F28PL - Programming Languages

Question D (Essay-style, functional programming)

Tommy Lamb

H00217505

1. Explain *recursion*.

   Recursion is where a section of code in normal operation causes itself to be executed with a different state or dataset. With respect to functional programming it is where a function makes a function call to itself using different argument values. Any recursive function must also have a base case, some set of arguments which does not result in a recursive function call, lest the program never end. Recursion for many applications is considered easier to understand and more intuitive than for-next or while loops as a method of iteration; however compilers - at least traditionally - are unable to optimise recursion to the same level.

2. Explain *tail recursion* and how it differs from plain old recursion. If you want to write efficient code, which is better, and why?

   Tail recursion differs from standard recursion in that the recursive call is the last computational activity carried out by the calling function. In using it the initial function has completely finished executing by the time the recursive call is made, and so no state information has to be held in the call stack. This means that regardless of how often the function is called, the stack will take up a tiny amount of memory compared with plain old recursion. Naturally this is desirable, and leads to more efficient code in terms of memory usage.

3. Explain the *Church-Rosser property*, and how it is relevant to programming language execution.

   The Church-Rosser property describes something where the order of execution/evaluation is irrelevant. In relation to programming languages it describes whether the various individual lines of code, functions, methods, and what-not have to be executed in the order programmed, or can be executed in any order and produce the same end result. Due to the direct manipulation of memory, imperative languages are decidedly **not** Church-Rosser. Declarative languages on the other hand are inherently Church-Rosser, as they describe what the system should do, but not how. This allows programs to be chopped up and pushed around by compilers and systems much more easily as they needn't worry about changing the result of computation, and leads to far superior parallelisation. This is a boon given the trend for more cores/execution units per CPU as we approach the limits of single-core technology (transistor size leading to quantum effects, heat, clock speed, etc). With a Church-Rosser language every single CPU core/thread can be utilised, all running different instructions without worrying about conflicts or which instruction finishes first.

4. The cost of running a calculation is calculated as follows:
   $programmercosts + computingcosts = totalcosts$. A more detailed equation is
   $designcosts + programmercosts + debuggingcosts + computingcosts = totalcosts$.
   With these equations in mind, describe some of the trade-offs involved in choosing a programming language for an embedded chip in a car versus an Android app.

   - Design Costs:
     A declarative language will have comparatively low design costs compared to that of an imperative language, as the intricate details of execution and side-effects are completely removed from the programmers control. Conversely an imperative language will have slightly higher costs, as each sub-program must be designed very closely to prevent unintended side effects of derailing the software as a whole. This is a consequence of the lack of the Church-Rosser quality.

     What with all the graphical bells and whistles the design costs of an Android app are already rather high, without requiring the additional burden of designing each sub-routine flawlessly. A declarative language would help limit the cost of app design, which means it can be sold for cheaper, which means more people buy it, which means more turnover, which means more profit. The embedded car system however (presumably) has no fancy graphical interface, nothing to design other than the code itself around the specified input and output. Choosing between one paradigm and the other would fall to other arguments here, as either are acceptable.

   - Programmer Costs:
     Any programmer who knows anything knows how to program imperatively. So in choosing an imperative language the developers have their choice of who to hire and fire. Declarative languages on the other hand are less widely known, and the pool of potential programmers smaller. This means that declarative-competent programmers can demand something of a premium compared with imperative-competent programmers (as they are a somewhat rarer breed). Counter to this is the fact that programmers tend to produce the same amount of code lines per unit time regardless of language. With the abstract nature of Declarative languages rendering each line more powerful, more functional code can be produced per unit time in Declarative than Imperative languages. With the same work taking less time and/or less programmers, the Declarative programmer premium may well be negated, or completely reversed.

     In the fast paced world of mobile apps, there's no time to lose in getting an idea out there before some early bird steals that particular worm. The fastest and easiest programmers to acquire are Imperative-trained as they practically grow on trees, allowing a new software house to open relatively quickly. The automobile market is a very different beast. New car models are in the design stage for years, and are the products of large, rich multi-national companies which can afford to be more selective - both financially and time-wise. With money and time whichever route they pick will make only a limited difference.

- Debugging Costs:
  As well as Declarative languages doing more work per line of code than in Imperative, the number of bugs introduced by any given programmer to a project is directly proportional to the number of lines of code produced. With less lines of code it logically follows that any given Declarative project will have fewer bugs than an equivalent Imperative project and thus the time and financial costs of debugging a declarative program are significantly lower.

