Abstract:

The design of a multi-processor reduction machine, supporting a class of user-defined reduction languages, is presented.

A reduction machine is based on a "by need" or demand driven model of computation in which an instruction is executed only when a result it generates is requested. In such a machine, an instruction is a (variable length and possibly nested) expression and the machine language is referred to as a reduction language.

In the multi-processor reduction machine presented, the asynchronous operation of a processor is controlled by a swappable, user-defined, state transition table. Each table supports a particular reduction language. To ensure the harmonious operation of the processors a state transition table is generated automatically for a user, in a similar way that a parser generator is used to generate table drive parsers.

A Multi-processor Reduction Machine for User-defined Reduction Languages*

By

Philip G. Treleaven and Geoffrey F. Mole
TRELEAVEN, Philip


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1980.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 150.)

Abstract

The design of a multi-processor reduction machine, supporting a class of user-defined reduction languages, is presented.

A reduction machine is based on a "by need" or demand driven model of computation in which an instruction is executed only when a result it generates is requested. In such a machine, an instruction is a (variable length and possibly nested) expression and the machine language is referred to as a reduction language.

In the multi-processor reduction machine presented, the asynchronous operation of a processor is controlled by a swappable, user-defined, state transition table. Each table supports a particular reduction language. To ensure the harmonious operation of the processors a state transition table is generated automatically for a user, in a similar way that a parser generator is used to generate table drive parsers.

About the author(s)

Dr. P.C. Treleaven and Mr. G.F. Mole are both working at the University of Newcastle upon Tyne, in the Computing Laboratory and Electrical Engineering department respectively.
1. MOTIVATIONS

Computer architecture design has remained remarkably stable over the past thirty years (1). These designs are based almost exclusively on the von Neumann organisation with its sequential control flow model of computation. Recently, however, a number of researchers have questioned the continuing adequacy of the von Neumann architectural model. To quote from John Backus’ 1977 ACM Turing Lecture (2), "Conventional programming languages are growing even more enormous, but not stronger. Inherent defects at the most basic level causes them to be both fat and weak ...". Or as remarked by Wayne Wilner, investigating architectures for VLSI (3), "Indeed, some experts define VLSI to be that level of circuit technology for which traditional architecture becomes less advantageous than some alternative architecture".

The motivations for new architectures are coming from three particular directions:

1) new forms of programming
2) architectures that utilise concurrency
3) microelectronic circuits that exploit VLSI.

New simpler forms of programming are being investigated to provide improved programming methodologies and languages, which can theoretically reduce the complexity and cost of software. Functional or applicative languages, in which program transformations are simply the results of applying functions to their arguments, are the most well-developed class of very high level language. Traditional examples include LISP and the other Lambda Calculus-based languages, and the combinator constructs of APL. Languages such as LISP and reduction languages are "substitutive" languages. Implementation on conventional hardware is bound to be inefficient because conventional concepts contradict the concept of substitution. Functional programs are basically expressions which must be evaluated. Their benefits include that they are side-effect free, their correctness is easier to verify and a program can be easily decomposed for concurrent execution.

The study of architectures that utilise the concurrency in problems is motivated by the need to increase the performance of computers (4), while noting that the natural physical laws place fundamental limitations on the performance increases obtainable from technology alone. Currently, novel multi-processor architectures are being proposed which are intended to be general-purpose, as opposed to special-purpose high-speed computers such as Cray 1 and Iliac IV. For this reason, these novel architectures are based on new "naturally concurrent" models of computation, in which program evaluation is either data driven (5-11) or demand driven (12-15). In data driven models an instruction is executed immediately its input data is available, whereas in demand driven models an instruction is
executed when a result it generates is requested. Interestingly, both classes of computer are conveniently programmed in functional languages.

Investigations of microelectronic structures that exploit VLSI are driven by the rapidly increasing levels of integration of functions on silicon. This potential of VLSI may be realised either through (i) "customised" chips (16) that are programmed to perform some specialised task, or (ii) "single part" systems (3). These customised chips, similar to programmable logic arrays, will be field-programmable and might even popularise do-it-yourself chip design. The single-part system approach, on the other hand, attempts to reduce design and fabrication costs of systems by employing one "universal" processor/memory chip. To be successful, the basic chip in either approach needs to be simple and versatile, and must be easy to interconnect to its duplicates for increased performance. For such chips to be used to build larger computers it is also necessary to program and control them in a decentralised, concurrent manner.

