Proceedings of DiSS'03, Disfluency in Spontaneous Speech Workshop, 5–8 September 2003, Göteborg University, Sweden. Robert Eklund (ed.), Gothenburg Papers in Theoretical Linguistics 90, ISSN 0349–1021, pp. 59–62.

In dialogue with a desktop calculator: A concurrent stream processing approach to building simple conversational agents

Torbjörn Lager

Department of Linguistics, Göteborg University, Sweden

Abstract

Human spontaneous face-to-face conversations are characterized by phenomena such as turn-taking, feedback, sounds of hesitation and repairs. A simple and highly modular stream-based approach to natural language processing is proposed that attempts to deal with such things. A basic version of the model has been implemented in the Oz programming language.

1. Introduction

"And you do Addition?" the White Queen asked.
"What's one and one and one and one and one and one and one and one and one and one?"
"I don't know," said Alice. "I lost count."
"She can't do Addition," the Red Queen interrupted.

Lewis Carroll: "Through the Looking Glass"

Suppose we would like to hook up a desktop or pocket calculator to speech recognition and speech synthesis hardware and software, so as to enable people to solve arithmetic problems in an interactive fashion, in spoken dialogue with the resulting system. Imagine for example being able to ask something like [tu: plʌs θri:] and after a short pause receive the answer [faɪv]. Imagine further being able to follow up with [taɪms tu:] after which the system will respond [ten].

What is the best way to accomplish this? It would not be very hard at all of course, unless we insisted on building a system that mimics human dialogue processing. Then it becomes considerably harder. Human spontaneous face-to-face conversations are characterized by phenomena such as turn-taking, feedback, sounds of hesitation and repairs [2]. Also, ambiguity and the resolution of ambiguity work in a different way in spoken language. We know very little about how to handle these things.

The purpose of the present paper is to propose a computational framework – Concurrent Stream Processing – in which modeling of this kind of interaction becomes simple and natural. As we shall see, the main attraction of this approach is that two important things come almost for free: incrementality and modularity. Incrementality makes seamless interaction possible. Modularity helps us fight the complexity inherent in building systems like these.

Although the main motivation behind the choice of type of application is simplicity, its potential use in ubiquitous computing – perhaps as a calculator for the blind – should also be obvious.

2. Dialogue game design decisions

In this section, a relevant dialogue game will be designed, by considering one by one the features required.

2.1. Basic moves

From here on, dialogue fragments will be presented in the form of 'musical score' transcripts. For example, corresponding to the four turns in the conversation above we have:

```
U: 2+3   *2
S:     5   10
```

In a transcript like this, time flows from left to right, and characters that are aligned horizontally represent simultaneously occurring sounds. Periods of silence are transcribed as space characters. Thus, we see that there are periods of silence during which neither the user nor the system speak. Typically, this is where a change of turns is taking place, and indeed, it is the very occurrence of a pause of that particular length that signals to the system that it is allowed to grab the turn in order to present a (possibly only intermediate) result.

The example also shows that evaluation in this game is *incremental* in the sense that intermediate results are calculated, and may also be presented, along the way towards a final result.

Next, consider the following exchange:

```
U: 2+3 *2          2 +3*2
S:        10              8
```

First, note that the user's first turn is almost identical to the combination of his first two turns in the previous example. The only difference is in the length of the pause; in this example it is not long enough to invite a response.

Secondly, note that although the two questions in the last example consist of basically the same sequences of phonemes, the pauses are inserted differently. The position of the pauses are very significant, since they determine whether a question is to be interpreted as (2+3)*2 or as 2+(3*2). This is the way *ambiguity* will be treated in the game. The pairing up of parentheses is not used much in spoken natural language, so we disallow them altogether. We use one disambiguation device only – the pause – which means that only two levels of syntactic embedding can be handled, and we assume that this is sufficient.[1]

[1] Compare the use of comma (,) in written text: *John and Mary, or Paul* vs. *John, and Mary or Paul.*

Finally, note the long stretch of silence between the second and third turn. It illustrates the fact that, depending on the state of the dialogue, a pause – long or short – may sometimes not mean anything at all.

2.2. Sounds of hesitation

The perhaps main purpose of *sounds of hesitation* is to prevent the other party from grabbing the turn. This is functionality that we want to support in our dialogue game. For example, if our user is not interested in receiving intermediate results, yet knows he is not able to speak with a pace fast enough to avoid that, he might instead say:

```
U: 23+30errrr+312
S:                  365
```

In other words, the user produces a sound of hesitation – transcribed here as "errr" – in order to prevent the system from grabbing the turn and presenting the result of evaluating 23+30.

