Yugoslav Journal of Operations Research
6 (1996), Number 2, 277-284

SOME NOTES ON THE FORMAL DEFINITION OF STREAMS

Dušan MALBAŠKI, Dragan IVETIĆ

Faculty of Technical Sciences,
Trg D. Obradovića, Novi Sad, Yugoslavia

Abstract: The paper considers the language constructs that give a real meaning to the structures makes applied methodology itself to be easy to understand and the program becomes close to the way of thinking about the problem on hand. The concept of streams is emphasized as a means for reaching the high quality of programs. Also, there are given an informal and formal definitions of streams. Formal definition of the stream consists of sixteen axioms based on the formulas of algorithmic logic and abstract functions. The meaning and the purpose of these axioms is that every implementation that satisfies them is a stream.

Keywords: Streams, program quality, formal specification, algorithmic logic.

1. INTRODUCTION

One of the main goals of software engineering is the development of high quality programs: programs that are readable and easy to understand, modify and maintain. Program quality depends mainly on the methodology applied, and consequently on how well the language constructs coverthe structures of the methodology. The existence of such constructs that give a real shape to the structures makes the methodology itself easy to understand and the program becomes close to the way of thinking about the problem at hand. During the historical development of programming languages many such constructs have been introduced, and flows (sequences, collections) belong to an important class of them. There are several languages that use flows, the most interesting being the following:

- CLU [6] which was introduced as an aid to the methodology that decomposes a problem to recognizable abstractions. This language incorporates data abstractions, procedural abstractions and control abstractions. CLU control abstraction defines a method for sequencing arbitrary actions enabling the user to define different classes of control abstractions. For instance, CLU FOR expression defines an iteration over the collection (flow) of objects of any type whereby the collection of the next object is made on the basis of a user defined iterator. It provides the manner of decomposition to the selection of the next object (iterator) and to the part that executes the operation on the object selected.

- Alphard [11] provides the user with broad control of abstract type implementation in order to support known methodologies and to enable precise specification of program behaviour, all in order to formally verify its implementation. Two iterative constructs are present, FOR constructs for iteration over the whole flow and NEXT for retrieval. Both constructs have rigorous definitions of control abstractions (especially the algorithms for selecting the next element and termination) thus enabling formal verification.

- Lucid [2] is a non-procedural language developed for formal proof of program cerrectness. A Lucid program may be viewed as a collection of commands that describe the algorithm through assignments and cycles, but at the same time a Lucid program can be interpreted as a set of purely mathematical statements about the results and effects of the program. A Lucid cycle is realized as an iteration over a sequence of values by unary operations of access to the first, current and next element (first A, A and next A) and by the binary operation of extracting the element from the sequence (as soon as).

All these languages have a common property: the sequences of values play an important role in the process of reaching the goal (e.g. obeying the methodology rules with readability, understandability and possibility of formal verification). However, every implementation of the sequence requires specific mechanisms of control (selection, access and extraction) thus limiting the use of these and similar languages which resulted in their still being unconventional.

2. THE CONCEPT OF STREAMS

As a consequence these constructs (sequences, flows, collections) evoluted into the new concept of STREAM which, together with appropriate control mechanisms, appears to be a powerful means to implement conventional programming languages.

Unformally a stream is a sequence of values with the same type, e.g. it can be viewed as a FIFO (first - in - first - out) queue in which the elementary load operation puts elements on its end, and the elementary unload operation takes an element from

the front. The number of elements need not be known in advance, and the evaluation of the element value may be postponed until it becomes necessary.

If stream X denotes a sequence of values

$$< x_0, x_1, \ldots, \text{eos} > \tag{1}$$

then the load operation is denoted as

$$\text{next (X)} := q$$

which changes the stream into

$$< x_0, x_1, \ldots, v(q), \text{eos} >$$

where $v(q)$ is the value of element q. The special element *eos* is used to denote the end of the stream and is automatically put on its end. The unload operation of stream X is denoted as

$$q := \text{next (A)}$$

so after applying it to stream (1) it becomes

$$< x_1, x_2, \ldots, \text{eos} >$$

and q has the value x_0 . This operation is clearly destructive.