From an examination of the current motivations for new architectures, we believe that a machine organisation will emerge in the next few years to supersede the von Neumann model. It also seems likely that such a machine will, by necessity, utilise VLSI to implement a concurrent architecture and be programmed in a functional language. At Newcastle University we are actively seeking such a machine organisation. One organisation being investigated is a multi-processor reduction machine — in this paper we present the system architecture and concepts embodied in the computer.

The term reduction machine is used here to cover a class of machines that process tree-structured expressions defined in terms of a reduction language, the instruction set of the reduction machine. In addition, these machines have a "by need" or demand driven form of program evaluation where an instruction is executed when a result it generates is requested.

A few designs for reduction machines have been proposed, notably by Berkling (12,13), Mago (14) and Keller (15). The Berkling design, in fact, has been built and is operational at GMD in Bonn. The machine consists of a single stack-based processor which, to evaluate an expression, repeatedly scans it, moving the expression between the processors' stacks. The other two design proposals are influenced by the single-part system approach of VLSI. Each part is a primitive processor/memory pair and the system is configured into a tree-structure onto which an expression to be evaluated is mapped.

The multi-processor reduction machine design presented here, falls somewhere between the Berkling design and that of Mago or Keller. Each processor in the machine operates in parallel on the expression being evaluated, attempting to find a reducible sub-expression. The operation of each processor is controlled by a swappable, user-defined, state transition table. We believe the novelty of the design is in that (i) the machine can support a class of user-defined reduction languages, and (ii) the use made of parser
generator concepts to produce the state transition table for a given reduction language. Other interesting topics are the protocols used to synchronise the access of the processors to the expression, and also to avoid processor deadlock and starvation.

Section 2 describes the syntax and semantics of reduction languages, and their evaluation. A simple example language is also introduced. Section 3 describes our multi-processor reduction machine and presents the protocols which control its operation. Finally, in section 4 we briefly present the design of the software which generates the state transition tables and discuss its operation.

2. REDUCTION LANGUAGES

Reduction languages (17,18) are a rigorous attempt to relieve the programming problems, such as explicitly specifying flows of control and managing memory cells, normally associated with conventional computers. The style of programming is strictly functional, based on a few elementary mathematical constructs featuring a binary tree structure, from which complex expressions are built up by recursive application.

A reduction language program may be viewed as a set of definitions, name:expression pairs, and an expression to be evaluated. Evaluation is by a substitution process which traverses the expression and successively replaces reducible subexpressions by others that have the same meaning, until a constant expression representing the result of the computation is reached. These basic concepts are illustrated in Figure 1 by means of bracketed arithmetic expressions whose tree structure differs from that normally associated with reduction languages, but may be more familiar to the reader.

Figure 1 consists of three parts, namely the definitions and the expression to be evaluated, and the tree structure of the expression during two schemes of reduction. The tree is in the general form:

```
operator
  /   |
left subexpression right subexpression
```

representing the corresponding fully bracketed expression. An expression is reducible when both the left and right subexpressions are numbers or when the name of a definition is encountered. This evaluation continues until a constant expression is obtained, as for example with the value 12 in Figure 1. The number of subexpressions which can be reduced in parallel depends both on the expression being evaluated and on the organisation of the reduction machine. Figure 1 shows both a sequential pre-order (i.e. left most inner most) evaluation and a fully parallel evaluation.
Since a reduction language permits a strictly functional style, not unnaturally its most fundamental form is:

```
   tree
   | apply to
   | function
   | apply to function argument
   | argument
```

Here function and argument may themselves be non-trivial tree-structured expressions, and the "apply to" operator is regarded as a constructor that relates the two subexpressions.

For a reduction language to act as the instruction set of a machine, it is necessary that all computational problems be representable in the language. Hence the reduction language must contain primitives for: (i) arithmetic and logical operations, e.g. + 10 12, (ii) conditional operations, e.g. IF p t f, (iii) iteration and recursion, e.g. WHILE p t, and (iv) list manipulation, e.g. CAR x. In addition the language must provide binding operations of the type "call by value", which associates a name with a value, and "call by name", which associates a name with an expression. Such a complete set of primitives is given, for example, by Backus (2).

