COMP2048

Final exam

Student number: 45004830

Name: Bosheng Zhang

1. an example needed that demonstrates Turing machines are more powerful then FSM's e.g. $(a^n)(b^n)$ can't be performed by an FSM since it lacks a memory component.
   The reason is, you have to reach the final state only when number of 'a' and 'b' are equal in the input string. And to do that you have to count both, the number of 'a' as well as number of 'b' but because value of 'n' can reach infinity, it's not possible to count up to infinity using a Finite automata.
   a TM to accept the language $La^nb^n = \{a^nb^n \mid n > 0\}$
   High level description of the TM for $\{a^nb^n \mid n > 0\}$: “on input string w:
   1. if there is unmarked a, do mark the left most a;
   2. scan right till the leftmost unmarked b, if there is no such b then reject, otherwise mark the leftmost b
   3. go stage 1 until no unmarked a left;
   4. check to see that there is no unmarked b left, if there are then reject, otherwise accept the string”

2. Turing machine is a mathematical calculation model that defines an abstract machine. It envisions replacing people with electromechanical read/write heads, some instructions, and a memory that stores intermediate results. These memories also record the state of the calculation. This state-based approach provides us with an intuitive understanding of computers and computing. A limitation of Turing machines is that they do not model the strengths of a specific arrangement well.

   Lambda calculus is an alternative method that will show that all calculations can be deconstructed and derived from a simple concept, and this concept requires the construction of all necessary logic, loops, numbers and recursion to create a Turing complete system. one drawback is some algorithms could not be expressed elegantly in a functional programming style.

   Lambda calculus is an abstract coding scheme, which can express the calculation itself according to the concept of a function without any physical machine assumptions. It allows us to use simpler languages and symbols than Turing machines to solve computing problems, and at the same time better supports parallel computing, advanced programming, and is comparable to Turing machines in computing power. Lambda calculus is more like a high-level language, it is more abstract and algebraic, while Turing machine is more like assembly language, it provides a more concrete and algorithmic method.

3. In λ-Calculus, we define Boolean logic by using two parameters and define that if the first one is selected, it means true, if the second one is selected, then it is false.
   In other words, we may define true T and false F as the following in λ-Calculus:
   T := λx. λy. x == f(x, y) = x
   F := λx. λy. y == f(x, y) = y
   The NOT operation flips the value; thus, we select the opposite value according to the current argument. The inputs are Booleans p and q, e.g. selector functions, then we can define the NOT as an opposing selector function:
   NOT := λp.pFT
   NOT T == (λp.pFT)T = T(FT) = F
   NOT F == (λp.pFT)F = F(FT) = T
   In the body, we can see the Boolean p will select either F if it is true and T if it is false according to our convention for true and false.
   Likewise, we can define **the AND operation** as the selector function that requires both p and q are both true, thus we can represent in λ-calculus as
   AND := λp. λq. pqF
   AND T T = (λp.λq.pqF)TT = T(TF) = T
   AND F T = (λp.λq.pqF)FT = F(TF) = F
   In the body, we can see if p is false, return false because both Booleans need to be true and the value of q is not requirement to make the decision, otherwise return the value of q that will decide the final value of the operation.
   we can also build **the OR operation** based on similar principles as
   OR := λp.λq.pT q
   OR T F = (λp.λq.pTq)TF = (λq.TTq)F = T(TF) = T
   OR F F = (λp.λq.pTq)FF = (λq.FTq)F = F(TF) = F
   We select true if p is true, if it is not true then maybe q is true, thus return q. The value of q will determine the result.

4. The usual recursive form of the factorial function in most procedural languages would be:

```
int factorial (int n) {
    if n == 0
        return 1
     else
        return n*fac(n-1) <-- in programming language
}
```

We can attempt to convert the above procedural 'function' directly to λ-Calculus for which we might (incorrectly) write:

FAC = λn.If ISZERO(n)(1)MUL(n)(FAC(PRED(n))))

on the contrary, since functions in λ-Calculus are anonymous, we can convert the above expression correctly by utilizing the Y combinator while setting Z for ISZERO as

FAC = Y(λf.λn.Z(n)(1)MUL(n)(f(PRED(n)))) = YR

and set R = λf.λn.Z(n)(1) MUL(n)(f(PRED(n))).

