Flow Language Proposal

PLT 4115 Fall 2015

Mitchell Gouzenko (mag2272), Zachary Gleicher (zjg2012)
Adam Chelminski (apc2142), Hyonjee Joo (hj2339)

Intro

Flow is a language that aims to process streams of input using the Kahn Process Network (KPN) model through a variety of user-defined transformations. As the name of the language suggests, the goal of Flow is to enable cascading of data over seamless sequences of operations and functions (aka transformations). Data will pass through transformers, each of which will have a well-defined input protocol. The syntax of Flow will make it intuitive to connect transformations with each other. We plan on compiling Flow into multithreaded C code.

Motivations

Data processing has become one of the most important tasks for computers, and we want to design a language that makes it easy to create pipelines for reading, transforming, and writing data. We think that traditional sequential programming languages do not provide the most intuitive method for pipelining data, but rather, can be better organized with an alternative model of computation that resembles a directed acyclic graph. The nodes of the graph, which we have decided to call transformers represent workers that perform a specific task and route data to other transformers. Flow would be a useful language for implementing the following three types of programs:

- A program that processes a stream of log messages.
- A program that processes a stream of MIDI messages to produce sounds.
- A program that analyzes streams of financial data.

Language Overview

The most important concept in Flow is the transformer. A transformer is similar to a function in that it contains code to execute and accepts arguments in the form of an input protocol. Furthermore, transformers can have persistent state variables. However, rather than returning a value to the routine that invoked it, a transformer can only send data forward to zero or more other transformers. A Flow program is built by creating a network of these transformers, each of which performs a task on a single mutable element of data. Each transformer has a queue of data elements that were sent to it but not yet processed.

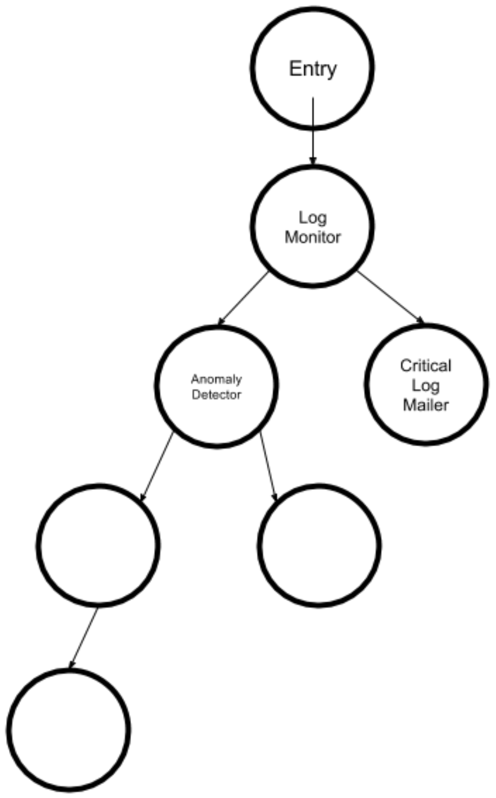
Like other programming languages, Flow also supports traditional functions that return values and accept arguments. Regular functions can be invoked in the code of a transformer as well as in other regular functions. Functions cannot send data to transformers.

A special entry transformer is used as an entry point for a Flow program. This transformer will not have an input protocol and will instead read input from a raw data stream.

The last important piece of Flow is the protocol, which defines the type of data that a transformer can accept. Each transformer specifies a protocol for its input. A protocol is defined similarly to a C-style struct, except that it can also contain a method that defines how to convert a stream of data into an element of the protocol type. This makes it easier for programmers to read from raw data streams in the entry transformer.

Syntax Example

Below, we present a sample program that parses application error logs from stdin. These logs vary in severity, origin (within the application), and are not easily readable by the human eye. The flow program we propose aims to route these logs to appropriate destinations in a logical way. The graphic below illustrates a KPN of the transformers that compose this program, along with their descriptions.



Entry: The entry point transformer that formats raw input from stdin into log messages.
Log Monitor: Transformer which routes logs to the appropriate transformers. It makes decisions based on metadata that it collects about the stream. For instance, when the average interval between error logs becomes too small, the log monitor starts forwarding data to the Anomaly Detector.
Anomaly Detector: Looks for the anomaly that has caused errors to start occurring more frequently.
Critical Log Mailer: Sends an email to the lead engineer on the team that is in charge of the feature that has caused a critical error.

```
int BATCH_SIZE;

protocol time_stamp{
      int second;
      int minute;
      int hour;
      int day;
```

```
    int month;
    int year;
}

protocol log_message{
    from(int source){ // Source is a file descriptor
        //Initialize the protocol structure from file stream
        this.warning_level = . . . ;
    }
    int warning_level;
    string message;
    string module_name;
    time_stamp tstamp;
}

transformer entry {
    log_message message;

    transform() {
        // invokes the ‘from’ method of message
        message <- stdin;

        // send the message to the log monitor transformer.
        message -> log_monitor;

        // The program will repeat this indefinitely until EOF is reached.
        // At this point, the program will terminate cleanly by stopping the
        // entry transformer, and emptying the queues of the other
        // transformers. Alternatively, the keyword ‘exit’ in any transformer
        // will start the clean termination process.
    }

}

transformer log_monitor{
    // Persistent variables to preserve state
    int interval_average;
    int position;
    int [10] interval_list;

    transform(log_message message){
        if (log_message.warning_level > 10){
            // If the warning level is particularly high, send the log to
            // a transformer that can determine which team to email it to.
            message -> critical_log_mailer;
        }

        update_moving_average();

        if(interval_average < 2){
            // Something is broken. Start sending the logs to a transformer
            // that knows better how to detect anomalies in the log stream
            message -> anomaly_detector;
        }
    }

    void update_moving_average(){
        this.interval_average = ...;
    }
}

transformer critical_log_mailer {
    transform(log_message message){
        // Code that emails the appropriate team
        ...
    }
}

transformer anomaly_detector {
    transform(log_message message){
        // Looks for the source of the error
        ...
    }
}
```

Goals

- **0.8 (MVP)**
    - transformers with
        - input protocol
        - ability to send to a transformer
    - each transformer executes on its own thread
    - a special entry transformer
    - normal C-style functions
    - control statements
        - if, else, while, etc.
    - local variables
    - protocols
        - with a ‘from’ stream method
    - a way to check for cycles in the program at compilation time
    - an ‘exit’ keyword
- **1.0 (Final Project)**
    - persistent state within a transformer
    - a standard library of functions based on the C standard library
    - a built-in string type
    - the ability to invoke C code within Flow code with a special block so as to make it easy to use existing C libraries
- **1.2 (Stretch goals)**
    - multiple input sources to a transformer
        - special type of transformer
    - a clean way to define optional and repeated types within a protocol
    - the notion of transformer factories to modularize transformer functionality