  When creating an Android app, no-one really cares *that* much about debugging; as long as the app is more than half finished, release it and patch a few things every month or so, whenever the team get around to it. This 'meh' approach to bugs arises because the worst that can happen is someone has to restart their phone, grumble about the app, and go straight back to using it anyway. Declarative languages don't have much sway here as a result. Cars are completely different: something goes wrong, and people can and do die. Recalls are expensive and extremely brand-damaging: locating 100-something million vehicles around the globe and convincing the owners to abandon it for the day at a garage is no mean feat. So when making (control) software for a car you simply *must* know that it'll work properly and flawlessly. In this environment declarative languages are something of a no-brainer.

- Computing Costs:
  As stated more expansively in Q6, the average Declarative program is more efficient than the average Imperative program, however Imperative software can be vastly more optimised when the target environment is well defined. An embedded chip in a car will likely be a cheap, not particularly powerful, single core affair. It also represents a constant, unchanging, and perfect-information environment. This makes it a perfect target for well-optimised Imperative programming to make the most of the available hardware, especially when that hardware can't take advantage of the parallelisation inherent to Declarative programming. The average Android smartphone on the other hand is a complete unknown to the developer, and so optimising for hardware becomes virtually impossible. Many modern smartphones do have multiple core/thread CPUs however, which make them ripe pickings for a properly compiled Declarative app, allowing extra power to be garnered over a single-thread, single-core un-optimisable Imperative application. That being said most people also don't care *that* much about app performance, and Android may not even support Declarative languages for app production.

5. "Pure functional programming has no global state."
    1. Explain what this assertion means
    2. Give one advantage and one disadvantage of using a language without global state.

    This asserts that a purely functional program or function, in having no global state, will produce exactly the same output for the same input, no matter when, where, and how it is run. This is because there are no variables which persist after execution as any variables are local to the function they're defined in. It also means that each function lives inside its own little bubble with the only glimpse of the outside world through the returns of internal function calls and arguments passed to it. The only way to share information and change a function's output is by changing the arguments passed to it, and not using a global variable which it references.

    An advantage is that any time a function is run, it acts entirely predictably and consistently. Since nothing persists from one call to the function to the next it can only act as it did the first time (provided identical arguments) every time. One execution can never indirectly influence the next, deliberately or accidentally.

    A disadvantage is that this makes sharing information between functions and maintaining program-level variables somewhat more clumsy. By only accepting information into a function via arguments it forces a very carefully controlled and strict hierarchy of function calls which can make the end task much harder to achieve. This can also lead to inordinately large amounts of arguments being passed to functions in order for information to disseminate through the program.

6. Explain the different optimisations possible in imperative versus declarative programming, and the implications for space and time efficiency and development cost.

   Declarative programming broadly takes the job of optimising away from the developer and places it at the feet of the compiler. Naturally the programmer can optimise to some extent, like using tail-recursion rather than basic recursion or using better algorithms, however they cannot optimise for the hardware they're using/targeting. This can make Declarative language source code more portable, as the compiler can be designed exactly for the hardware it will run on, which in turn can optimise the Declarative code for that hardware specifically. This means developers largely don't need to worry about what environment it will run in, they just think about the abstract time and space complexity of their algorithms. This means that the average widely distributed declarative program will be at least, if not more, optimised for any given system than Imperative programs, without any developer intervention (making it virtually free). Of course this relies on good compilers being available for the system environment.

   Imperative programs on the other hand leave the lion's share of optimisation to the programmer; with the way Imperative languages work, compilers can do little in optimisation without risking changing the program's operation. With programs that are to be widely distributed this makes optimisation something of a drop in the ocean. Of course the abstract concepts of algorithm growth rate, and space and time complexity are still available, but the hardware optimisation is nigh on impossible: will this run on Windows, MacOS; Intel 4-core, AMD 8-core, ARM 6-core; 8GB RAM, 4GB RAM; AMD RX 470 Graphics card, Integrated Graphics; and so on. However when the exact hardware is known, or even closely approximated, Imperative languages give developers the ability to optimise right down almost to the byte (specific language and hardware dependant), where humans prove more effective than compilers. As a result Imperative optimisation runs the gamut from virtually non-existent to impressively powerful. A good example of the latter are computer games found on old hardware like Spectrums and Mega Drives. With the simple fixed hardware developers often came up with inventive and creative methods for dragging every measure of power from the console, and then gave them names like "blast processing". Of course this comes at a price. Unlike the Declarative compilers human programmers don't work for free, so optimising can become very costly indeed, growing with size of program and number of hardware/software environments optimised for.