Stream Programming Language

Andrew Clayman
&
Palak Jain

1 Introduction

In designing a programming language, one is faced with a multitude of decisions, most of which have long term consequences on its effectiveness, the efficiency, the breadth, and usability, among much else. In approaching the task at hand, we kept an open mind to the various options, looked through the problems to be solved, and carefully planned out the various aspects of a language that we thought would be best suited to solve the problems. It seemed to us that scope of the problems was fairly narrow and specific without an extreme amount of complexity. We decided to design a language that embraces simplicity and would be specialised to an extent that would allow us to efficiently solve the problems, but not so rigid as to overly constrain the range of solvable problems.

2 Language Overview

One of the first decisions we made was whether to create an imperative or a functional language. Although stream processing is a common feature of functional languages, when we thought about solving the problems, we seemed to naturally solve them in a step-by-step manner, taking the data from the streams, then manipulating it in a certain way, then outputting the altered values. Being a domain specific language, without an exceedingly complex scope, we decided to include various features that likely wouldn't be found in a commonly used programming languages, and similarly we concluded that many, often basic, features of programming languages, wouldnt really be necessary in the solutions of the type of problems given. Our language provides primitive, specialised functions to pull streams of integers from a file, and a way of selecting one of the streams by index. On the other hand, our language does not provide support for user defined functions; we decided that as long as the language provides sufficient functions for solving problems in the given domain, function creation wouldnt be necessary. For more complex problems, a function creation provision however would be very valuable. A push and pop function are also provided for working with the streams. The language follows a call-by-name evaluation strategy. Most other aspects of the language also keep true to this simplicity and specialisation based on necessity, with an emphasis on utility.

3 Syntax

The syntax of the language isnt directly modeled after a certain other language, but rather includes a mix of features found in various languages, each aspect being a fit for an intended purpose, be it for the readability and writability of the language, or what we thought would be best for the parsing and processing of it. The only way of looping provided by the language is a while loop, and the structure is similar to C based languages in that it starts with a while token, the Boolean is surrounded by parenthesis, and the procedure to be repeated is enclosed by curly brackets. Unlike these languages, statements arent ended with semicolons, but rather a new line like Python or Haskell. Indentation however is not important in our language as code blocks are grouped with use of curly brackets. The syntax of function application is like that of Haskell, the function name followed by the arguments,

though the value returned, expressed by the return token, is always encased in parentheses. Variable declaration and assignment is always performed together and is expressed as the name of the type with the first letter capitalised, i.e. Bool, List, Matrix, Int, followed by the alphanumeric variable name, an equal sign, and lastly, what is to be assigned to the variable. The getLists token represents a function used to take the streams from the file. The arguments following the "get" function are the index desired, and the variable representing the set of streams. The pop function returns the first element of the parenthesis encased stream following the pop token, and is used during an Int assignment. The push token followed by an integer or name of an integer, followed by a list or name of list, is used to add an integer to the front of a stream. The language provides an if / else statement, with a structure like that of the while loop (parenthesis for the Boolean and curly brackets for the procedure to be executed). There is no provision for an else if as we didnt think it would be necessary, and if needed it could be replicated with nesting. There is also the possibility to use an if without an else. The if / else statement is typically used in conjunction with the empty token, which is followed by a List or name of a List, and the empty function returns a Bool indicating if the given stream has no remaining elements. Lists are represented as they are in Haskell, with square brackets and comma separated elements, and an empty list with a closing square bracket directly following an opening square bracket. True and False tokens are capitalised in the language. Only basic operators (+, -, *) are provided, and there is no capability for comparing (<, >, ==).

4 Scope

The scoping of the language involves static / global variables only. A side-effect of not separating the declaration and assignment of a variable is that there are no name clashes. If a variable is to be changed, it is essentially a reassignment to the same name. A programmer can choose to either redeclare the same name, or choose a different name for a variable. The decision to not have the same variable name applied to different scopes was based on the idea of it not being necessary given the level of complexity required to solve the domain specific tasks.

5 Type Checking

The language is strongly statically typed. Types are declared rather than inferred. Again, with no separation of declaration and assignment of variables, it isnt possible for a variable to change type, though the same name can be declared with a different type. The interpreter checks that the value provided for a declaration / assignment matches the type stated in the declaration. The type of the value is checked along with some functions and statements, for example with a while loop and if / else statement, a Boolean is required and checked for. Similarly, when the streams are taken with the getLists function, the input is required to be of Integers.

6 Error Handling

If a value of the wrong type is provided within a program, the interpreter will throw an type error, telling the programmer what the type required is, along with the type actually encountered. If the last statement of a program is not a return statement, an invalid end of program error is thrown. When taking a stream by its index, there is a check to make sure the index isnt greater than the number of streams, and if so an error is thrown.

7 Additional Features / Programmer Convenience

For the purpose of program documentation and debugging the language allows commenting. Like in Haskell a comment is started with two dashes, and everything following on that line is ignored by the interpreter. Since if / else statements tend to take up a lot of space, and for some people the bracketing can be tedious, we provided a common syntactic sugar for the if / else statement of the form: (Expression) ? Statement : Statement.

8 Execution Model

The interpreter follows a big-step (substitutional) execution model. The decision for this was based on its simplicity, and the problem space not requiring the extra features small-step semantics provides, namely concurrency, non-terminating program solutions, precise management of control flow, or maximally efficient execution.

The execution proceeds statement by statement, repeating or skipping statements in a while loop, or conditionally executing statements within if / elses. At each point of execution, the interpreter holds a triple with the program statement currently being processed, the rest of the statements of the program, and the variable storage. The simple scoping of variables allows for simple storage of variables. The variable store consists of a pair of strings, the first being the name of the variable, the second representing the value. Rather than being stored as a binding with its type, the type of the value is uniquely determined from its string representation. Program state changes with the assignment / declaration of variables and when pushing to / popping from Lists. When an assignment / declaration of a previously not used variable name takes place, the name-value pair is added to the list of pairs that is the variable store. If the name has already been used, that element of the variable list is found and the value is replaced. When an existing List is pushed to or popped from, the List is found and altered in the variable store.