Homework 4: Pi Calculus

Due date: November 3 (Wednesday) at 11:59pm

Overview

This homework consists of two parts:

1. Implement a translation from the lambda calculus to pi calculus.
2. Write a few programs in the pi calculus.

This assignment returns to Python as the implementation language. Your assignment will be graded on the `myth.stanford.edu` cluster, so while you can develop anywhere be sure to test your code there. When running code on the myth machines, use the command `python3` so you test your code with the same version of Python we will use to grade your code. You are optionally allowed to work with a partner for this assignment. One member of each pair should submit the assignment on Canvas (instructions at the bottom).

Part 1: Translating Lambda Calculus to Pi Calculus

We will use a version of the pi calculus with the following concrete syntax, where P, Q are processes and x, y, c are variable names:

P, Q	::= `0`	do nothing
	| P `|` Q	run P and Q in parallel
	| `!`P	replicate P (i.e. $!P = P \mid !P$)
	| x `->` c`.` P	send x on channel c, then P
	| x `<-` c`.` P	receive v from channel c, then $P[x := v]$
	| `&`x`.` P	ν, local channel x in P
	| P `+` Q	run P or Q (and discard the other)
	| `[`x `=` y`].` P	run P if $x = y$
	| `[`x `!=` y`].` P	run P if $x \neq y$

See `tests/syntax.pi` for examples. In particular, send and receive have different syntax than was used due to the limitations of ASCII, and you need to terminate a sequence of send and receives with another process, which will often be `0`. Prefixes (`!`, send, receive, `&`, `[=]`, `[!=]`) extend to the nearest | or + unless overridden with parenthesis, i.e.:

$$\bar{c}x.\, P \mid Q \equiv (\bar{c}x.\, P) \mid Q$$

There is an additional restriction on the choice operator: all the alternatives must be *guarded*, i.e. start with a send, receive, or equality test. In particular, you can't write `(P | Q) + R`. This is to ensure that the choice operator `+` only has to look ahead "one step" to determine which branch to use, and thus doesn't need any backtracking. If we didn't have this restriction, programs would get into deadlocked states not due to a coding error, but because the choice operator picked the wrong branch. Note that this restriction eliminates the non-determinism in the choice operator, which makes the implementation much simpler.

While we don't support the true recursive definitions mentioned lecture, we do allow definitions for processes that are substituted directly, without α-renaming or arguments. These names must appear starting

with a capital (e.g. `ECHO`) to distinguish them from the channel names in lowercase. Remember that unlike the lambda calculus, there is no β-reduction rule: at runtime, processes aren't substituted—only channel names are replaced. This direct substitution means that you can include free variables in early definitions, and bind them later:

```
def F = x -> private. 0;
def G = &private. (F | y <- private. 0);
```

Translation

We will translate the lambda to pi calculus using Milner's lazy encoding. The reduction is specified by a translation function $\mathcal{T}(e, f)$:

$$\begin{aligned}
\mathcal{T}(x, f) &:= \overline{x} f.\, 0 \\
\mathcal{T}(\lambda x.\, M, f) &:= f(x).\, f(\underline{u}).\, \mathcal{T}(M, \underline{u}) \\
\mathcal{T}(M\, N, f) &:= \nu \underline{c}.\nu \underline{d}.\big(\mathcal{T}(M, \underline{c}) \mid \overline{\underline{c}} \underline{d}.\, \overline{\underline{c}} f.\, 0 \mid\, !\underline{d}(\underline{v}).\, \mathcal{T}(N, \underline{v})\big)
\end{aligned}$$

Note the duality of the result channel f: it is used both as a value to control the evaluation order of computation, and as a channel over which an abstraction and application communicate channels for the value of the formal parameter (d) and where the results should go (f).

