmountable. Being new to functional programming made the beginnings of the project slow, but once I learned more about F#, programming became productive. The project (although we're still in the midst of it) seems to be well suited to the given deadline, and was both interesting and a good introduction to the power and features of functional programming.

I think we should have clarified the transitive case more early on, as we went off in the wrong direction without enough time. But other than that the project was done very well.