PERGAMON Computers and Mathematics with Applications 39 (2000) 127–143

An International Journal
computers & mathematics
with applications

www.elsevier.nl/locate/camwa

Building a Model of a Useful Turing Machine

J. A. PIOTROWSKI
School of Computing and Information Technology
University of Western Sydney—Nepean
P.O. Box 10, Kingswood, NSW 2747, Australia

(Received January 1995; revised and accepted March 1999)

Abstract—The principal ideas of a universal computer are gradually introduced into a functional model of the Turing machine. The strict sequentiality of this model is confronted with parallelism observed in real computers.

This paper follows up the presentation [1] and is complemented by [2,3].

Keywords—Education, Turing machine, Functional programming, Models, Parallelism, Sequentiality.

1. INTRODUCTION

Abstract machines, formal languages, and methods are well established in computer science, but their importance is often questioned by students. A successful introduction of an abstract machine, reported by Piotrowski in [3], caused considerable interest. Consequently, students wanted to see how such a theoretical system could have become a 'blue-print' of a computer.

This paper presents a conventional computer as the Turing machine. It is a very simple model, yet the set of instructions executed by this machine forms a core subset of any one-address assembler. A machine, which was compared with a dataflow machine in [1], used roughly the same language as the one presented here. However, the intention of this presentation is not to implement an assembler of any kind, but to show the close relationship of computers used nowadays with an original idea based on the concept of a recursive function.

A composition of functions is used extensively in this paper. The composition of Turing machines, which is based on this notion, allows to define neatly a relatively complex *real* machine as the Turing machine.

Names are carefully selected to play an explanatory role. They are introduced in a hierarchical manner. These names may represent memory sections, such as registers, or other components of a real computer. These names are introduced to indicate certain potential for parallel access or parallel operation. The use of the composition of functions on the other hand, emphasizes explicit sequentiality.

The presented model is a system of functions and, although the support of computer in the process of defining these functions is formally unnecessary, it proved to be very helpful in validat-

0898-1221/1999/$ - see front matter Typeset by AMS-TEX
PII: S0898-01221(99)00319-3

ing the model[1]. For this reason, the syntax of Miranda[2] is adhered to, as this particular system has been used for the purpose of verification. Its language is very similar to a conventional mathematical notation, at least as it is used in this paper[3]. The expected level of acquaintance with Miranda is the same, as for example, in [1]. The more systematic introduction of the language can be found in an introductory paper by Turner [4], for example.

This presentation is an outline of a lecture on a general purpose computer viewed as a formal system. It has been prepared for an upper-division undergraduate subject on fundamentals of computer systems, and is normally supported by the laboratory session similar to the ones described in [1–3].

2. TURING MACHINES

2.1. Definitions

The issue of automated computing and a problem of computability lead to the first concept of an abstract machine. Turing [5] and Post [6] independently introduced a definition, known as a Turing machine. It has an obvious interpretation and this makes it an often used reference point in theoretical dissertations, as well as in popular publications.

The Turing machine M is a six-tuple

$$M = (S, \Sigma, \delta, s, B, Y),$$

where

S is a finite set, called a set of *states*;

$s \in S$ is called the *initial* state;

$Y \subseteq S$ contains *final* states;

Σ is a finite set of *symbols*, and is called the *tape alphabet*;

$B \in \Sigma$ is a special symbol called a *blank*;

$\delta : S \times \Sigma \to S \times \Sigma \times \{\leftarrow, \rightarrow, \downarrow\}]$ is a partial function.

A particular Turing machine is interpreted in the context of a sequence of symbols[4] $t \in \Sigma^*$, and a general driver function

$$\mathcal{D} : S \times \Sigma^* \to Y \times \Sigma^*,$$

which maps a pair $(s, t) \in S \times \Sigma^*$, called the initial *configuration* of the machine M, into a final configuration of this machine. It is usually achieved by repeated application of the transition function to a sequence of symbols on the tape.

A set of *internal* states S' of a machine M is a subset of S such that $S' \cap (\{s\} \cup Y)$ is empty. Let $M_1 = (S_1, \Sigma_1, \delta_1, s_1, B_1, Y_1)$ and $M_2 = (S_2, \Sigma_2, \delta_2, s_2, B_2, Y_2)$ be two Turing machines. For $t \in \Sigma^*$ such that $\mathcal{D}(s_2, t) = (y_2, \tau)$ and $\mathcal{D}(s_1, \tau)$ are both defined, we say that for this t the *composition* of M_1 and M_2 is defined. This model consists of a number of lower level machines joined by the composition operation. In all instances but one, the final state reached by the lower level machine is not important[5]. In the case of the machine represented by the function **decoding**, the final state must be preserved, because it is used in the next stage as the initial state of another machine.

All lower level machines have disjoint sets of internal states and use the same tape alphabet $\Sigma = \Sigma_1 = \Sigma_2$ and the same blank symbol $B = B_1 = B_2$.

---

[1] A set of definitions presented in this paper is available in electronic form at **http://www.cit.nepean.uws.edu.au/~jerzy/papers/Turing/**.

[2] Miranda is a trademark of Research Software Ltd.

[3] All definitions in this article are given in Miranda, but they can be translated easily into Haskell, or its derivatives such as Hugs or Gofer. A program **mira2hs** written by D. Howe can help in this work.

[4] It is often called the *tape*. Post calls it a symbol space, which "is to consist of a two way infinite sequence of spaces or boxes" [6, p. 103], while Turing makes a direct reference to a paper tape [5, p. 231].

[5] Hence, the same name **Done** is used in all such situations.

2.2. Model's Assumptions

A possibility of experimentation with models is usually appreciated by students. Hence, definitions are expressed in a functional language which allows them to evaluate functions individually.

The Turing machine is commonly interpreted as a device with a read/write head able to move along the tape, which is of unlimited length. At any given time, the machine is in a certain state, which can change after the move. The tape is subdivided into cells, each capable of storing a symbol.