Implementation

You will implement the translation by completing the function `translate` in `translate.py`. Notes:

- The input is a `lam.Expr`, as in HW1. You must return a `pi.Proc` as described above.
- See `src/pi.py` for the classes which implement the Pi calculus terms. Of these, you will only need `pi.Send`, `pi.Receive`, `pi.Nu`, `pi.Parallel`, and `pi.Replicate`. Do nothing (0) is represented by `pi.Parallel([])`. The reference solution is about 25 lines of code.
- Generate fresh variables in your translation for c, d, u, and v, to avoid problems with shadowing. You can use variables starting with exactly one underscore (`_c0`, `_c1`, etc.), which won't be used in the input or by the grading script.
- Test your code by running `python3 src/main_lam.py tests/*.lam`. Your output will consist of the translation itself, and the result of evaluating the code. For a lambda term that reduces to a free variable (like the tests), you should get something like:

  ```
  Final state(s):
  __result__ -> a. 0 | ...
  ```

  That is, the "result" channel is sent to the free variable `a` in this example. This seems backwards, but it is a result of the encoding. The `...` will be the left over processes that were used to do the computation.
- You are not allowed to modify the code in `src/`, or add subclasses to the classes defined there.

Part 2: Programming in the Pi Calculus

For this part, you will write some programs in the pi calculus. See `tests/syntax.pi` for a reference of the concrete syntax. You are also allowed definitions, which as in previous assignments will be inlined when used. After the definitions, there can be one or more pi calculus expressions. These expressions are each an independent instance of the pi calculus, so you can have multiple independent tests in one file. These tests use free variables to represent globals. Make sure your implementation does not clash with these symbols, by making sure your main process has the free variables indicated (helper processes aren't restricted). Implement the following by completing `problem*.pi`:

1. **Broken echo server**. Write a process `ECHO` which copies an arbitrary number of values from channel `in` to channel `out`. We have supplied two clients to exercise your code, but in this problem and the following ones, your solution will need to work for any number of clients. Note that your implementation is "broken" because each client may receive a different message than it sent. Your program should have two possible final states, up to equivalence (see below). Additionally, the only free variables should be `in` and `out`. The solution can be written in one line.

   In this and the following problems, your output is considered correct on the supplied testcases if (1) each final state matches the output pattern of one of the expected outputs, and (2) each output pattern in the expected outputs appears in at least one of your final states. Depending on how you solve the problems, your solution(s) may have more than one internal state with the same output. This is fine as long as the output parts match the expected output.

2. **Fixed echo server**. Write a process `FIXED_ECHO` which implements an echo server by modifying the interface. Your implementation will receive a private channel from `in`, then a value on that channel, and send the value back on the same channel. A client should be able to execute:

   ```
   &c. c -> in. v -> c. v2 <- c. [v = v2]. 0
   ```

   and the comparison should succeed. Your program should have only one possible final state, up to equivalence. Here equivalence means that there can be multiple output states as long as they only vary in the internal state of your implementation, i.e. the messages passed on the public channels should be the same. The free variables should be only `in`. The solution can be written in one line.

3. **Value server**. Write a process `VALUE` which implements a value server that allows reading and writing a global value. Your process should support two requests: get and set, over channels of the same names. You should use the interface from the fixed echo server: each interface channel should receive a channel on which the arguments, if any, are sent, followed by the response values, if any. In this problem, get takes no arguments and should return the current value stored, and set takes one argument, overwrites the value, and returns nothing. Clients can use the interface like this:

   ```
   &c. c -> set. v -> c. P
   &c. c -> get. v <- c. Q
   ```

   The free variables should be `get` and `set`. The reference solution is about 5 lines.

4. **Increment**. We will represent numbers in the pi calculus using two symbols `_zero` and `_succ` (these "symbols" are channels that we don't send or receive on). A number n then is represented by a channel over which some process will send n `_succ` followed by a `_zero`, and then stop. The `_succ`s, if any, must come **before** `_zero`, and each number can only be used once. For example, this is one possible process that makes `n` represent 2:

   ```
   (_succ -> n. _succ -> n. _zero -> n. 0)
   ```

   For this problem, write `INC` which increments numbers by one and returns them. Your process should use the private channel interface on `inc`, i.e. a client with some number `n` should be able to:

   ```
   &c. c -> inc. n -> c. n1 <- c. P
   ```

   and in `P`, should expect `n1` to be a channel encoding $n + 1$. You may assume that no other process is concurrently reading from the number you receive. The free variables should be at most `inc`, `_zero`, and `_succ`. The reference solution is 3 lines.

