Specifying Input/Output by Enumeration

Walter W. Wilson and Yu Lei
Dept. of Computer Science and Engineering,
The University of Texas at Arlington

Input/output is awkward in declarative languages [2]. Some functional languages like LISP have procedural read and write operations. Prolog has ugly read and write "predicates" that execute in sequence. Haskell monads provide pure functional I/O but still involve a sequence of actions.

A logic programming language called "axiomatic language" [3,4] has a different approach that may be more declarative. Axiomatic language is intended as a way to specify the behavior of a program as seen by an external observer. It is based on the idea that any function or program -- even an interactive program -- can be specified by a static infinite set of symbolic expressions that enumerate all possible inputs -- or sequences of inputs -- and the corresponding outputs.

For a batch program that reads an input file and writes an output file, these expressions could have the following form,

```
(Program <name> <input> <output>)
```

where <*name*> is the symbolic name of the program and <*input*> and <*output*> are symbolic expressions for files. These expressions would be generated for all possible input files and the corresponding output files.

In this writeup a symbolic expression is defined as a symbol, a character in single quotes, or a sequence of symbolic expressions separated by blanks and enclosed in parentheses. We also use a character string in double quotes to represent a sequence of characters.

Consider, for example, a program that reads an input text file and writes an output file consisting of the input lines in sorted order. This program could be defined by symbolic expressions such as the following,

```
(Program Sort
  ("horse" "zebra" "donkey")
  ("donkey" "horse" "zebra"))
```

which shows the sorting of a particular 3-line input file. A set of these expressions for all possible input text files is a complete specification of the sorting program.

Suppose we want to define an interactive program that reads and writes lines of text. This can be defined by symbolic expressions that represent the inputs and outputs of an execution. The form of these expressions could be as follows:

```
(Program <name> <out> <in> <out> <in> … <in> <out>)
```

Here <*out*> is a sequence of zero or more output lines written by the program and <*in*> is a single input line typed by the user. Each Program expression thus represents a possible execution history. A set of Program expressions that covers all possible input sequences represents a complete specification of the external behavior of the interactive program.

Consider an interactive program that reads an input string and checks if the left and right parentheses are balanced. Other characters are ignored. An "Ok" is output if the input string parentheses are balanced, else an error message is written, with the message positioned at the point of the error. An empty input string terminates the program. The following is an example Program expression:

```
(Program Balanced
   ("Enter strings with balanced parentheses ...")
    "(ab()(3 2 1)x)"
   ("Ok")
    "((ABC)))()"
   ("       ^ extra right parenthesis")
    "(123"
   ("    ^ missing right parenthesis")
    ""
   ("Bye!"))
```

(Note that output strings are in parentheses since there can be multiple output lines between each user input line.) These expressions would be defined for all possible sequences of input lines.

Axiomatic language is just a formal system for defining these infinite sets of symbolic expressions. It can be described as pure Prolog with LISP syntax, HiLog [1] higher-order generalization, and "string variables". Axiomatic language programs for the sorting and balanced parentheses examples can be found at the language website [5]. These can be considered pure specifications since we are defining external behavior without having anything to say about how an algorithm would implement it. The difficult problem of transforming specifications to efficient algorithms remains to be solved.

References

[1] W. Chen, M. Kifer, D.S. Warren, HiLog: a foundation for higher-order logic programming, J. Logic Programming, Vol 15, No. 3, 1993, p.187-230.

[2] S. Peyton Jones, Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, in C.A.R. Hoare et al, Engineering Theories of Software Construction, IOS Press, 2001, p.47-96.

[3] W.W. Wilson, Beyond Prolog: software specification by grammar, ACM SIGPLAN Notices, Vol. 17, Issue 9 (Sept. 1982), p.34-43.

[4] http://www.axiomaticlanguage.org

[5] http://www.axiomaticlanguage.org/examples