Apart from initial and final states, each machine often has some internal states. The following lines introduce states as an algebraic data type state. Not all states are listed below[6]. This is only a sample.

```
state ::= Ahead |
          Done |  LD | ST | ADD | SUBT | INCR | HLT | JMP | JZ |
          Eval | E0 | E1 | E00 | E01 | E10 | E11 | ...
```

The first on this list is the state Ahead. It is the initial state of the lower-level machine which moves the read/write head to the last 'bit' of a current 'word'. The name of this machine is ahead (this and similar machines will be described in Section 2.3).

The state Done is the final state and is used by all machines with the exception of the machine decoding. This latter machine has the initial state Eval, internal states such as E0, E1, E00, and final states such as LD, ST, etc. The list of all final states is called final.

```
final  :: [state]
final  =  [Done,LD,ST,ADD,SUBT,INCR,HLT,JMP,JZ]
```

The *tape alphabet* Σ is called a symbol[7]. It could have been limited to two distinct symbols (like Zero and One), with a Blank as the universal separator B. However, increasing the number of symbols makes definitions easier to understand, and allows one to think in more abstract terms. The symbols used in experiments are as follows:

```
symbol ::= Zero | One | Blank |
           ZERO | ONE | ZerO | OnE | ZERo | ONe |
           D | ACC | PC | MAR | IR | OP | MM
```

The symbols Zero, One, and their look-alikes, such as ZERO and ONE[8], are not given any meaning as yet, but a numerical interpretation is the most natural. On graphs, they are represented by 0, 1, 0', and 1'.

The remaining seven symbols help to define the architecture of the computer.

It is assumed the tape is unlimited in both directions and is represented by two lists and a single symbol, which is called a window[9]. The read/write head can see only one cell at a time, and this cell is called a window.

```
tape       == ( lside_tape, window, rside_tape )
lside_tape == [symbol]
window     ==  symbol
rside_tape == [symbol]
```

Having defined the state and the tape, one can define[10] a configuration on which the driver function acts upon.

```
configuration ::= Config state tape
```

[6] A complete list can be found in Appendix 2.

[7] This is an example of the Miranda definition introducing a new type (likewise the definition of the **state**). The **symbol** is a name of a user-defined, algebraic type whose elements belong to the following set { **Zero**, **One**, **Blank**, **ZERO**, **ONE**, **ZerO**, **OnE**, **ZERo**, **ONe**, **D**, **ACC**, **PC**, **MAR**, **IR**, **OP**, **MM** }.

[8] The latter ones play a role of place-markers.

[9] These definitions introduce new names for composite types known as type synonyms. They do not offer the same protection as algebraic types.

[10] The type constructor **Config** combines two types: **state** and **tape** into a new entity, and labels it as something distinct from other objects. It is a more general use of the algebraic type definition.

There are three moves: `Left`, `Right`, and `None`,

```
move ::= Left | Right | None
```

which correspond to $\leftarrow$, $\rightarrow$, and $\downarrow$, respectively, (Section 2.1).

When the head moves to the `Right`, for example, the first element[11] of a list `rside_tape`, becomes visible in the `window`. The functions[12] `hleft` and `hright` take the `tape` as an argument and return its new content as a result. They represent the *head* moving to the left and to the right, respectively.

```
hleft, hright :: tape -> tape
hleft  ([]   , c, rs) = ([], Blank, c:rs)
hleft  (c':ls, c, rs) = (ls, c'   , c:rs)
hright (ls, c, []   ) = (c:ls, Blank, [])
hright (ls, c, c':rs) = (c:ls, c'   , rs)
```

These functions lengthen the `tape`, if necessary filling the newly appended memory locations with the `Blank`.

Atomic actions of the machine are defined by the *transition function* `tr`. They depend on the symbol actually seen in the `window` and also depend on the `state`. In the process of execution of such actions, the machine changes its state, leaves a symbol in the window, and moves the head. There are many transitions defined in the sequel, and all of them are of the type

```
tr :: state -> window -> ( state, window, move )
```

The function `tr` defines transitions, but we need the `driver` function to put them into sequences[13]. This function recursively calls the function `tr`, redefines the `tape`, changes the `state`, and 'stops' when the final state is reached.

```
driver :: configuration -> configuration
driver (Config s t) = (Config s t),      if member final s
                    = driver (Config newS newT), otherwise
                      where (l, w, r) = t
                            (newS, neW, dir) = tr s w
                            newT' = (l, neW, r)
                            newT  = hleft  newT', if dir = Left
                                  = hright newT', if dir = Right
                                  =        newT', if dir = None
```

The initial state is singled out in the definition of the Turing machine and it is its important, defining component[14]. The function `tm` accepts the initial state as its first argument.

```
tm :: state -> configuration -> configuration
tm newS (Config s t) = driver (Config newS t)
```

This allows one to refer to it in a short way[15], and for example `tm Dlim` represents the Turing machine starting from the initial state `Dlim` (see also the description in the following section). In this way, the configurations and their changes are given a lesser significance as they are pushed into a background.

---

[11] It is defined as `(hd rside_tape)`, which reads: the head of the list `rside_tape`.

[12] These functions describe outcomes for different patterns of arguments. The first line introduces two names of functions: `hleft` and `hright`, both of the same type `tape -> tape`. The following line defines the new content of the `tape`, when there was nothing to the left of the *read/write head* (an empty list `[]`). The `Blank` is routinely added for convenience, but it is not strictly necessary. Patterns can be used as arguments of functions and their elements can occur in the right-hand expressions (`c'` in the second line of the `hleft`). Such definitions are often shorter than equivalent guarded expressions.

[13] It corresponds to the function $\mathcal{D}$ from Section 2.1.

[14] "One box is to be singled out and called the starting point." [6, p. 103].

[15] An idea of making a function to accept its arguments one at a time, or in other words, that it suffices to consider only functions of *one* argument, was first suggested by Frege in 1893 [7] and independently by Schönfinkel [8]. Curry and Feys [9] later exploited this restriction and introduced notation `(f x y)` to denote `((f x) y)` which replaced an older one: `f (x, y)`. Thus, a *function application* is seen as a left associative operation.

