PLS
COMS W4115

Bennett Rummel, bwr2116

June 11, 2014

1 Introduction

PLS (PipeLine Script) is intended to be a small, imperative, dynamically typed, statically scoped, programming language designed to process streams of data. It will allow the user to construct pipelines for filtering or otherwise transforming sequences of data. It is intended to be embedded in a larger program (development time permitting) but we will provide a command line utility for simple use cases and demos. Programs will be compiled to bytecode instead of machine language.

2 Motivation

Many common programming tasks involve applying a sequence of operations or analytics on a stream of data. For example, imagine a server process that writes log data about every request it receives from users into a text file. If a programmer wanted to analyze this log data and calculate some statistics about a specific user, he or she has a few common options. However, a common solution in one-off cases is to construct a sequence of commands in a shell to filter (e.g. "grep"), transform (e.g. "awk", "sed", etc.), or otherwise prepare a set of data before passing it to some sort of script in another language to perform the actual business logic. Such a scheme requires knowledge of many different tools but is often simpler than implementing the entire sequence in a single language (e.g. "python" or "perl"). One of the goals of PLS is to provide a similar programming model in a unified package.

3 Language Overview

3.1 Kernels and Pipelines

The main component of a PLS program is the `pipeline` type. A `pipeline` is made of a sequence of one or more processing `kernel`s. A `kernel` consumes values of arbitrary types from a source stream (output from the preceding `kernel` in the pipeline) and optionally emits a value to the sink stream. `kernel`s can access (but not create) global data, create local-scope variables that will be destroyed after the scope in which they are defined is left, and static local variables that are preserved across processing instances of the `kernel`.

A program in PLS is simply a special user-defined pipeline called `main`. Values will be pushed through the pipeline one-by-one until a special "end" value is processed, after which the program is terminated.

3.2 Data Types and Expressions

PLS should support common scalar types (`integer`, `float`, `string`, `bool`) as well as heteregenous containers (`tuple`s) and homogeneous arrays. A `none` type will also be provided to represent the abscence of a value.

Both `array`s and `tuple`s will be indexed using square brackets and zero-indexed indices, e.g. `my_array[0]`. Additionally, both `array` and `tuple` types are mutable but not extendable (i.e. elements can be changed but not added or removed).

Expressions will use infix notation. Common C arithmetic and boolean operators (+, -, *, /, %, !, ==, ! =, >, <, >=, <=, ++, −−) will be supported with similar semantics. Assignment will use the = operator. Data types will also be implicitly convertible following similar rules to C, e.g. `float`s can be truncated `integer`s, `integer`s can be converted to boolean "true" if non-zero, etc. Strings will have to be explicity converted to other types.

3.3 Control Structures

Control structures will also follow C (the astute reader will have detected a pattern):

```
if (boolean_condition) {
    // do something if boolean_condition == true
} else if (...) {
    // ...
} else {
    // finally, do something else
}

while (condition) {
    // ...
}

for (i = 0; i < 10; ++i) {
    // ...
}
```

Additionally, the `foreach` construct will be offered for iterating over arrays:

```
foreach (value : some_array) {
    // do something with value
}
```

3.4 Functions

User defined functions are also supported and can be called from with in processing `kernel`s. Functions in PLS take zero or more parameters and return a single value. Like `kernel`s, functions can access global data, local data, and static local variables. For simplicity, parameters are passed by value.

4 Syntax and Examples

Statements in PLS are terminated by a semicolon. Whitespace is not significant beyond separating identifiers (C-style). Comments begin with a two consecutive forward slashes and continue until the next newline (C++ style). Blocks consist of one or more statements surrounded by curly brackets. Variables also follow similar scoping and rules as in C. Further details are probably better illustrated in the examples below.

A simple program that simply copies its input to its output could be written as follows:

```
kernel identity(x) {
    emit(x);
}

main = identity;
```

Here we define the "identity" `kernel`; it consumes a single value from its source (called `x` in this case) and `emit`s it, pushing it to the next kernel in the `pipeline`. Since there is no further `kernel`, it simply outputs the value.

For a more illustrative example, consider the following program:

```
function avg(x, y) {        // Function definitions look a lot like 'kernel's
    return (x + y) / 2;     // Note they return a value instead of emitting one
}

kernel average_kernel(x) {
    emit(avg(x[0], x[1])); // Emit the result of calling 'avg' with the values
                           // in the first two indices of 'x'
}

kernel filter(x) {
    static index = 0;       // 'index' initially has value 0
    index = index + 1;      // Increment index
    if (index % 3 == 0) {   // Only emit every third record
        emit(x);
    }                       // 'else' condition not specified
                            // Note that this kernel may not emit any value
}

main = filter | average_kernel;
```

The input for this program would be a sequence of records consisting of two numeric values in a tuple or array. The program will output a sequence of numeric values representing the average of every third pair of values. Here we use two kernels: first, the `filter kernel` emits every third value it consumes. Second, the `average_kernel kernel` averages the two numeric values in its source value by calling the user defined `avg` function. Finally, we construct a pipeline consisting of the two `kernel`s using the "pipe" operator.

5 Stretch Goals

The reader may have noticed that `kernel`s are not much different than traditional functions. However, we believe that this language could (and should) be extended to support much more complicated `pipeline`s,

e.g. those with multiple output and input streams to split and merge input streams. Such a system could be used perform much more complicated tasks than those mentioned above, for example processing heterogenous streams of time-series data in and event-driven architecture. However, such a language and the infrastructure it would require to be useful would likely be outside the scope of this project. Every effort will be made to implement this functionality should time permit and the language has thus far been designed with this kind of eventual extension in mind.

Other features not illustrated above, such as passing function arguments by reference and supporting functions as first-class types (for closures and the like) may possibly prove easier to implement and will be included if at all possible. Finally, some sort of associative type (hashes, maps, etc.) would likely prove useful and will also be implemented if time permits.