Delayed Semantics

by Sven Nilsen

Motivation

Our motivation is having multiple algorithms listening to a stream of data. A tokenizer splits up the stream into tokens. There are 'k' number of different token classes. What data type each token is does not matter.

Assuming the application of listening to the stream is a hard problem. It is difficult to create a single algorithm that interprets the stream perfectly. Each algorithm knows it does not possess all the knowledge required. In the output of the algorithm it leaves the part of the stream it can not understand untouched. With other words, the purpose of each algorithm is to replace parts of the stream where it can interpret the data. The technical term is a 'monoid' function, because the tokens in the stream are in the same set for input and output.

Each instance of an algorithm can be a heavy operation. In such cases it requires a lot of resources to interpret recursively encoded data. This tells us something about the kind of problem. The key is coined in the term “partially understood information”. Delayed semantics is about dealing with the relationships between multiple algorithms to make sure that there is no collision. There is a way of solving this by delaying one algorithm when another algorithm gets active on a subset of the input.

The problem is in contrast with streams where semantics is controlled by special flags preceding the data. These are much easier problems, because one can just collect the input when triggered by a flag. In delayed semantics, we need to solve how much of the stream is stored in memory. The focus changes from doing stuff with the data to the internal state and relations between algorithms. The system that manages and enforce safety has a property of self reflection. It might be able to implement new algorithms in real time without compromising safety.

All algorithms can operate on the same queue storing sequence of tokens. The amount of memory involved is restricted to the size of the queue. This makes it possible to consume very little memory and use the memory for more algorithms instead.

Proving Safety

First we prove using number theory that it is possible to design a system without collision in semantics. Every token lives in an address space of size 'k'. For any given sequence of tokens, we can uniquely identify the sequence with a number:

$$k^n$$

The input required for an algorithm is also a such number. Two algorithms having the same number means a collision. Therefore, if every algorithm is associated with a unique input, then there can not be any collision. Given a set of algorithms, we can generate a subset that matches the sequence of tokens so far. As the sequence of tokens stored in memory grows, the generated subset should shrink.

An algorithm can contain some internal logic that prevents it from colliding with another algorithm even they have the same input. For example, if A converts text to 'int' and B converts text to 'f64'. They both become active on text. It is not correct that each algorithm must be uniquely identified, only that those who do are exclusive logically. One way to solve non-exclusive algorithms is creating guarded statements where the next algorithms only processes input if the previous ignore it.

The complexity arises because most algorithms only work on a limited set of input types. If a single token is off by one, then a sequence of such tokens gets off exponentially with the size of the sequence. It is not reasonable to have one algorithm for every combination of types. Some algorithms can do conversion or require the token type to have a method for the conversion.

The safety we can guarantee is restricted to no collision between algorithms. The lack of one directional convergence toward a consistent interpretation of the input stream is not guaranteed. It might be possible to overcome this problem by giving additional information about the algorithms.

Encoding Token Types

If we only have two types of tokens, then the type of sequence can be encoded as a binary number. The input type of an algorithm must match the tokens.

A possible representation is giving each token type a letter and use regular expressions to define the input type for an algorithm. The text generated from a sequence of tokens is living in a type space, so the regular expression is matched in the type space and not necessarily in data. This technique can make it easier to use delayed semantics in languages supporting regular expressions.

An algorithm might require arbitrary length of inputs. This leads to an interesting design space of encoding the stream. For example, if we have an arbitrary sequence of numbers, we might flag it somehow to not activate algorithms requiring a fixed number of inputs.

Subset Algorithms

The question is how much complexity we should do to keep the number of active algorithms low. If an algorithm processes some data, should we then reinterpret the output with other algorithms? Here is an example:

[0, 2+3]

One algorithm A can interpret arrays while another B adds numbers. We start activating A because we read the beginning of the array, but in the middle we activate B. B starts later than A, so it is a subset algorithm and A should read the output of B. We do not need to understand how A and B works. The only information required is knowing at which position A and B got active.

What if one algorithm does not want sub algorithms? What if it has its own subset of algorithms? This leads to the idea of sets of algorithms as data.

Sets of Algorithms vs Sets of Tokens

A set of algorithm is similar to a set of tokens. A set of tokens describe what kind of information we can read from the stream. No algorithm can create output that is a new kind of token. A set of algorithms describe how many systems we compose together in order to understand the input.

One of the first questions we ask:

If an algorithm dictate a set of algorithms as subset algorithms, can this set contain the algorithm itself?

Another way to attack this question is by :

Can a set of tokens contain a token representing a stream?

It makes sense, because a stream can be encoded as a file and files might be a valid type. When we describe a set of algorithms, we might also refer to a file containing a list of the algorithms to use.

Is it then strictly necessary to separate algorithms from tokens? An algorithm needs of course a physical implementation, but there is no theoretic obstacle to address that implementation.

Manager Flow