2.3. Examples

The transition function and a set of states play a central role in the definition of the Turing machine. While the state determines a stage in processing, the transition function provides a prescription for some 'action'. A hierarchy can be introduced by considering certain states to be more important than others. Such states are made initial and together with accompanying transition functions define building blocks, which can be connected by means of the composition of machines to form a higher-level building block.

This section shows how lower-level machines are combined into a more complex one. The lower-level machines are the `rewind`, the `on`, and the one which searches for a special symbol `PC`[16]. They are combined into a higher-level machine, called the `pc`. Its purpose is to find a program counter and mark it as being *in use*.

The first machine moves the head to the left of the tape in search of a delimiter `D` (Figure 1a). This symbol marks the end of the working area on the tape. The initial state of this machine is called `Dlim` and its transition function is defined as follows:

```
tr Dlim  s = (Done, D, Right), if s = D
           = (Dlim, s, Left ), otherwise
rewind = tm Dlim
```

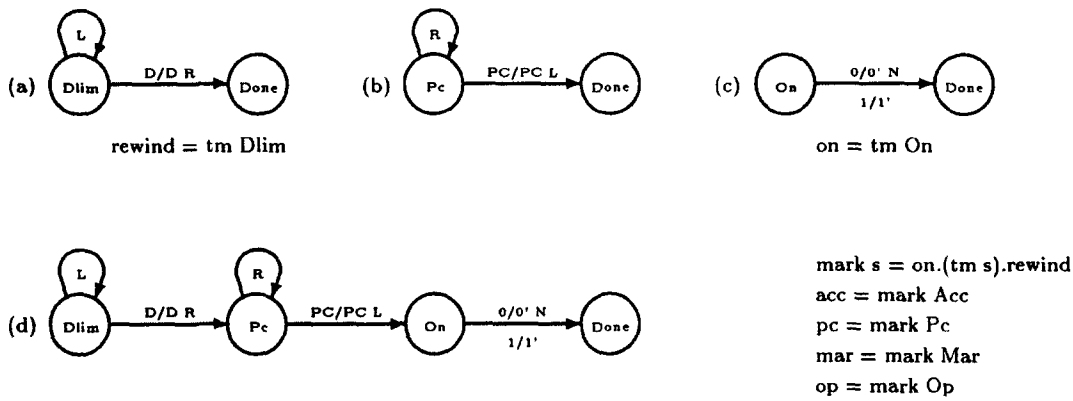

Figure 1. Setting markers.

The second machine seeks a particular identifier `PC` by moving the head to the right (see Figure 1b).

The purpose of the third machine is to mark a sequence of symbols on the tape, as active. A convention adopted in this paper tells one to change `Zero` into `ZERO` and `One` into `ONE` to mark a particular section:

```
tr On Zero  = (Done, ZERO, None)
tr On One   = (Done, ONE , None)
```

On the graph (c) in Figure 1, it is encoded as `0/0'` and `1/1'`.

Having these three machines, one can make their composition. Part of the tape, which is identified by the name `PC`, can be then marked as active by the following function:

```
pc = on.tm Pc.rewind
```

A combined graph (d) in Figure 1 represents the outcome of the composition of these three machines.

Other sections of the tape, identified by names such as `ACC` or `MAR`, for example, are activated in the same way, hence, the function `mark` (see Figure 1) uses a parameter to represent initial states of machines similar to the `pc`[17].

[16] A nore systematic description of delimiters in Section 4.1; see also Figure 1.

[17] The definitions of functions: `pc`, `acc`, `op`, `ir`, and `mar` are in Appendix 2. Formally, it is a collection of machines, which for convenience is represented by a single function with guarded expressions representing these machines.

Sometimes, processing depends on a final state in which the previously used machine stopped. The machine **decoding** is an example of such a machine. It ends in the state which becomes the initial state from which the machine **execute** starts. The transition function of the **decoding** is presented graphically in Figure 2.

This machine starts in the initial state **Eval** (in a centre of this picture).
It is assumed that the read/write head is placed at a right-most symbol of the sequence representing an operation code of an instruction and that three symbols define this code. We assume further that the first symbol is primed to indicate that this section of the tape is active.

```
decoding c = (tm opCode.rewind) (Config opCode t)
             where (Config opCode t) = tm Eval c
```

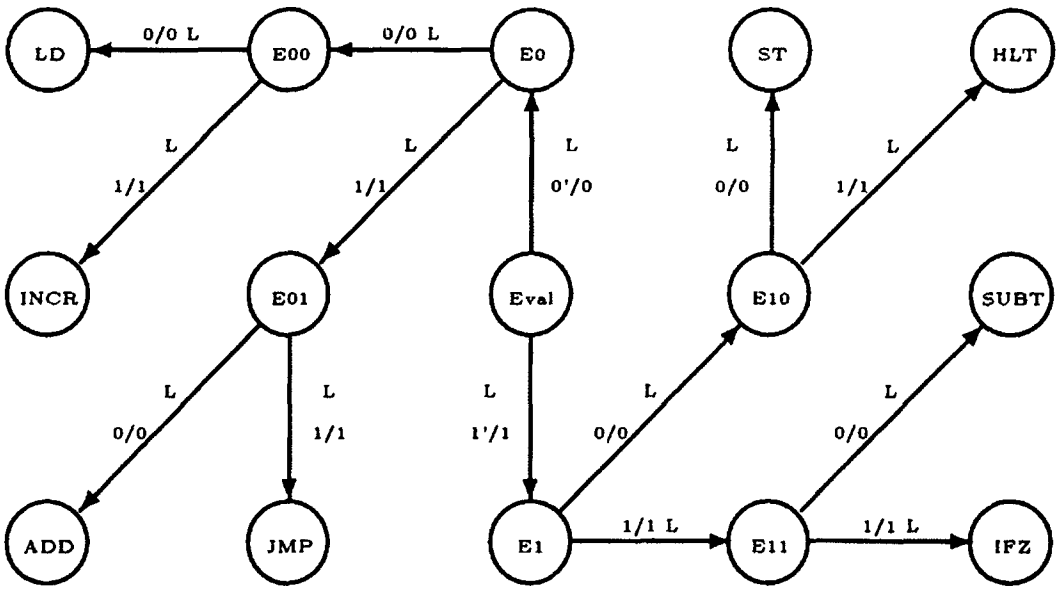

Figure 2. Eval of op-code.