In general then, when a dialogue is in a state where a speaker induced pause *would* mean something, and what it *would* mean is not intended by that speaker, it is important for the speaker not to be silent and thus, as it were, 'unintentionally' produce a pause. We will assume that this is an important 'rationale' behind sounds of hesitation such as "err". It is most likely not the whole story, but it is a mechanism that seems to do the job in our dialogue game.

2.3. Self-repair

The proposed dialogue game also supports a limited form of *self-repair*, i.e. the ability of the system to understand the speaker's attempts to repair his own utterances, and to react properly. Consider the following example:

```
U: 2+2no3       2-3no+3
S:         5             5
```

Indeed, in the proposed game, utterances of the expressions `2-no2+3`, `2-no+3`, `2-3no2+3`, `2-3no+3` and `2-3no+` all mean the same thing, and evaluate to the same answer, namely to the number `5`.

Note that there is room for subtle forms of interaction between the processing of sounds of hesitations and the processing of self-repairs. Consider:

```
U: 1+2errrrrrno3
S:               4
```

What happens here is that when the user has uttered [tu:], he immediately realizes that this is not what he intended. However, as he is not yet certain about what to say instead, he produces – while thinking – a sound of hesitation. He thus prevents the system from taking the turn and answer the question not intended (something that would make subsequent straightforward self-repair impossible). Had the user been silent instead of generating this sound, the dialogue might have ended up in the following confused and clearly undesirable state:

```
U: 1+2   no3
S:     3
```

This concludes the description of our dialogue game design. We have not said anything about openings and closings of dialogues. In a realistic application, such things would also be important, but will be ignored here.

3. The Alice demonstrator

A demonstrator and research tool has been implemented – nicknamed *Alice* – which allows a user to enter into conversations of the kind described above. The processing architecture of Alice is based on the notion of *concurrent stream processing*. A *stream* is an ordered, open-ended and potentially unbounded sequence of tokens. Stream processors are *transducers* that transform input streams into output streams. The following code, written in the Oz programming language [4], forms the top-level of the Alice system.

```
thread {Listen S0} end
thread {FilterSoH S0 S1} end
thread {Repair S1 S2} end
thread {Chunk1 S2 S3} end
thread {Chunk2 S3 S4} end
thread {TakeTurn S4 S} end
thread {Speak S} end
```

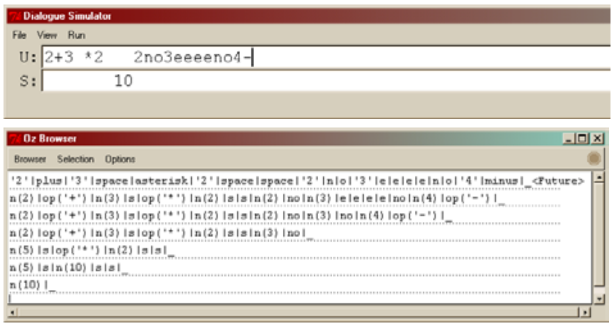
Here, `FilterSoH`, `Repair`, `Chunk1`, `Chunk2`, and `TakeTurn` are transducers, composed so that `FilterSoH` reads the stream `S0` produced by `Listen` and creates another stream `S1` which is read by `Repair`, and so on. The resulting stream `S` is eventually spoken by `Speak`. As a clear proof of modularity we note that this is the *only* point in the system where the modules communicate.

The transducers are run in parallel – each in its own thread. It means that the system is able to listen, speak, and 'think' – e.g. perform all the important language processing steps in between listening and speaking – at the same time (although it doesn't mean that it always does).

The threads are so called *dataflow threads*, i.e. they suspend on the availability of data [5]. Given two transducers running in separate threads, the second one will suspend if and only if it needs some part of the output stream of the first which is not yet available. An interesting and extremely useful consequence of this setup is full incrementality: if input is given incrementally, then the output will be computed incrementally as well.

A comparison with ordinary (finite-state) transducers may be useful. Ordinary transducers transform predetermined input strings into output strings, shutting out the world during the process of computation. Our concurrent stream transducers are transducers of incrementally generated streams, which means that interaction with the external world during computation is possible. This, of course, is crucial when processing dialogue.

The Alice GUI contains two 'musical score notation' fields. The user inputs his contributions in the field marked "U:" and – while the user types – the system responds in the field marked "S:"



```
U: 2+3 *2    2no3eeeeno4-
S:         10
```

```
'2'|plus|'3'|space|asterisk|'2'|space|space|'2'|n|o|'3'|e|e|e|e|n|o|'4'|minus|_<Future>
n(2)|op('+')|n(3)|s|op('*')|n(2)|s|s|n(2)|no|n(3)|e|e|e|e|no|n(4)|op('-')|_
n(2)|op('+')|n(3)|s|op('*')|n(2)|s|s|n(2)|no|n(3)|no|n(4)|op('-')|_
n(2)|op('+')|n(3)|s|op('*')|n(2)|s|s|n(3)|no|_
n(5)|s|op('*')|n(2)|s|s|_
n(5)|s|n(10)|s|s|_
n(10)|_
```

Figure 1: The Alice demonstrator GUI.