Some binary operations are also defined on streams: if X and Y are streams of type (1) and if their elements are numbers then, for example, the sum of X and Y is

$$X + Y = < x_0 + y_0, x_1 + y_1, \ldots, \text{eos} >$$

where summation is done wherever possible and the end of the new stream is controlled by the special EXCEPTION mechanism. The other arithmetic operations are defined in an analogous manner.

Relational operators (e.g. $=, \neq, <, \leq, >, \geq$) are difuned similarly.

The operations mentioned are executed (under by control of) the special control mechanism for streams. Reaching the end of either stream X or stream Y or trying to take a value from an empty stream represents an irregular situation to which the control mechanism reacts by generating an EXCEPTION. Such exception interrupts the operation and transfers control to the exception handler that handles the situation. The handler for the end of stream X is denoted by

EXCEPTION X (eos) => (* routine for processing empty stream X *)

Also, reaching an element with some special value v in the stream Y is denoted

EXCEPTION Y (v) => (* routine for processing *)

After the exception handler finishess normal processing continues. The exception mechanism enables the use of streams in programs that are decomposed into networks of process because it provides their communication and synchronization. As an illustration, consider a program R which is decomposed into two processes A and V which communicate through stream X. Process A forms stream X which is at the same time an input to process V. Then, in process V an exception handler is defined for handling the empty state X, as

EXCEPTION X(eos) => (* wait for A to prepare a datum *)

which provides the synchronization. In the case of time-critical operations it is possible to incorporate a time clause into the handler.

Among other things, streams have shown to be a good replacement for program loops. The loops are mostly used to execute action for one, some or all occurences of an abstract object. They are the most common control mechanisms in programming alhough they make programs less readable and understandable (often leading to error) and, above all, the programmer is not always able to represent a real object by a loop without anticipating its implementation.

To illustrate this, consider the problem of summing the positive elements of the set of integres S, [11]. In the majority of modern programming languages the program sequences would look like this:

A)
```
sum ← 0;
for i ← 1 step 1 until S.size do if S [i] > 0 then sum ← sum S [i]
```

B)
```
p ← S;
sum ← 0;
while p <> nil do if p.value > 0 then (sum ← sum +p.value; p ← p.next);
```

C) if the positive elements of S belong to [dg .. gg] it could be written

```
sum ← 0;
for i ← dg to gg do if i > 0 then sum ← sum + i;
```

None of these sequences is satisfactory. Firstly, all of them suggest sequential summation that is essentially not the same as abstract computation. Secondly, sequence (A) assumes vector implementation and sequence (B) list implementation. Sequence (C) does not suggest either of the two, but can be highly inefficient for cardinality that is less then gg - dg+1. In sequence (B) the selection of the next element is not separated from the action over the object, which can be a source of error (not including p ← p.next leads to an infinite loop). It would be much better if we simply write

```
sum ← 0;
for i ∈ S do if i > 0 then sum ← sum + i;
```

without anticipating the implementation of the set S.

The programming loop that solves the problem can be realized as an iteration over the stream of integers. For such purposes we use two kinds of so-called quantified streams, **X such that P(x)** and **X while P(x)** where P(x) is a predicate. For stream X defined as

$$< x_i, x_{i+1}, x_{i+2}, \ldots, x_m, \ \text{eos} > \tag{2}$$

a qualified stream **X such that P(x)** would be the stream

$$< x_{j1}, x_{j2}, x_{j3}, \ldots, x_{jn}, \ \text{eos} >$$

where $i \le j_k \le m$ and $P(x_{jk}) = T$ for each $k \in \{1, \ldots, n\}$ and $P(x_p) = \perp$ for every $p \in \{i, \ldots, m\} \wedge p \neq jk$.