As stated previously, one of the aims of our project is to investigate a reduction machine that will support a class of reduction languages. Such a class of languages might vary from expressions being represented in infix as opposed to pre-order, to context sensitive languages equivalent in complexity to Pascal. For the purposes of illustration in this paper we will use a simple language with the following syntax:

\[
\begin{align*}
\text{<expression>} & ::= \text{<operand>} | \text{<operator>\{<expression>\}} \\
\text{<operand>} & ::= \text{<value>} | \text{<name>} \\
\text{<operator>} & ::= + | - | \ldots | \text{LOAD} | \text{STORE} | \text{APPLY}
\end{align*}
\]

where \{ \ldots \} defines zero or more repetitions of the enclosed material. Here \text{<operator>} defines the standard primitives for arithmetic and conditional operations etc, LOAD and STORE which are operations on definitions, and APPLY which applies a function to its arguments. (Note the inclusion of a STORE operator, to increase the generality of the machine, means that programs are not necessarily side-effect free.) A subexpression in this language is considered reducible when it has the following format:

\[
\text{<reducible expression>} ::= (\text{<operator>}\{\text{<operand>}\})
\]

and evaluation halts when only an \text{<operand>} is left in the expression.
A reducible arithmetic or logical expression consists of an operator followed by one or more values. When reduced, the operator is applied to the values and the subexpression is replaced by the result. For example "(+ 4 2)" is replaced by "6". A reducible LOAD subexpression consists of a load operation "L" followed by a definition name "n". When evaluated the subexpression "((L n) n)" is replaced by the definition corresponding to the name. A reducible STORE subexpression comprises a store operator "S" followed by a name "n" and a value "v". Reduction of the expression "((S n v))" causes the name to be bound to the value and any definition previously associated with the name to be deleted. In this language "((S n v))" is deleted from the expression; alternative strategies would be to replace it by the name "n" or the value "v". Finally, a reducible APPLY subexpression consists of an APPLY operator, followed by the name of a definition "f" and its arguments "x". The subexpression (APPLY f x) is replaced by the definition "f" in which all occurrences of its formal arguments are replaced by "x".

Figure 2 illustrates the reduction language using a simple example which evaluates the definition of "a" and then overwrites its definition with the result. In Figure 2 the program is represented in pre-fix form, with LOAD (i.e. L) and STORE (i.e. S) operators being specified explicitly. At stage 1 of the evaluation only the LOAD is performed since the second operand of the STORE is not an operand. Substitution then continues, in parallel, till stage 4 when the addition and subtraction are reduced. This causes the multiply to become reducible at stage 5. Finally, at stage 6, the value of a, 12, is stored to become the new definition.

It is appropriate at this point to examine some of the subtleties of the simple reduction language and a multi-processor reduction machine. Firstly, fully bracketed expressions are used. These provide sequencing during evaluation and allow the evaluation to start in parallel at an arbitrary number of points. Secondly, the brackets are used in conjunction with the LOAD and STORE operators. Conceptually, all items in the expression being evaluated are visible and reducible subexpressions are processed in parallel. The brackets are used to synchronise the LOAD and STORE operations.

LOAD and STORE operators can also be used to support "call by name" and "call by value" parameter passing mechanisms; as shown below:

1) (APPLY f x) - call by name
2) (APPLY f (L x)) - call by value

Having briefly presented this reduction language, in the next section we examine the reduction machine.

3. REDUCTION MACHINE

The machine functions necessary to evaluate reduction language expressions may be derived from the discussion of language primitives.
in the previous section. Basically we need storage mediums for both the expression and the definitions, a means of controlling the traversal of the expression and recognising a reducible subexpression, a processor to perform this reduction, and some means of resuming the traversal.

Figure 3 shows the system architecture of our multi-processor reduction machine. It consists essentially of three major parts, (i) a common memory containing the definitions, (ii) a set of identical, asynchronous, processing units (PU), and (iii) a large segmented shift register containing the expression to be evaluated. This shift register comprises a number of double ended queues (DEQ) containing the parts of the expression being traversed, and a backing store to hold surplus parts of the expression. Each processor has direct access to the common memory and two double ended queues.