It is up to the user to indicate the passing of time by padding with space characters in the "U:" field as he sees fit.

By means of the Oz Browser the user is able to inspect the streams as they grow – a convenient feature when debugging.

Everything the user writes in the "U:" field is tokenized in an incremental fashion and each token is put in a stream. For example, if the user writes

```
3errrr+14
```

the stream will look as follows:

```
n(3)|e|r|r|r|r|op(+)|n(14)|s|_
```

The system is not yet able to take spoken input or produce spoken output, but this is planned for the future.

3.1. Processing sounds of hesitation

As noted in Section 2.2, the purpose of sounds of hesitation is to prevent the other party from grabbing the turn. We suggest that by just being uttered, they have already served their purpose, because it means that – by necessity – pragmatically significant pauses not intended by the speaker have not occurred, and this is all that is needed. Consequently, the proper way to deal with sounds of hesitation in the system is to treat them as noise and to remove them – to filter them out so as to stop them from reaching deeper into the cascade of transducers.

What this means in practice is that a stream such as

```
n(3)|e|r|r|r|r|op(+)|n(4)|s|_
```

is simply transduced into

```
n(3)|op(+)|n(4)|s|_
```

3.2. A rewrite model of self-repair

In Alice, the transduction relevant to self-repair is implemented as a sequence of rewrite rules, a selection of which is shown here:

```
n(_)|no|n(N) => n(N)
n(_)|op(_)|no|n(N)|op(Op) => n(N)|op(Op)
op(_)|n(N)|no|op(Op) => op(Op)|n(N)
```

A stream such as

```
n(3)|op(-)|no|n(3)|op(+)|n(4)|s|_
```

gets transduced into

```
n(3)|op(+)|n(4)|s|_
```

3.3. Parsing as two levels of chunking

In this section we consider the parsing problem as it manifests itself in our dialogue game. We want our parsing strategy to be incremental, and we want to resolve potential ambiguity in an intuitive way. Let us begin by reviewing our options.

With a mindset tuned to the parsing of written arithmetic expressions – and as victims of the "written language bias" in linguistics [3] – we might consider using a grammar such as

```
E -> E+E|E-E|E*E|E/E|N
```

to parse our arithmetic expressions.

However, it is easy to see that this is a bad idea. This grammar is highly ambiguous, and will produce numerous parse trees for moderately complex expressions.[1] There are two standard ways to avoid this ambiguity problem. One is to introduce parentheses into the language and have strict rules for writing arithmetic expressions ensuring that there are always a sufficient number of parentheses to determine the order of operations. The other is to have precedence rules which tell us how to evaluate an expression (e.g. multiplication and division are performed before addition and subtraction). These strategies can be – and often are – combined, e.g. in the form of a grammar such as

```
E -> E+E|E-E|F
F -> F*F|F/F|(E)|N
```

We will use neither strategy. The user should not have to 'speak' parentheses, and the use of precedence rules alone is not flexible enough. The solution here is basically to throw away everything we have learned about parsing of written arithmetic expressions. We will suggest a very simple chunking approach instead, part of which can be paraphrased as follows: "Once you detect a short pause in the input stream of sounds, go ahead and evaluate the chunk that you have heard so far. Remember the result, because the user may soon want to be presented with it, and/or it may serve as an operand in a larger expression of which you have so far only heard a part."

It turns out that two levels of chunking, implemented by composing two simple stream transducers, are sufficient.

Figure 2 depicts the first transducer in the cascade.

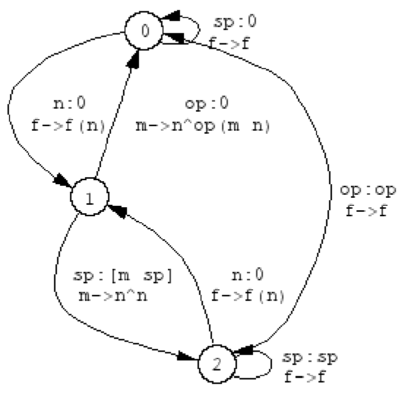


Figure 2: Level one chunker.

This transducer has three states. Each state has a dynamically changing value associated with it, either a number or a unary function from numbers into numbers.[2] The transducer also has transitions, each of which is labeled with two pairs of the form

`In:Out`
V_1`->`V_2

where `In` and `Out` are tokens and V_1 and V_2 denote the value of the leaving state and the arriving state, respectively. (Initially, the value of the start state (0) is the identity function.) The pair `In:Out` will map the token `In` in the input stream to `Out` in the output stream. In case `Out` is `0`, `In` maps to *nothing*.