The qualified stream **X while P(x)** for X defined in (2) is defined as

$$< x_i, x_{i+1}, x_{i+2}, \ldots, x_j, \ \text{eos} >$$

where $P(x_k) = T$ for each $k \in \{i, \ldots, j\}$ and $P(x_{j+1}) = \perp$. Qualified streams are used to organize STREAM-FOR expressions in the following way:

1) FOR X such that P(x) DO operation or
2) FOR X while P(x) DO operation

which guarantees execution of the operation specified for every element for qualified streams. Thus, the sequence which solves the problem of summing the positive elements of the set of integers S, where S is represented by stream S, and using STREAM-FOR expression, has the form

```
sum ← 0;
for S such that S > 0 do sum ← sum + S;
exception S(eos) => exit.
```

The concept of stream is used in Pascal-like conventional language in order to enable

1. increasing readability and understandability
2. realising the program as a set of modules which communicate through streams
3. realising concurrent programs, synchronized through streams.

Streams have also proven to be a very appropriate means to specify programs as well as to automatize of all phases of their lifetime [4]; [5]. In order to provide an all-around mathematical environment for such a use we need a purely formal definition of the stream as a mathematical object.

3. FORMAL DEFINITION OF THE STREAM

There exist many methods for formal specification of a programming object. For instance, it could be specified through a set of abstract functions (see [8]) or may be defined in terms of set-theoretical approach (such as a priority queue in [1] and [10]). The first approach has a disadvantage that does not define a state of the object in an explicite way, whereas the second approach ssuffer form the problems of essential properties of a set, e.g. unorederness. We try to avoid those problems by introducing a kind of mixed approach, where a stream is defined through its (abstract) states and operations over those states are given either in terms of conventional methematical notation or by using the formulas of algorithmic logic. The general form of the algorithmic-logic formula is

$$P \rightarrow [A]\, Q \tag{3}$$

where P and Q are predicates and A is an action (operation). The meaning of such formula is "starting from the state for which predicate P is true, after applying operation A, the object must and in a state which the predicate Q is true". Specifically,

$$T \rightarrow [A]\, Q$$

means that whatever the starting state was, after applying A for the resulting state must hold $Q = T$. Also,

$$P \rightarrow [A]\, \perp$$

means that in the state P the operation A is not applicable (thus avoiding the so called "universal error (state)" used in [8], [7]).

The formal definition of the stream must above all obey the fact that it is a FIFO queue, and second it must introduce the two most important qualifiers: such_that and while.

We start with the some kind of signature-like structures which consists of a set of abstract states S, a set of elements D, a set $B = \{T, \perp\}$ and a set of operation names which we define together with their domains and co-domains:

EMPTY: $S \rightarrow B$
IN: $S \times D \rightarrow S$
OUT: $S \rightarrow S$
belongs: $S \times D \rightarrow B$
accessible: $S \times D \times D \rightarrow B$
such_that: $S \times P(D) \rightarrow S$
while: $S \times P(D) \rightarrow S$

where P(D) is a set of predicates defined over the set of elements D.

The first three of them actually define the stream to have a FIFO structure, and the others are connected with qualifiers such_that and while. Note, also, that instead of having an universal element eos (and "universal" means that it could be of any type) we introduce a state "empty" (theoretically could be even more than one such state) for which the value of the function EMPTY is T. Also, it is obvious that all possible states can be reached from the state(s) in which EMPTY is true. Thus, the first axioms are

(**A1**) ($\exists s \in S$) EMPTY (s) = T

(**A2**) Every state can be reached from the state s with EMPTY(s) = T by the use of operation IN.

The next six axioms represent the FIFO structure of the stream. Note that, according to the nature of expressions (3) we will not explicity write the state as an argument of a function, but for other kinds of expressions we will do so. Also, according to the notation from [3] the sequence of operations $A_1,...,A_n$ in an expression of the type (3) is denoted by

$$P \rightarrow [\ A_1; A_2; \ldots; A_n\ ] Q$$

Also the expression

$$P \rightarrow [< A >] Q$$

means that the operation A is applied 0 or more times in order to make Q = T.