The **decoding** is evaluated in three stages. The first one (i.e., **tm Eval**) determines the **opCode**, but also leaves the read/write head in a wrong position. Hence, the **rewind** is applied next, but this machine ends normally in the state **Done**, and not in the state **opCode**. That is why the third machine **tm opCode** is used to make the **opCode** visible again.

3. REPEATED COMPUTATIONS

An approach based on incremental improvements is well known in mathematics as converging sequences were used in definitions of new numbers as early as in the 17^{th} century [10,11]. A solution can sometimes be found in a predetermined number of steps, but often only some measure of progress can be defined. Then, providing the processing steps converge, one can interrupt iterations when the progress is sufficient.

This approach is presented graphically in Figure 3. It contains a block called COMPUTE, which represents an elementary step of processing, and two other blocks: ANY PROGRESS and BACK UP. They represent testing of the progress of processing. There are two large sections on the tape: *work space* and *back-up*. They are placed next to each other. The first of the two is used by an individual processing step. It contains all the data required by this step, as well as the results produced. The section *back-up* contains the same kind of information, but pertaining to a previous step. The task of the block ANY PROGRESS is principally to check whether the two sections are identical. If they are not the same, the block BACK-UP copies the work space onto the back-up area.

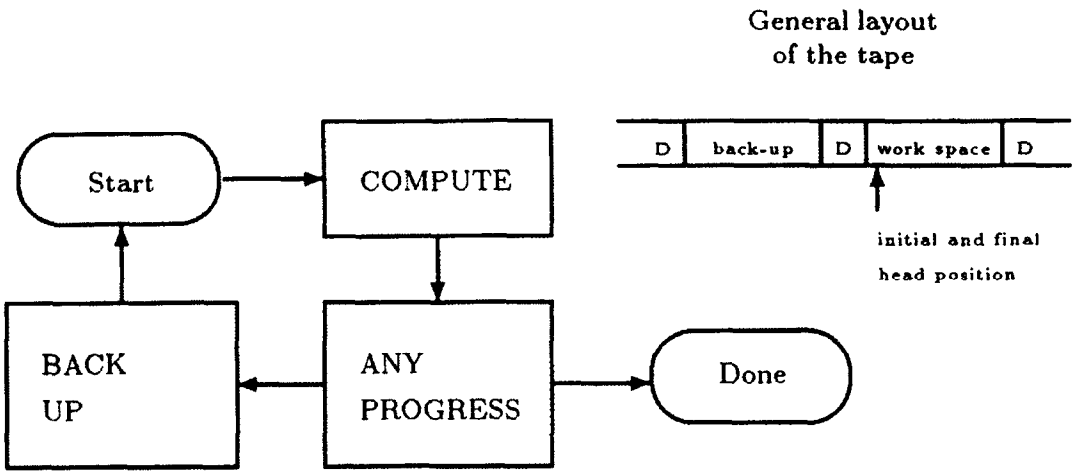

Figure 3. Scheme of iterative computing.

This fairly general scheme ensures that the machine `bigCycle step` performs the processing `step` iteratively and stops in obviously nonpromising situations. This modest exit condition does not guarantee that the `Done` state (or any other final state) is reached, for the COMPUTE block (represented by the machine `step`) may be ill-defined or looping may involve more than one processing step.

```
bigCycle :: (configuration -> configuration) ->
            configuration -> configuration
bigCycle step c = newC,                 if newS=Done
                = bigCycle step newC, otherwise
                  where newC = (anyProgress.step) c
                        (Config newS newT) = newC
anyProgress = tm AP
```

In the following example, the `sample1` represents the tape content which is used by the machine `bigCycle (rewind.pc)`. It is assumed that, before and after checking if `anyProgress` occurred, the read/write head observes the first symbol of the work space. Hence, the presence of the `rewind` after the `pc`. It is further assumed that the tape must have the same 'structure'[18] in both its parts, i.e., back-up and work space.

Finally, due to the definitions of the `hleft` and the `hright`, the area to the left of the read/write head should be interpreted in reverse order.

```
sample1 = ([D,PC,Zero,Zero,ACC,Zero,Zero,Zero,D],
           Zero,[One,One,ACC,Zero,One,PC,D])
Miranda bigCycle (rewind.pc) (Config undef sample1)
Config Done([D,PC,ONE,Zero,ACC,One,One,Zero,D],Zero,[One,One,ACC,Zero,ONE,PC,D])
```

Alternatively, one can use the function `shConfig` which shows the results in a format which resembles the tape better.

```
Miranda (shConfig.bigCycle (rewind.pc)) (Config undef sample1)
Done      ...D,Zero,One,One,ACC,Zero,ONE,PC,D,Zero,One,One,ACC,Zero,ONE,PC,D...
                                              ^^^^
```

An application of the function `bigCycle` not only turns the *pc* on, but also copies the work space, unlike a direct application of the step `(rewind.pc)`.

[18]The structure of the memory is defined by special symbols, such as `PC` or `ACC`. They must occur in the same order in both sections of the tape. Moreover, lengths of 'binary' sequences in between special symbols are fixed and are expected to be the same in both areas (see Section 4.1 for a complete description).

```
Miranda (rewind.pc) (Config undef sample1)
Config Done ([D,PC,Zero,Zero,ACC,Zero,Zero,Zero,D],Zero,
             [One,One,ACC,Zero,ONE,PC,D])
```

An equivalent and faster functional description of the iterative processing delegates the task of comparing and copying of the tape content to a lower level of string operations, which are serviced by a language translator:

```
cycle c = c,          if t = newT
        = cycle newC, otherwise
          where newC = compute c
                (Config s t) = c
                (Config newS newT) = newC
```

With either arrangement of repetitions, the next problem is to design the `compute` function which would resemble typical actions of a computer. This involves decisions concerning a structure of memory and a selection of elementary operations implemented directly by separate machines.