Processing works as follows. The transducer takes a stream of tokens as input, reads the stream one token at a time from

[1] In fact, they are known to have a combinatorial (Catalan) number of syntactic parses. E.g. 2+3*2+2*6+4 has 42 parses.

[2] `^` is the lambda abstraction operator (e.g. `n^n` is the identity function), and `f(n)` applies a function `f` to an argument `n`.

left to right, and writes corresponding tokens to an output stream.

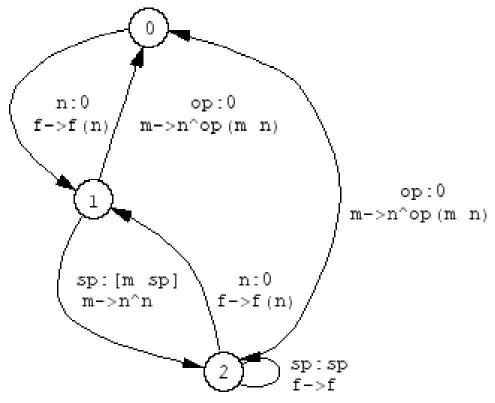


Figure 3: Level two chunker.

The result of running this transduction on the input stream

```
n(2)|op(+)|n(3)|s|op(*)|n(2)|s|s|_
```

is the output stream

```
n(5)|s|op(*)|n(2)|s|s|_
```

This implements the first level of chunking. The result of running the transducer depicted in Figure 3 on this output is the following stream:

```
n(5)|s|n(10)|s|s|_
```

and this completes the parsing process.

3.4. Turn-taking

The turn-taking mechanism of our system is based on the transducer in Figure 4.

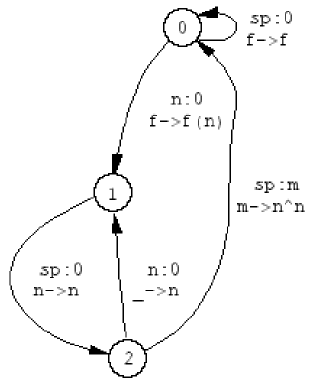


Figure 4: Turn-taker.

Given the stream

```
n(5)|s|n(10)|s|s|_
```

the turn-taking transducer produces

```
n(10)|_
```

and this is what is spoken to the user. Note that the intermediate result (5) does not pass through the filter and thus is not spoken. The pause is simply not long enough to allow the system to grab the turn.

4. Discussion

Although the full validity of our approach can only be determined once speech has been added to the system, we like to think of the work reported in this paper as a first attempt to build a truly asynchronous dialog system based on concurrent stream processing techniques.

Our dialogue domain of choice – numbers and arithmetic operations on numbers – is undoubtedly very simple, and it would of course be interesting to try to build dialogue systems over more complex domains using our approach. Even so, despite the fact that a lot of pragmatics phenomena (presuppositions, implicature, etc.) do not show in our dialogues, there is still room for a fair amount of variation, which we have yet to explore fully. For example, there are *feedback moves* [1] that would be natural to have in our dialogue game. Let us close this paper by looking at a few of those, commented very briefly.

User makes incomplete utterance. System prompts for completion:

```
U: 2+2+                  3
S:             + what?   7
```

User does not hear, and therefore (twice) prompts for (rephrased) repetition:

```
U: 2+3   what?       *2    what?
S:     5         2+3=5    10        5*2=10
```

User's query is ambiguous. System enforces disambiguation:

```
U: 2+3*2            yes
S:         2 +3*2?      10
```

Presumably, these moves would lend themselves to straightforward implementations in our framework.

5. Acknowledgements

Thanks to people in the Gothenburg Dialogue Applications Group for discussing these things with me, and in particular to Robin Cooper for reading and commenting on an earlier draft of the paper, and to Aarne Ranta for engaging me in an interesting email exchange concerning the mathematical properties of spoken mathematical language.

6. References

[1] Allwood, Jens, Joakim Nivre & Elisabeth Ahlsén. 1992. On the Semantics and Pragmatics of Linguistic Feedback. *Journal of Semantics*, vol. 9, pp. 1–26.

[2] Levinson, Stephen. 1983. *Pragmatics*. Cambridge University Press.

[3] Linell, Per. 1982. *The Written Language Bias in Linguistics*. Department of Communication Studies, University of Linköping, Sweden.

[4] The Oz/Mozart Consortium. 2003. See: `http://www.mozart-oz.org`

[5] van Roy, Peter & Seif Haridi. Draft. *Concepts, Techniques, and Models of Computer Programming*. See: http://www.info.ucl.ac.be/people/PVR/book.pdf