(**A3**) EMPTY $\rightarrow$ [IN(a); OUT] EMPTY

(**A4**) $\neg$ EMPTY (s) $\Rightarrow$ OUT (IN(a, S)) = IN(a, OUT(s))

(**A5**) EMPTY $\rightarrow$ [IN(a)] (FRONT = a)

(**A6**) (FRONT = a) $\rightarrow$ [IN(b)] (FRONT = a)

(**A7**) EMPTY $\rightarrow$ [OUT] $\rightarrow \perp$

(**A8**) EMPTY $\rightarrow$ [FRONT] $\rightarrow \perp$

Finally, the following set of axioms provides the use of qualifiers such_that and while:

(**A9**) belongs (a) $\rightarrow$ [< OUT >] (FRONT = a)

(**A10**) accessible (a, b) $\rightarrow$ [< OUT >] (FRONT = a) [< OUT >] (FRONT = b)

The axiom A9 means that in the ordinary interpretation the element a is somewhere in the queue and A10 that the element a is nearer to the front of the queue than element b (or that they are equal).

If P $\in$ P(D), e.g. P is a predicate than the following exioms complete the set

(**A11**) P(a) $\wedge$ belongs (a) $\rightarrow$ [such_that (P)] belongs (a)

(**A12**) $\neg$ P(a) $\rightarrow$ [such_that (P)] $\neg$ belongs (a)

(**A13**) accessible (a, b) $\rightarrow$ [such_that (P)] belongs (a) $\wedge$ belongs (b) $\Rightarrow$ accessible (a, b)

(A14) $(\exists a)\ (\neg P(a) \wedge \text{accessible}(a, b)) \rightarrow [\text{ while } (P)]\ \neg \text{ belongs } (b)$

(A15) $\text{accessible}(a, b) \rightarrow [\text{ while } (P)]\ (\text{belongs } (a) \wedge \text{belongs } (b) \Rightarrow \text{accessible}(a, b))$

(A16) $P(b) \wedge \text{belongs } (b) \wedge \neg (\exists a)\ (\neg P(a) \wedge \text{accessible}(a, b)) \rightarrow$
$[\text{ while } (P)]\ \neg \text{ belongs } (b)$

4. CONCLUSIONS

The set of axioms A1-A16 (provided that A1 and A2 are purely theoretical) are used to define a stream in such manner to give some insight into it structure. The meaning and the purpose of these axioms is that every implementation that satisfies them is a stream. It is also interesting to stress that when constructing the axioms we did not mention the type of the elements of set D that opens an important possibility to interprete them as operation thus leading to some new possibilities of analyzing streams as automata which will be a topic of future research.

REFERENCES

[1] Agafonov, V.N., *Program Specification*, Nauka, Novosibirsk, 1987 (in Russian).

[2] Aschroft, E. A., and Wadge, W.W., "Lucid, a nonprocedural language with iteration", *Comm. of the ACM*, 20 (1977) 519-526.

[3] Gries, D., *The Science of Programming*, Springer Verlag, New York-Heidelberg Berlin, 1981.

[4] Ivetić, D., and Malbaški, D., "Using streams in program specification based on temporal logic", *Second Balkan Conference on Operational Research*, Thessaloniki, Greece, 1993.

[5] Ivetić, D., "An approach to the formal specification as a base for program development", M.Sc. thesis, Faculty of Technical Sciences, Novi Sad, 1994.

[6] Liskov, B., Snyder, A., Atkingon, R., and Schaffer, C., "Abstract mechanisms in CLU", *Comm. of the ACM*, 20 (1977) 564-576.

[7] Lu, X.M., and Dillon, T.S., "An algebraic theory of object-oriented systems", *IEEE Trans. on Knoledge and Data Eng.*, 6 (3) (1994).

[8] Joseph Martin, *Data Types and Structures*, Prentice-Hall, 1986.

[9] Nakata, I., and Sassa, M., "Programming with streams in Pascal-like language", *IEEE Trans. on Software Eng.*, 17, (1991) 1-9.

[10] Salwicki, A., *Algorithmic Theories of Data Structures*, Lecture Notes Computer Sci., 1982.

[11] Shaw, M., Wulf, W.A., and London, R.L., "Abstraction and verification in alphard: defining and specifying iteration and generators", *Comm. of the ACM*, 20 (1977) 553-564.