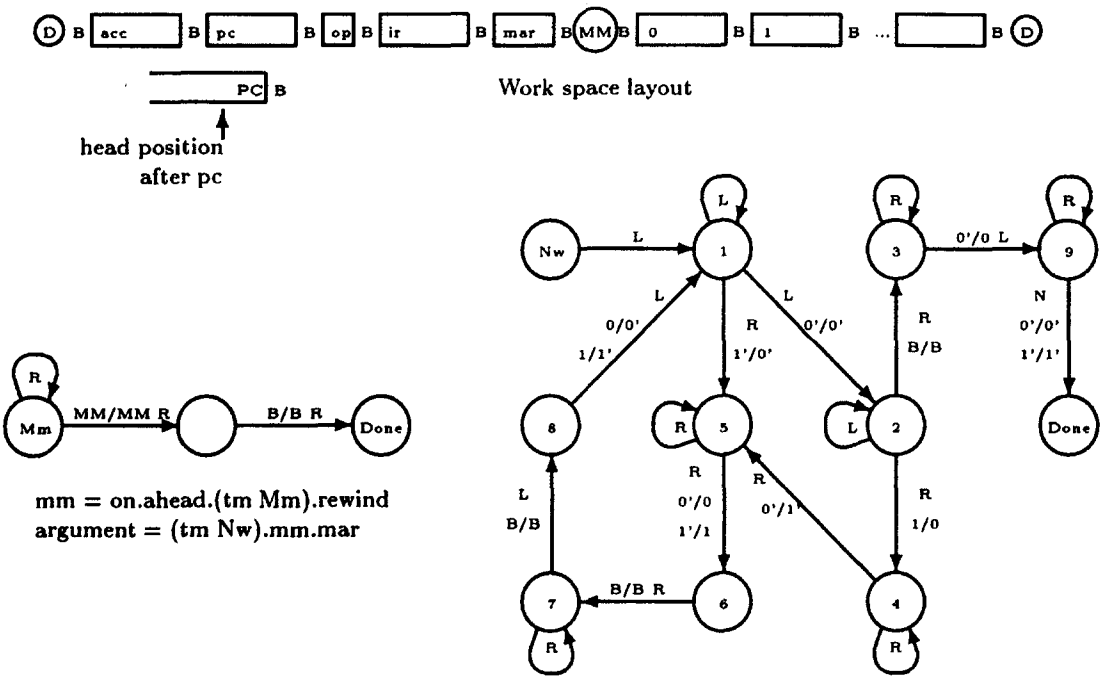

Figure 4. Setting a marker in memory.

4. THE STRUCTURE OF MEMORY

4.1. Names and Indexing

The tape represents memory in this model and it is assumed that it is *word* oriented and its structure is fixed. The tape is a homogeneous container of memory cells with strictly sequential access, but it becomes a set of words due to labelling of sections of the tape.

The model assumes a specific layout of the *work space*. It is shown in Figure 4. Words in the tape are separated by `Blanks` (denoted by `B`). Generally, registers are towards the left-end of the tape, while an addressable memory is to the right of the special 'empty' word containing only the marker `MM`.

Registers have individual labels, which are the rightmost symbols of respective words. The remaining part of the memory has a common name `MM` and its words are identified by addresses. These addresses are generated whenever words in memory are referred to. This difference in

characteristics influences the search algorithm, which becomes slower (compare the function pc from Section 2.3 and the function **argument** in Figure 4).

There are five registers in this computer.

The *instruction register* (ir) keeps the instruction which is actually executed; two other registers are associated with it:

- op which holds operation code of the instruction, and
- the *memory address register* (mar) for an address of an operand.

Their shorter length and location on both sides of the ir is essential for the way the *op-code* and the *argument* are extracted from the instruction (see functions **defOp** and **defArg** in Section 4.2).

The two remaining registers are necessary to perform other phases of a computer's operation: *fetch* the instruction and *execute* it. The *accumulator* (or acc) is designed to store temporarily the left argument of an operation and its result. The *program counter* (or pc) holds the address of the currently executed instruction.

The above registers and their functions constitute the main assumptions of a computer, which is known as a stored program, sequential, and one-address machine.

The model assumes further, that a word must be marked to take part in the computer's activity. The register is marked by the function **mark** (see Figure 1) and the memory cell is marked by the function **argument**, which leaves the same kind of marking. However, before the cell in memory is marked, its address must be placed in the memory address register as the function **argument** uses this register as a counter. After completion, operations routinely clear all markings in the words involved.

4.2. Transfers within Memory

Transfers from one place of the tape to another are performed between marked locations. The nature of the tape implies two possible transfers: the **left_transfer**, and the **right_transfer**.

The graph of the **right_transfer** is shown in Figure 5 and the similar graph of the **left_transfer** can be prepared by students as an exercise.

The first applications of transfers are in the **load** and the **store** functions, which are characteristic for the one-address machine. They represent transfers to and from a default register—the *accumulator*.

```
load  = rewind.left_transfer.acc
store = rewind.right_transfer.acc
```

However, since commands are stored in the same memory as other data items, a **fetch** function can be defined to place the instruction in the *instruction register*. To accomplish this, the content of the pc must be **right_transferred** to **mar** (which is to the right of the pc); next the corresponding word in the memory is marked by the **argument** and finally **left_transferred** to the ir.

```
fetch = rewind.left_transfer.ir.argument.
        rewind.right_transfer.pc.mar
```

The above are regular transfers as they involve words of standard length. However, when either the source-word or the destination-word is shorter, the following rewriting rules apply:

- rewriting stops, when there is
  - no more data items to transfer or, there is
  - no room for data, whichever comes first;
- rewriting begins from
  - the leftmost data item for the **left_transfer**
  - the rightmost data item for the **right_transfer**.

