THE UNIVERSITY OF WESTERN AUSTRALIA
SCHOOL OF COMPUTER SCIENCE & SOFTWARE ENGINEERING

CITS3242 Programming Paradigms

Project: Decentralized Experiment Unification

PART 1 DUE DATE: 4pm, Fri 11th October 2013 (15% = 12% code + 3% indiv. report)

PART 2 DUE DATE: 11am, Thu 31st October 2013 (25% = 20% code + 5% indiv. report)

GROUP MEMBERS DUE: 5pm Wednesday September 19th 2013

This is the programming project for CITS3242 Programming Paradigms. It contributes 40% of your final mark - 15% for the first, and 25% for the second part. For each part one fifth of the marks will be based on an individual report.

You should work on the code for this project in groups in two. Both group members should submit the names, student IDs and logins of the group via the cssubmit system by the above date. Students working alone should also indicate this via cssubmit, but those choosing this option be marked exactly the same as pairs.

One member of each group should submit the groups completed code by the above due dates for each part via https://secure.csse.uwa.edu.au/run/cssubmit. The code for part 2 should include code for part 1. Also each group member should separately write and submit an individual report for each part by the same deadlines.

You are expected to have read and understood the School of Computer Science Policy on Plagiarism, available via http://www.csse.uwa.edu.au/departmental/publications/policy.on.plagiarism.html. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the sole effort of your own group.

Background and Overview

Following successful experiments with Large Hadron Collisions, physicists unexpectedly discovered 2 previously unknown particles of possible great importance.[1] This resulted in an international effort to construct a collection of automated labs for studying these particles by combining them in different ways. There are many scientists worldwide with access to use the labs, and each will submit experiments by computer over a network.

Your task is to build a prototype of the software that will be used to coordinate the experiments. This includes the following main features:

- Rules for combining experiments: in many cases running one experiment is sufficient to determine the result of another experiment. Each lab has its own set of rules that determine when the result of one experiment suffices to determine the result of another one - and the rules can depend on checking sufficiency between other experiments. The first part of the project involves writing a function that determines sufficiency between two experiments based on these rules, which will be used to reduce the number of experiments performed in the second part of the project.
- Mutual exclusion: due to the potentially explosive nature of the particles in the labs, it is critical that each experiment complete without even very slight interruptions.
- Decentralization: to spread the role of coordination evenly and securely, the system will consist only of the scientists' computers (clients) and the labs, with a network connecting them. The coordination software will thus run on each of the client computers. Further, a negotiated agreement strictly requires that client computers only be involved in computation and network communication when that

[1] The "physics" in this document was invented just for this project. It is almost certain that it is not accurate.

client has submitted an experiment recently. The client currently using a lab is responsible for maintaining a queue of waiting clients, and will generally hand over to the next client in the queue when it finishes its experiment and will refer future requests for that lab to that next client. But, having referred a request for the lab from another client it shouldn't receive further requests for the same lab and from that client until it performs another experiment.

Starting point provided

The page http://undergraduate.cs.uwa.edu.au/courses/CITS3242/project contains an F# project, as a zip file, that you should use as your starting point. Much of the structure of the program has already been written, allowing you to focus on the key aspects related to the various paradigms studied in this course. This starting point includes:

- Some basic data types, such as the representation of experiments.
- The basic structure of the program, including starting implementations of types/classes for labs and clients. Most of your work will involve completing the coordination parts of the client class, as well as supporting functions for determining which experiments should be run together.
- Some code to help you with debugging - this includes code allows multiple threads to print as events occur, with the interleaving between threads make obvious by using different colours an indentation based on what object the thread is running in. (This was surprisingly helpful in constructing a sample solution.)

Experiments and unification

Experiments are represented by the following data type:

```
type exp = A | B | Mix of exp * exp | Var of string
```

Basic particles and mixing: The two new particles are represented as **`A`** and **`B`**

`Mixing:` An experiment involves introducing some particles, and then "mixing" them together, two at time to form compounds. Mixing is asymmetric – **`Mix (A, B)`** may have a different result from **`Mix (B, A)`**.

Rules and Variables: Due to the nature of the labs, each one has a set of rules that determine when performing one experiment suffices to determine the result of another experiment - allowing just the first experiment to be performed instead of both of them. As an example, a lab might have the following rule:

Mix(A, B) suffices instead of Mix(B,A)

To allow more general rules, they can include variables such as:

Mix(*x*,*y*) suffices instead of *x*

Rules like this can be used for any values of the variables (*x* and *y* in this case) - and each time the rule is used the *x* and *y* may be different . However, rules can also be conditional on other suffices relationships holding. Examples are rules like this:

Mix(*x*,*y*) suffices for *x* if *x* suffices for *y*

Mix(*x*,*y*) suffices for Mix(*xx*, *yy*) if *x* suffices for *xx* and *y* suffices for *yy*.

We can even have rules that use variables that only appear in the conditions. Examples are rules like this:

Mix(*x*,*xx*) suffices for *y* if: *x* suffices for *z* and *z* suffices for *y*

Each time we use such a rule, we must find such a particular *z* that satisfies the rules. Such rules are more difficult because we can't determine what *z* is directly from the two inputs Mix(*x*,*xx*) and *y*.

