Process Algebraic Model of Parsing

Atreyee Ghosal

20161167

April 29, 2020

Introduction

The broad aim of this project was to implement a model of natural language parsing based on concurrency and processes as a representation of a word.
The portion of the project done so far has been focused on implementation. The following have been implemented:

- An implementation and reduction system for a basic process algebra.
- Exploration on how process algebra reduction rules can be applied to natural language parsing.
- Demonstration of the above on natural-language-like examples (a test suite)
- A parser (the CFG kind) for process algebra, to enable user input.

Core Components

Process

Each process represents a transition system

- Parallelism: several processes can run in parallel, "bonding" i.e: exchanging channels with each other
- There is no ordering between processes, and processes do not communicate with each other in any particular order. The list-ordering used in the implementation is only for convenience.

A naive implementation of processes:

```
data Channel = Output Name | Input Name | Empty

data Process =
  Empty Name
  | Prefix Channel Process
  | Parallel [Process]
```

Reduction System

So, the reduction system is, more appropriately, a transition system, as nothing gets reduced- rather, processes undergo a state transition on communication.
The initial input to our system is a set of processes running in parallel. The final output of our system is also a set of processes running in parallel, but where each process is "stable" i.e: it does is in a defined end state.

Transition Rules

The only transition rule for this simple system involves processes offering/accepting channels.
Two processes running in parallel can communicate like below:

```
P1 = subject? P1'
P2 = subject! P2'

c? P1 | c! P2 -> P1' | P2'
```

Note that the parallel operator '|' is commutative and associative.
Let's call this rule the **reaction function**.

Reaction Function: Naive

We define a "matching" channel pair as two channels that cam communicate, i.e: a pair of channels (!c, ?c)
A function *inverse (c1, c2)* returns a boolean value stating whether c1 and c2 are matching.
Therefore, a naive implementation of the reaction function would be:

```
reaction :: Process -> Process ->
            [Process, Process]
reaction (Prefix ch pr) (Prefix ch' pr') =
        if (inverse ch ch') then [pr, pr']
        else [(Prefix ch pr), (Prefix ch' pr')]
reaction p q = [p, q]
```

Parsing As Reduction

If we want to model natural language parsing as a process algebra, we need to define the end result of the parse- i.e: a final state of the system that, once reached, qualifies as a parse.
If we restrict channels to only mandatory arguments and nothing more, then we can define a stable process as an *Empty* process, as in, a process without any channels.
But what do processes and channels represent, anyway? How are we tying natural language to a model made for concurrent systems?

Kaaraka

Kaaraka is a syntactico-semantic theory proposed by Panini for Sanskrit grammar, and refined in modern times to be suitable for free-word-order languages like Hindi.
There exists a complex semantic element to Kaaraka theory. For this presentation, however, we focus on the syntactic element.

- The core of an utterance is the action, *kriya*, which is represented by the verb.
- The notion of a "kaaraka" is somewhat similar to the notion of a case. There are six "kaarakas" in the basic theory, which have been expanded in the modern Computational Paninian Grammar.
- Action Verb (Unfulfilled Action) + Arguments -> Fulfilment of Action i.e: change in state of the world
- So, process algebra- being a state-based model- is semantically truer to Kaaraka theory, as well as being syntactically truer w.r.t free word order.

Parsing As Reduction

There are six basic kaarakas in kaaraka theory: karta ('agent'), karman ('deed'/'object'), adhikarana ('location'), karana ("instrument"), sampradana ('bestowal'), apadana (lit. 'take off'). For transitive verbs, only two of those arguments are necessary, and the remaining four are optional.
If we extend our model of natural language to account for the possibility of optional arguments, then we need to change our definition of a process.

```
data Process =
  Lexeme Name
  | Prefix [Channel] [Channel] Process
  | Parallel [Process]
```

Finality of a process is now not determined by type, but rather by a function- checking that the first list under the **Prefix** constructor is empty.

Reaction Function for Multiple Channels

To simulate a channel-based form of communication where each process has multiple channels, we iterate through all possible channel pairings of processes P and Q, and take the first set of "matching" channels.

```
chPairs =
    [ ((c1, c2), inverse c1 c2) |
     c1 <- (c ++ cOpt),
     c2 <- (c_ ++ cOpt_) ]
```

Reaction Function For Multiple Channels

When two channels communicate, they are removed from the list of available channels for that process. Thus, the function returns a new **Prefix** process with the same lexeme and one channel removed.

```
case pair of
  [] -> [(Prefix c cOpt p), (Prefix c_ cOpt_ p_)]
  s -> let
    channels = head ls
    cnew = del (fst channels) c
    cOptnew = del (fst channels) cOpt
    cnew_ = del (snd channels) c_
    cOptnew_ = del (snd channels) cOpt_
    in
    [(Prefix cnew cOptnew p),
    (Prefix cnew_ cOptnew_ p_)]
```

Reaction Function: Incoherent Sentences

In our model, both reactive and stable processes are formed by the constructor **Prefix**.
Thus if the system attempts to reduce any type of process, it means that there's a single word without any connections in the sentence, and thus the sentence is incoherent. We raise an error in that case:

```
runParallel p1 p2 =
        error $ "Process " ++ (show p1)
            ++ " " ++ (show p2)
            ++ " cannot communicate!\n"
            ++ "Word makes no sense"
```

Reduction: Interactive

The first iteration of the overall reduction function is interactive- we are given N processes running in parallel, and we can choose two processes to communicate. (This simulates FIFO channels)

```
runTwoInParallel :: [Process] -> Int -> Int
                    -> [Process]
runTwoInParallel plist i j =
  remainder ++ result
        where
          remainder =
                del (plist !! j)
                          (del (plist !! i) plist)
                result =
                          runParallel (plist !! i)
                                      (plist !! j)
```

Reduction: Automatic

In the second iteration of the reduction function, the function is given N processes. It simulates parallel processes by communicating between each possible pair of processes, repeatedly, until the system is either stable (a full parse) or unchanged between two successive runs (no further reduction and the system is not stable- i.e: sentence is unparseable).