5. **Copy**. One downside of this representation of numbers is that they can only be used once. In this problem, you will fix this by writing a process `COPY` which takes a number and returns two fresh copies of that number. Your implementation must necessarily consume the input number. An example client usage is:

```
&c. c -> copy. n -> c. n1 <- c. n2 <- c. P
```

where `n1` and `n2` are copies of `n` with the same value. Note that you must implement copy so that `n1` and `n2` are **independent**. That is, each must correctly implement the number protocol, even if the other isn't being used. This means that a naive implementation that loops over the input and immediately tries to send to both output channels won't work, as one output will block the other. Instead, you will need to fully build some intermediate representation for each output up front, which can then be consumed lazily by the client(s).

The free variables should be at most `copy`, `_zero`, and `_succ`. The reference solution is 10 lines.

Test your implementations by running `python3 src/main_pi.py problem*.pi`. Note that testing your expressions will be a little more interesting because of the non-determinism inherent in the pi calculus. In particular, the broken echo server should have exactly two possible outputs: one where each client gets its own value, and one where they are crossed. This output should look like:

```
Final states:
--- State 0 ---
a -> c1. 0 | b -> c2. 0 | ...
--- State 1 ---
b -> c1. 0 | a -> c2. 0 | ...
```

Where `...` are terms related to your implementation. Because our language does not have any means of "printing" or returning a value other than sending a message, we use these unmatched sends to indicate the result of our program. Later problems will have a similar interpretation, but `...` will be much, much larger.

Additionally, you can use the `--trace` flag to print out all the possibilities at each step. For complex programs, this may be difficult to follow, especially when there are many parallel possibilities. This is one of the reasons that distributed systems are hard! To follow a single possible execution, use the `--single` flag as well. This will also highlight with color the places where the reduction is happening each step, which can be helpful when tracing a complex program.

Tips

- The key to solving the later problems is making effective use of replication (`!`) with state. The problem is that replication always produces a new process from the **same** template, so any state a prior copy had is lost when that process ends. A general template to solve this issue is by dividing the template process into two parts: the first is a private phase that receives the relevant state from the prior process or some initialization, and then a public phase where the process interacts with the world. Then, after the process has made one interaction, it prompts the replication for a new copy and sends the relevant state to the new copy. For example, here is a program that alternates sending two values to `output`:

  ```
  !(c <- new. x <- c. y <- c.
      x -> output. &c. c -> new. y -> c. x -> c. 0)
  | &c. c -> new. a -> c. b -> c. 0;
  ```

  Notice how it makes use of a channel `new` which is sent channels `c` both in the initialization and in the "recursive" call, and that the arguments are sent through fresh, temporary channels `c` so they don't get mixed up between concurrent clients.

- Make use of `--trace --single` to visualize the reductions that are occurring. You can also copy parts of this output, or even sub-expressions into a new file to run just part of your program: the intermediate states are also valid as input.

- It can be easy to lose track of variables and let them unintentionally escape the desired scope. The evaluator will print out the free variables for each definition to assist with this.

- The association of send, receive and `|` and `+` can be subtle. Don't be afraid to add extra parenthesis or indentation to keep things in order.

Submission

- Regardless of whether you are working in a pair, navigate to Canvas → People, then click on the "hw4-groups" tab and sign up for a hw4 group. If you are working with a partner, make sure you both sign up for the same group.
- Generate a tarball file `solution.tar.gz` by running `python3 src/submit.py` and upload the tarball file to Canvas. **Make sure the script gives you no errors or warnings.** Malformed submissions cause grading delays.
- Your solution tarball should include `translate.py`, `problem[1-5].pi`, and `README.txt`.
- Edit `README.txt` to include your student ID number (the 8-digit number) and the student ID number of your partner (if you are working in a pair).