For example, if we run to calculate the factorial of 1, the expression will be

FAC 1 = Y(R)1 = R(Y(R))1

Then if we expand the above expression more detailly:

FAC 1 = λf.λn.Z(n)(1)MUL(n)(f(PRED(n)))(Y(λf.λn.Z(n)(1)MUL(n)(f(PRED(n))))) 1
= λn.Z(n)(1)MUL(n)((Y(λf.λn.Z(n)(1) MUL(n)(f(PRED(n))))))(PRED(n))) 1
= MUL(1)(Y(λf.λn.Z(n)(1)MUL(n)(f(PRED(n))))))(PRED(1))
= MUL(1)(R(YR))(PRED(1))
= MUL(1)(R(YR))0

The zero will trigger the base case if condition and return 1, allowing recursion to be simplified to the final result.

Relevant combinators:

Y := λf.(λx. f(xx))(λx. f(xx))

YR = (λx.R(xx))(λx.R(xx)) = R((λx.R(xx)(λx.R(xx)) = R(YR) = R(R(YR)) …

Y combinators demonstrate that even simple systems based on compact notation can not only encode recursion but can be used in conjunction with other combinators to reduce the computation itself to a few fixed modular combinations of functions and their parameters.

PRED - predecessor, the factorial function applied to the predecessor

Z(n) - the predicate is zero

Z := λn.λf.λx.n(λx.F)T

MULT := (λn.λk.λf.n(kf)) → returns the product of two numbers

PRED := (λn.λf.λx.n(λg.λh.h(gh))(λu.x)(λu.u)) → returns the predecessor of a number by subtracting 1

5. **Deutsch Problem**: The problem Deutsch's algorithm tackles can now be stated as follows. Given a block box U_f implementing some unknown function f : {0, 1} → {0, 1}, determine whether f is "constant" or "variable". Here, constant means f always outputs the same bit, i.e. f(0) = f(1), and variable means f outputs different bits on different inputs, i.e. f(0) not= f(1).
   Why? There is an easy way to determine whether f is constant or variable — just evaluate f on inputs 0 and 1, e.g. compute f(0) and f(1), and then check if f(0) = f(1). This naive (classical) solution, however, needs two queries or calls to U_f (e.g. one to compute f(0) and one to compute f(1)). Therefore surely, no more than two queries to U_f is enough to solve this problem. Can we do it with just one query? In classical condition, the answer is no.

   But quantumly, Deutsch demonstrated how to achieve this with just single query. Suppose one measures the first qubit of the output state $|\psi\rangle$ (this qubit marks which term in the superposition we have) with a standard basis measurement $\{|0\rangle\langle 0|, |1\rangle\langle 1|\}$. Show that the probability of outcome 0 or 1 is $|\alpha|^2$ or $|\beta|^2$, respectively, and that in each case, the state collapses to either $|0\rangle|f(0)\rangle$ or $|1\rangle|f(1)\rangle$, respectively. Thus, only one answer f(0) or f(1) can be extracted this way. *The intuition here is that you have entered a superposition of state. Therefore, you have another degree of freedom, you're feeding through possibilities of ones and zeros, rather than just a one or just a zero and that gives you additional information.*

6. 1.

i. 1. To compute simulation and computation, require **lambda calculus;**

2. To run communication, navigation and life support require **finite state machine;**

A finite state machine is a behavior defined by a set of finite or constant states composed of computational models, a starting state or an initial state, the input letters are used to determine an application-specific transition, a transition function, using application regulations or conditions to determine the next state-state transition and input trigger the state transition event.

Lambda calculus is an abstract coding scheme, which can express the calculation itself according to the concept of a function without any physical machine assumptions. It allows us to use simpler languages and symbols than Turing machines to solve computing problems, and at the same time better supports parallel computing, advanced programming, and is comparable to Turing machines in computing power.

ii. FSM defines functions such as acceptors that accept or reject input, classifiers that classify inputs, and transducers that generate output from a given input.