Further, to avoid trying infinite sequences of rules, there is a fixed number *maxRuleDepth* which limits the application of rules when solving sufficiencies. Using a single rule with no conditions has depth 1. Using a rule with conditions has depth equal to 1 + the maximum depth required to solve the conditions. For your prototype use *maxRuleDepth* = 8, but make sure it can be easily changed.

The first part of the project is to implement a function that determines sufficiency between two experiments for a particular set of rules, as described above. See the function

```
suffices : (unit -> rule) list -> exp * exp -> bool
```

in the starting point code. (The `unit -> rule` is so that rules can generate different variable names each time, see the starting point code, and the hint below.)

Hint: use a type for *substitutions*, which are maps from variable names to experiments. For any two input experiments, and a rule, calculate a substitution that specifies an experiment as the value for each variable in the rule so that the rule matches the two input experiments. (This is generally called *unification* and is an important concept in logic programming and type inference.)

Experiment Unification/Combination: To optimize the time and cost involved in each experiment, each time a lab becomes available, the client taking over coordination of that lab should check the queue of waiting clients for that lab for clients which are *compatible* with its own experiment – i.e., those that suffice for its own experiment. Either one of these or its own experiment should be performed, choosing the one that suffices for the most other experiments in the queue, favouring smaller experiments in the case of ties. (See the starting point code.)

Thus, your main task for this part of the project is to generate all possible combined experiments from the set of experiments in the queue – with the requirement that the experiment of the coordinator (from the front of the queue) is always included.

Decentralized Coordination

The coordination part of the project requires you to write code that will run on each client that allows the clients to coordinate with each other to ensure that only one client submits an experiment at a time, and to communicate their experiments to each other for combination according to the first part.

Your client class should meet the following requirements.

1) There are a certain number of clients, numbered 0, 1, ..., *n*-1, and a certain number of labs, numbered 0, 1, ..., *r*-1 (with $r < k$).

2) Each client will need to use one of the labs from time to time, and only one client is allowed to be using each lab at any particular time (otherwise the labs may explode).

3) When a lab is needed, the client will call the **`doExp`** method on its own instance of the client class, passing the experiment desired.

4) The Client class should locate a lab that is not being used by another client, and submit the experiment. If all labs are busy, it should wait and use the first lab that becomes available, and at that point choose the most efficient combination of it's own experiment with other clients waiting in the queue.

5) While using a lab, a Client should maintain a queue of other waiting clients.

6) When the client is done with the lab, it should pass the lab to the next client in the queue.

7) Each Client should interact directly with the others to locate labs, and wait for them to become available. This interaction should be via methods calls only. **No shared data structures are allowed.** Waiting should be done via the appropriate monitor mechanisms. **Busy waiting, or periodic polling are not considered appropriate.**

8) Each Client should keep track of which client holds which lab, but should only update this information when the Client is involved in the lab being allocated to a different client. When a lab is reallocated and a Client is not involved, its information will thus be out of date.

9) When a Client receives a request from its client, it should contact the Client of the holder of each lab, according to its information, to request the lab. If the contacted Client no longer holds the lab, it should forward the request on to the holder according to its information. When eventually the lab is located, the original client's Client should be informed when the lab becomes available, and the lab should be allocated to that client if no other lab has become available first.
10) When a Client obtains a lab, it should cancel its requests for the remaining labs.
11) When a Client obtains a lab, the Client releasing it should update its information on who holds the lab. Conversely, if a request is canceled, the original requester should update its information. In either case, any Client involved in forwarding the request should also update its information.
12) Initially client 0 holds lab 0, and 1 holds 1, and ... and client *r-1* holds *r-1*. The other clients hold no labs, and every Client knows this initial allocation.
13) A Client which holds no lab and does not request one should not be involved in any communication with other Clients, except for the limited period where it forwards requests if it just held a lab.

Submission and Assessment

You should submit the F# source code for each part, and each individual should separately submit an short (1-2 page) individual report for each part by the deadlines for each part. For the first part you must include an implementation of the `suffices` function. For the second part you must include an implementation of the Client class. **Make certain that you submit both the code and individual report.**

Your code will marked on how well it meets the specification, as well as its elegance, clarity (how easy it is to read), and how appropriately it uses the features of F#. **For Part 1, you should use function features rather than imperative features of F#. For Part 2, code that uses shared data structures, busy waiting or periodic polling is likely to receive a low mark, since this defeats much of the point of the exercise.**

For each part a fifth of your mark will be based on a brief report (roughly 1-2 pages, 3 at most) that describes:

- The structure of your program and how it works, particularly focusing on the concurrency and functional aspects.
- Any particularly interesting features of your program.
- How well your program works, including any situations or tests for which it doesn't work.
- The steps you followed in building your program, focusing on the problems that you needed to solve in this process.
- A reflection on your experience with this project. Was it too easy or too hard? Too long or too short? Interesting or boring? A good learning experience or mostly pointless? What could have been better?

Getting Help and Forming Groups

Help will be available from Rowan and Arran in the lab sessions. You are also encouraged to make use of https://secure.csse.uwa.edu.au/run/help3242 to discuss issues related to the project with other students and the teaching staff.

Please also use help3242 to find project partners – post if you need a partner, check to see who else is looking for a partner, and post to tell people when you've found a partner. (And pause to consider the concurrency issues involved in this project partner system.)