These rules are useful when the content of ir is to be parted into the *op-code* and the *argument*. The extraction of the op-code is done by the **defOp** function and the argument of the instruction

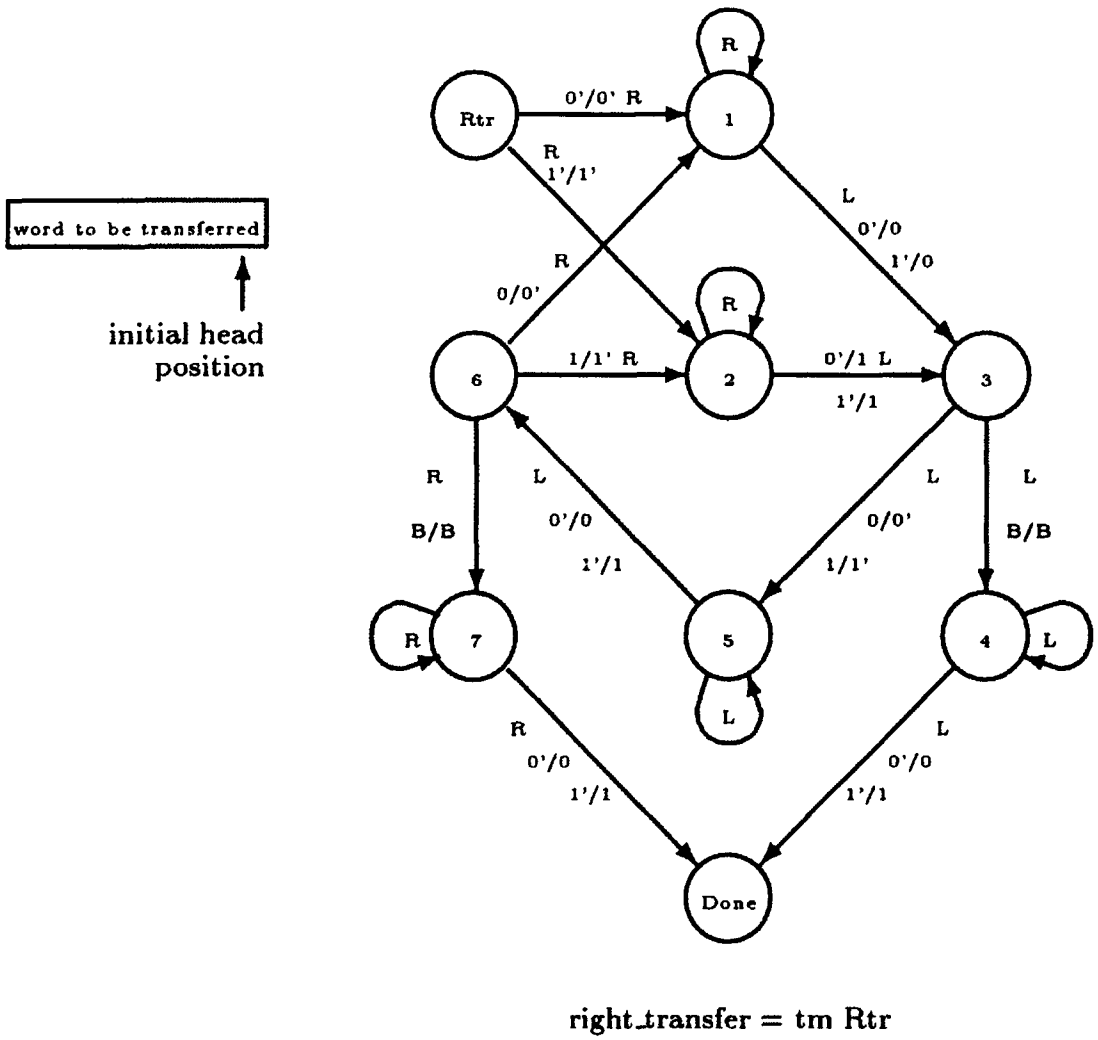

Figure 5. Word transfer to the right.

is defined by the `defArg` function. These functions exploit the fact that the two destination registers are of appropriate length.

```
defOp  = op.rewind.left_transfer.op.ir
defArg = rewind.right_transfer.ir.mar
```

5. ELEMENTARY OPERATIONS

Addition is usually implemented by a separate component. Designing a Turing machine `Add`, which adds two marked sequences of equal length, is an independent and relatively easy problem. We assume, it replaces the left sequence by a result. Now, if the *accumulator* is marked, it may be used as a default operand/destination in a standard one-address instruction which implements addition.

```
add    = tm Add
adder = add.acc
```

The algorithm of arithmetic negation, described by the function `ar_negate`, is based on two's complement notation of numbers.

```
ar_negate = increment.on.lg_negate
lg_negate = ahead.(tm Lneg)
increment = ahead.(tm Incr)
```

The function **decrement** could be defined by a separate graph, but the definition below is more convincing, although perhaps more complex than one would expect.

```
decrement = ar_negate.on.increment.on.ar_negate
```

To perform subtraction, the second operand must be arithmetically negated by an **ar_negate**, before it is sent to the **adder**[19].

```
subtr = ar_negate.acc.adder.ar_negate.acc
```