1. We require a finite state machine in communication, because the finite state machine is applicable in the communication interface, where the inputs need to be parsed when the message is received. Here, we utilize the acceptor to parse message, then classifiers to classify the message then transducers to give response.

2. We require a finite state machine in navigation, as finite state machine can parse message, this allows the system to change to the next state and its associated action/behavior.

3. And we can utilize FSM on life support, as we can use finite state machines to represent an (real or logical) object that can exist in a limited number of conditions ("states") and evolve from one state to next according to a fixed set of rules. E.g. the system will trigger the action of releasing oxygen when the oxygen density is lower than a certain amount.

4. In computing simulation and calculation, we use lambda calculus instead of Finite State Machine and Turing Machine. The reason is that the computing power of the FSM is lower than the lambda calculus, because the memory of FSM is limited by the number of states. As well as there is a constraint on preventing bugs and errors, we prefer lambda calculus. As the concept of functional calculation is to center around the problem and reduce it to a pure and precise arrangement of function symbols. And λ-calculus has significant benefits to support large-scale parallel computing, because unsolicited memory /hardware access (such as write operations) is kept to a minimum to decrease the opportunity of simultaneous access, which will cause the program to crash and guarantee that the operation is always well-defined.

iii. As the hardware limit us the system to be a digital computer. Turing's state-based calculation methods as well as the lambda calculus are easily implemented with digital circuits that have produced the modern computers we use today.

As the programming interface is restricted to an input of 280 characters. A state machine is usually a highly compact way of expressing a complex set of rules and conditions and processing various inputs. A state machine can be presented in a bit code beside its procedural equivalent and runs greatly efficiently. And in lambda calculus, λ notation to functions to allow long strings of functions to be composed or chained together that is

both minimal and compact.

iv. 1. To achieve communication, the FSM will read the input data, and classify data, then if classification corresponds to next expected state, we change state and give back information associated with the new state, otherwise we keep current state.

2. To achieve navigation, the input characteristics are related to specific local attributes of the environment (obtained from sensor data), such as straight paths, left and right turns. The FSM allows the spacecraft to automatically follow a series of states/behaviors to reach its destination, first selecting the appropriate local response behavior for each current state, and then detecting the changes in the current context/state, coming with a sequence of states/actions that codes the topological (global) path into the FSM (sequence of states/actions).

3. To achieve life support, we define can define several states/behaviors to be the actions we make and conditions for switching states. For example, if the life support is to maintain oxygen density inside the spaceship, then we can define the condition as if oxygen density is lower certain amount, then switch to release oxygen state.

4. To compute simulation and computation, we just have to write the required function body, and give them input. Note function only accepts the input described by the symbol and produces output, does not change the input and any other variables, and does not produce additional output, such as output to the terminal screen. And lambda calculus is Turing complete, so equivalent to Turing machines

v. The hardware and computational models will interact through type input by a keyboard and output on screen like the desktops we use today. One should notice is in computing the λ-calculus the order principle holds, as well as the terms with the λ symbol, are processed from left to right.

vi. Off-the-shelf solution is designed and developed to satisfy the demands of as many users as possible. That is the reason it only includes the most popular functionalities for specific system. Using it very often means making a compromise, as it is not possible to find all the features you need in a single package, particularly if the system is rather unique. When choosing one among many off-the-shelf solutions available, we need to decide which features we can do without. And I would like to test the efficiency of the solution, since we would like to compute as much as possible within the shortest time possible.

2. **advantage**: provide the way forward by allowing massively parallel computing through qubits in multiple states simultaneously, it will exponential speed ups their computation;
disadvantage: The design, construction, and programming of quantum computers are extremely difficult. As a result, they are weakened by errors in the form of noise, faults, and loss of quantum coherence, which is critical to their operation and yet falls apart before any nontrivial program has a chance to run to completion.

Challenges: As this coherence loss (called decoherence) caused by vibration, temperature fluctuations, electromagnetic waves, and other interactions with the external environment will eventually destroy the computer's exotic quantum properties, the crew must maintain the environment is perfect to run this quantum computer.

Considering the current prevalence of decoherence and other errors, contemporary quantum computers are unlikely to return correct answers to programs with moderate execution times, which cause our mission are unlikely to achieve.