The major difficulty in designing a multi-processor reduction machine is synchronising the access of the asynchronous processing units to the expression under evaluation. In the proposed design, this synchronisation is achieved by the double ended queues which perform three roles. Firstly, they act as a buffer for part of the expression between two processing units. Secondly, they maintain the order of the expression. Lastly, only one processing unit can read the queue at a time and remove an item, so it is impossible for two processing units to examine the same item in an expression simultaneously.

The architecture of an individual processing unit is shown in Figure 4. It consists of four registers, containing information on the subexpression being traversed, the reduction table which contains the user-defined state transition table controlling the evaluation, the action unit performing the actions specified by the reduction table, and the operation store holding user-defined code for the action unit. A processing unit, in simple terms, is like a micro-programmed processor with the reduction table and operation store corresponding to a swappable microprogram (16).

Each processing unit can read or write to either of its double ended queues, the current direction (left or right) being maintained by the direction register. When an item is read or removed from a queue it is transferred into the input register. Associated with each item is a type field indicating whether the item is an operator, operand, bracket, etc. This input type information, together with the current state defined by the state register, is used to index into the reduction table. The reduction table entry defines an action to be performed, and new values for the state and direction registers. Lastly, the buffer register holds items of a subexpression previously read by the processing unit.

The basic aim of each processing unit is to traverse the expression attempting to build up a reducible expression in its buffer register. When a processing unit determines that the current expression in its buffer register can never be part of a reducible expression, e.g.
buffer | input | direction | state  
---|---|---|---
(+10) | () | right | -

It outputs the buffer register's contents (to the left in this case) and attempts to find a new reducible expression:

buffer | input | direction | state  
---|---|---|---
() | - | right | -

The operation of a processing unit for a given reduction language is controlled by the state transition reduction table. Figure 5 shows a reduction table for the simple language introduced in the previous section. For each of the input types of the language - empty "E", operand "X", operator "F", left bracket "(" and right bracket ")" - there is a corresponding column in the reduction table. Each row of the table corresponds to a state when searching for the following reducible expression:

\[
(\langle\text{Operator}\rangle\{\langle\text{Operand}\rangle\})
\]

Again \{\ldots\} defines zero or more repetitions of the enclosed material. Thus, if a processing unit is in state 3, reading from the right hand queue, and a right bracket is input a reducible expression has been found.

Recall, each entry in the reduction table, associated with the intersection of a given row and column, specifies an action to be performed, together with a new state and a new direction. A new direction has four possible settings: "S" means keep the same direction, "C" means change direction, while "R" and "L" explicitly give the new direction as right and left, respectively.

In Figure 5 seven actions are used and their operation is shown below. Each action is followed by an implicit read from the current queue into the input register.

1) wait - the "no operation" action, used to wait on an empty queue.

2) pass - outputs the contents of the input register to the queue not being read from. This action is used to scan for the start of a reducible expression.

3) push - transfers the contents of the input register to the appropriate end of the buffer register. Used during the building of a reducible expression.

4) pop - transfers the contents of the buffer register to the queue not being read from, moves the input to the buffer register. Used when the traversal of a reducible expression fails and a new one must be started.
5) reduce - invoked when a reducible expression is recognised. It outputs the reduced expression to the queue not being read from and returns the processor to the empty state.

6) error - used when the expression being read fails to satisfy the syntax of the reduction language.

7) give - transfers the contents of the buffer register into the queue being read from. This action is used to avoid deadlock.

Although these actions will serve for a variety of reduction languages they are user-defined, each being associated at the action unit with a particular routine. This routine is defined in terms of basic processor operations such as load register, read input, etc.

As an illustration of the operation of a processing unit, for the reduction table in Figure 5, Figure 6a shows a unit when it is about to recognise a reducible subexpression. The processing unit has just read a right bracket from the right queue (direction R) and has previously read the items "( L c", which are stored in the buffer register, placing it in state 3. An enclosing expression "( + 4" and ")" are in the left and right queues, respectively.

The action extracted from the reduction table is "reduce (1/S)'", which causes the contents of the buffer register to be interpreted and the definition of "c" to be loaded into the left queue. The processing unit next goes to the empty state "1", but continues to read the right queue "S". On reading the right bracket from this queue, the action "push (4/L)" is executed as shown in Figure 6b, this causes the processing unit to reverse direction and read in the remainder of the reducible expression from the left queue.