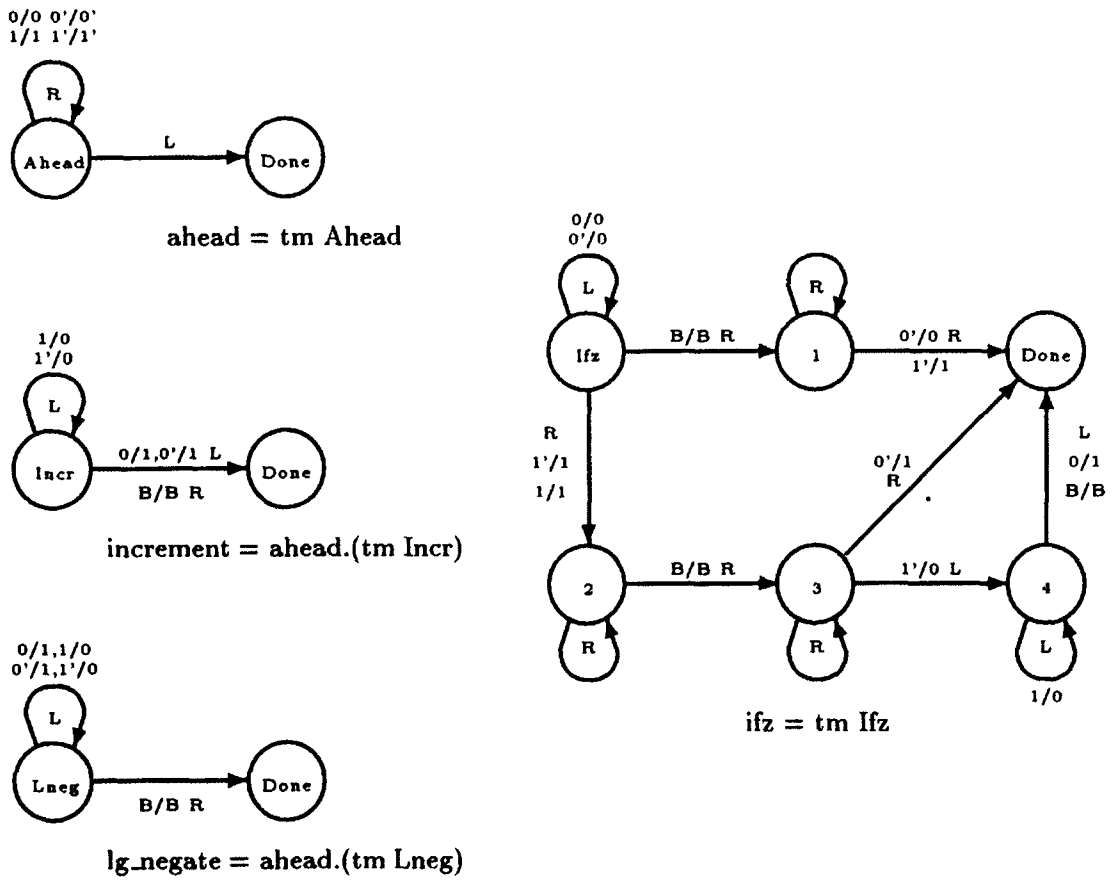

Figure 6. Conditional skip and other operations.

The conditional instruction is implemented by the `Ifz` graph, which is surprisingly simple (see Figure 6). The *accumulator* and the *pc* must be marked to enable a test for zero, possibly followed by the increment operation on the *pc*.

```
ifz = tm Ifz
```

6. THE INSTRUCTION—STEPS OF EXECUTION

The block COMPUTE (see Section 3) is represented by the function **compute**. It puts elementary actions of a computer into sequence which is normally performed in a *fetch-execute cycle* for each instruction.

```
compute = rewind.execute.decoding.defOp.defArg.update_pc.fetch
```

[19] A direct translation of the above: **subtr = adder.on.ar_negate** would leave not only the correct result in **acc**, but also the second argument in a negated form.

6.1. Command Selection

The model assumes that the instruction is addressed by a *program counter* which is routinely incremented by one. It implies contiguous placement of instructions in memory and that every instruction is stored in one cell.

```
update_pc = rewind.increment.pc
```

6.2. Decoding

Decoding means that the sequence from `ir` is divided by the functions **defOp** and **defArg** (Section 4.2) into the *op-code* and the *argument*. The latter can be used by the function **argument** (Figure 4) after it is transferred to the `mar`, but the *op-code* must be deciphered by the function `decoding` (Section 2.3).

6.3. Execution of Commands

Once the action prescribed by the *op-code* is decided, the command is executed by the function `execute`. It defines the meaning of all commands available in the model.

```
execute c = (       load.argument) c, if cd=LD
          = (      store.argument) c, if cd=ST
          = (      adder.argument) c, if cd=ADD
          = (      subtr.argument) c, if cd=SUBT
          = ( increment.argument) c, if cd=INCR
          = (        decrement.pc) c, if cd=HLT
          = (left_transfer.pc.mar) c, if cd=JMP
          = (          ifz.acc.pc) c, if cd=JZ
            where (Config cd t) = c
```

It is assumed that the first argument of operations: `ADD`, and `SUBT`, is available in the accumulator, the second argument is defined by the function **argument** (Figure 4), and that the result is left in the accumulator.

The command `INCR` increments the value defined by the **argument** while the remaining three control commands redefine the content of the *pc*.

The `HLT` command reverses the effect of the `update_pc`, which is performed routinely in every execution cycle. The unconditional jump `JMP` explicitly specifies the value in the `pc`, i.e., the address of the next instruction. The `JZ` tests the value in the accumulator and the content of the *pc* is additionally incremented, if a nonzero value is found.

7. EXPERIMENTATION

There is a scope for students' experimentation with the model. One may start by simply evaluating given functions. The next, relatively easy, exercise is to extend the definition of the `tr` function to include cases of the `Ltr` and the `Ifz`[20]. The case of `Add` is more complex, but it is still in the category of moderate difficulty.

Later, after gaining some confidence, students can add new instructions, such as shift, rotate, compare, etc. A bit more ambitious project would involve changes in the computer's architecture such as inclusion of new registers, or even two-address instructions.

8. CONCLUSIONS

Recursive functions serve well as statements defining the process of computing. A concise style of the functional approach together with an effective use of abstraction enabled to create a simple and clear model of a computer.

[20] Additional lines, which define these cases, can be contained in a separate file, which may be inserted into the previously used version of the `tr`. It is convenient when the students' solutions are to be verified.

Parallelism in processing has been mentioned a number of times as an issue recurring at various levels in computing. It begins with an idea of 'cutting' the tape into pieces and referring to them as indivisible entities—*words*.

Although the model generally assumes that one instruction is executed at a time, the complexity of instructions is unspecified and allows local parallelism. It means that by introducing names for registers, for example, we become indifferent as to how individual bits are transferred (sequentially or in parallel), or in the case of the adder, we simply assume that once it has been defined, it leaves the correct result somewhere. Thus, introducing names opens the way for parallel actions, although such actions can be translated into strictly sequential and much slower sequences of operations.