In the next section we examine further the generation of reduction tables, in particular the implicit protocols they must obey for the parallel traversal of expressions and for the avoidance of deadlock and starvation.

4. REDUCTION TABLE GENERATION

The flexibility offered by a multi-processor reduction machine supporting user-defined reduction languages raises a number of problems. Firstly, a reduction table must work in conjunction with other, possibly different, reduction tables. Secondly, there is the difficulty of locating reducible expressions in parallel. Thirdly, work - the expression to be reduced - must be distributed amongst the processing units. Fourthly, the units must avoid deadlock, starvation and saturation. Finally, it is necessary to minimise the size of a reduction table for a reduction language with a number of alternative reductions.
The majority of these problems can be overcome by identifying protocols to incorporate in the reduction table. For instance, searching for a reduction when a processing unit's buffer is in the empty state involves first locating a delimiting bracket. All other non-empty inputs are passed across to the adjacent queue. This protocol also helps other problems. To encourage the distribution of work amongst the processing units: we minimise the number of items of a potentially reducible expression held in a buffer register, output the contents of a buffer register immediately it proves not to be the innermost reduction and always output substituted definitions, returning a processing unit to the empty state.

When two adjacent processing units get into the position of each having part of a common reducible expression, deadlock is avoided by one of the units giving up its part of the expression. This is supported by the symmetrical actions "wait" and "give" which are used when an empty input (E) is detected. "wait" polls the empty queue while "give" releases the contents of the buffer register. In Figure 5, for example, when a processing unit is in state 4 or 5 reading left and an empty queue is detected, deadlock is possible since the adjacent unit may be reading right in states 2 or 3. Deadlock is avoided by the processing unit reading left, giving up the contents of its buffer register and changing direction. In this example, the decision as to which processing unit will "wait" or "give" up its objects is dependent on the location of the left bracket "('. Related to the problem of deadlock is the saturation of a DEQ - when the required capacity of a DEQ is exceeded. This seems best handled by including a "full" state in the state transition table which might make use of "wait" actions.

Having identified such a set of protocols for inclusion into a reduction table, there is the difficulty of enforcing their use. This is overcome in our proposal by using a reduction table generator, a software package, to automatically generate a reduction table for a user. The reduction table generator employs ideas similar to compiler-compilers that are used to generate table-driven LR parsers. Figure 7 shows the structure of the reduction table generator. Into the generator a user puts three blocks of information, (i) a BNF syntax of the reduction language, (ii) a list of the input types, and (iii) a BNF syntax of the possible reduction rules. From this information the generator produces a reduction table for the language and the action code which includes the necessary primitives that enforce the protocols.

Recall, a column of the reduction table is a possible input type and a row is a possible state encountered during the traversal of a reduction. Whereas constructor algorithms for LR parser generators construct the complete set of states that can be encountered during a left to right parse, a reduction table generator has to consider processing in either direction, and not all states will occur in the table.

Using the BNF syntax of the reduction rules, the generator first derives all valid possible states that can be encountered when
searching for a reducible application and stores them in a "state table". The rows of this state table are unique type sequences which determine some part of a reduction rule. The idea is to take one type sequence at a time and generate one row of the reduction table, in close collaboration with the syntax of the reduction language. This determines what item is allowed to follow a sequence, given the direction of processing. If there is more than one reduction rule partially defined by a particular type sequence, then all such rules are considered when trying to detect a reduction. Full details of the structure and operation of the reduction table generator are given in reference (20).

5. CONCLUSION

This paper has presented the system architecture and concepts underlying a multi-processor reduction machine supporting user defined reduction languages. The aim of the project, of which this investigation forms part, is to identify a machine organisation which will supersede the von Neumann model and allow VLSI to be utilised to expand our potential for effective computing. The major advantage of a reduction machine design is its underlying sound mathematical basis.

The investigation of our table-driven reduction machine is at an early stage. We have constructed a simple simulator for the machine and a reduction table generator for a class of simple expression languages (20), such as the language used for illustration in this paper. The exercise has demonstrated the initial feasibility of the concepts. At the next stage of the investigation we hope to design and implement a simple hardware simulator for the machine using conventional microprocessors.

ACKNOWLEDGEMENTS

Although only two names appear on the list of authors of this paper, the ideas presented were generated by a group of people while investigating the characteristics of reduction machines. The other members of the group are Noordin Ghani, Richard Hopkins, Simon Jones and Paul Rautenbach. Lastly, the UK Science Research Council are acknowledged for funding this research on computer architecture.

REFERENCES


(3) W. Wilner, "Recursive Machines". Xerox PARC, SSL Internal Memorandum (January 1978).


(14) G.A. Mago, "A Network of Microprocessors to Execute Reduction Languages". Int. Journ. of Computer and Information Sciences, Vol. 8, No. 5 and Vol. 8, No. 6 (1979).


a) program representation

<table>
<thead>
<tr>
<th>definitions</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t1 = (b + c) )</td>
<td>( (t1 \ast t2) )</td>
</tr>
<tr>
<td>( t2 = (b - c) )</td>
<td></td>
</tr>
<tr>
<td>( b = 4 )</td>
<td></td>
</tr>
<tr>
<td>( c = 2 )</td>
<td></td>
</tr>
</tbody>
</table>

b) pre-order evaluation

\[
\begin{array}{c}
\ast \\
/ \\
/ \\
| t1 | t2 + t2 + t2 + t2 + t2 + t2 6 t2 \\
\ast \\
/ \\
/ \\
| b c 4 c 4 2 |
\end{array}
\]

\[
\begin{array}{c}
/ \\
/ \\
| 6 - 6 - 6 - 6 - 6 2 12 |
\ast \\
/ \\
/ \\
| b c 4 c 4 2 |
\end{array}
\]

c) parallel evaluation

\[
\begin{array}{c}
\ast \\
/ \\
/ \\
| t1 | t2 + t2 + t2 + t2 + 6 2 12 |
\ast \\
/ \\
/ \\
| b c b c 4 4 2 |
\end{array}
\]

Figure 1: Reduction Evaluation of a Simple Arithmetic Expression.
a) program representation

\[ t_1 = (+ (L \ b) (L \ c)) \]

\[ t_2 = (- (L \ b) (L \ c)) \]

\[ b = 4 \]

\[ c = 2 \]

\[ a = (* (L \ t_1) (L \ t_2)) \]

b) parallel evaluation

<table>
<thead>
<tr>
<th>stage</th>
<th>state of evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(S a (L a))</td>
</tr>
<tr>
<td>2</td>
<td>(S a (* (L t_1) (L t_2))</td>
</tr>
<tr>
<td>3</td>
<td>(S a (* (+ (L b) (L c)) (- (L b) (L c))))</td>
</tr>
<tr>
<td>4</td>
<td>(S a (* (+ 2 (- 4 2))))</td>
</tr>
<tr>
<td>5</td>
<td>(S a (* 6 2))</td>
</tr>
<tr>
<td>6</td>
<td>(S a 12)</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 2: Usage of Brackets and LOAD/STORE Operators.
DEQ = double ended queue
PU = processing unit

Figure 3: Multi-processor Reduction Machine Architecture.
Figure 4: Processing Unit.
<table>
<thead>
<tr>
<th>Direction</th>
<th>State</th>
<th>empty (E)</th>
<th>operand (X)</th>
<th>operator (F)</th>
<th>brackets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>wait(1/S)</td>
<td>pass(1/S)</td>
<td>push(2/R)</td>
<td>push(4/L)</td>
</tr>
<tr>
<td>right</td>
<td>(</td>
<td>wait(2/S)</td>
<td>error(1/C)</td>
<td>pop(2/S)</td>
<td>error(1/C)</td>
</tr>
<tr>
<td>3</td>
<td>{X}</td>
<td>wait(3/S)</td>
<td>push(3/S)</td>
<td>error(1/C)</td>
<td>reduce(1/S)</td>
</tr>
<tr>
<td>left</td>
<td>{X}</td>
<td>give(1/C)</td>
<td>push(4/S)</td>
<td>push(5/S)</td>
<td>error(1/C)</td>
</tr>
<tr>
<td>5</td>
<td>F{X}</td>
<td>give(1/C)</td>
<td>error(1/C)</td>
<td>reduce(1/S)</td>
<td>error(1/C)</td>
</tr>
</tbody>
</table>

Format of table entries: action (new state new direction)

1 - 5

- S same direction
- C change direction
- R right
- L left

Figure 5: Example Reduction Table.