APPENDIX 1

There are three files available: body.m, types.m, and transitions.m All definitions introduced in this text are there. The `tr` has omitted cases of the `Ltr`, `Ifz`, and `Add`, which are suggested as exercises.

This appendix begins with the sample of the memory layout[21] (`sample2`) and is followed by its modified content after the execution of the command `LD 8`, then after the `ADD 9` and the `HLT`. This output has been produced by the complete model i.e., with the missing cases defined.

The output formatting functions used to produce the following results are included at the end of this appendix.

```
>  sample2
>  =([Blank],D,[Blank,
>    Zero,Zero,Zero,Zero,Zero,One ,Zero,ACC,Blank,
>                   Zero,Zero,Zero,Zero,PC ,Blank,
>                        Zero,Zero,Zero,OP ,Blank,
>    Zero,Zero,Zero,Zero,Zero,Zero,Zero,IR ,Blank,
>                   Zero,Zero,One ,One ,MAR,Blank,
>                                      MM ,Blank,
>    Zero,Zero,Zero, One ,Zero,Zero,Zero,Blank,
>    Zero,One, Zero, One, Zero,Zero,One, Blank,
>    One ,Zero,Zero, One ,Zero,Zero,Zero,Blank,
>    One ,Zero,One , One ,Zero,Zero,Zero,Blank,
>    Zero,One ,One , One ,Zero,Zero,Zero,Blank,
>    One ,Zero,Zero, One ,Zero,Zero,Zero,Blank,
>    One ,Zero,One , Zero,Zero,Zero,Zero,Blank,
>    One ,One ,Zero, Zero,Zero,Zero,Zero,Blank,
>    One ,One ,One , Zero,Zero,Zero,Zero,Blank,
>    Zero,Zero,Zero, One, Zero,Zero,One, Blank,
>    Zero,One ,Zero, Zero,Zero,Zero,Zero,Blank])
Miranda (sh_tape.compute) (Config undef sample2)
([D,Blank],Blank,
[One, One, One, Zero,Zero,Zero,Zero,ACC,Blank,
 Zero,Zero,Zero,One,                  PC,Blank,
 Zero,Zero,Zero,                      OP,Blank,
 Zero,Zero,Zero,One, Zero,Zero,Zero, IR,Blank,
 Zero,Zero,Zero,Zero,                MAR,Blank,
                                      MM,Blank,
 Zero,Zero,Zero,One, Zero,Zero,Zero,Blank,
 Zero,One, Zero,One, Zero,Zero,One, Blank,
 One, Zero,Zero,One, Zero,Zero,Zero,Blank,
 One, Zero,One, One, Zero,Zero,Zero,Blank,
```

[21] The special symbol `D` occurs only once as only work area is used in these experiments.

```
 Zero,One, One, One, Zero,Zero,Zero,Blank,
 One, Zero,Zero,One, Zero,Zero,Zero,Blank,
 One, Zero,One, Zero,Zero,Zero,Zero,Blank,
 One, One, Zero,Zero,Zero,Zero,Zero,Blank,
 One, One, One, Zero,Zero,Zero,Zero,Blank,
 Zero,Zero,Zero,One, Zero,Zero,One, Blank,
 Zero,One, Zero,Zero,Zero,Zero,Zero,Blank])
Miranda (sh_tape.cycle) (Config undef sample2)
([D,Blank],Blank,
[One, One, One, One, Zero,Zero,One, ACC,Blank,
 Zero,Zero,One, One,                 PC,Blank,
 One, Zero,One,                      OP,Blank,
 One, Zero,One, One, Zero,Zero,Zero, IR,Blank,
 One, Zero,Zero,Zero,               MAR,Blank,
                                     MM,Blank,
 Zero,Zero,Zero,One, Zero,Zero,Zero,Blank,
 Zero,One, Zero,One, Zero,Zero,One, Blank,
 One, Zero,Zero,One, Zero,Zero,Zero,Blank,
 One, Zero,One, One, Zero,Zero,Zero,Blank,
 Zero,One, One, One, Zero,Zero,Zero,Blank,
 One, Zero,Zero,One, Zero,Zero,Zero,Blank,
 One, Zero,One, Zero,Zero,Zero,Zero,Blank,
 One, One, Zero,Zero,Zero,Zero,Zero,Blank,
 One, One, One, Zero,Zero,Zero,One, Blank,
 Zero,Zero,Zero,One, Zero,Zero,One, Blank,
 Zero,One, Zero,Zero,Zero,Zero,Zero,Blank])
```

OUTPUT FORMATTING FUNCTIONS

```
>  sh_tape :: configuration ->  [char]
>  sh_tape (Config s (left, w, right))
>   = "("++xsht left++","++show w++",\n"++
>     xsht right++")\n"
>  xsht ts = "["++(concat.(map sht)) (init ts)++
>            show (last ts) ++ "]",        #ts >  1
>          = show ts,                     otherwise
>  sht s
>   = show s ++ ",\n ", s=Blank
>   = show s ++ ",",    s=Zero s=ZERO
>                       s=ACC s=D
>   = show s ++ ", ",   s=One  s=ONE
>   = (rjustify  3 (show s)) ++ ",", s=IR
>   = (rjustify 18 (show s)) ++ ",", s=PC s=MAR
>   = (rjustify 23 (show s)) ++ ",", s=OP
>   = (rjustify 38 (show s)) ++ ",", s=MM
>   shConfig :: configuration ->  string
>   shConfig (Config s t)
>    = ljustify ljust (show s)++shTape ljust t++"\n"
>      where ljust = 10
>   shTape :: num ->  tape ->  string
>   shTape lj (left, w, right)
>    = lSide++window++"...\n"++spaces (#lSide+lj)++
>      rep (#window) '?',                  if  rL=0
>    = lSide++window++","++show (last right)++
>      "...\n"++spaces (#lSide+lj)++
>      rep (#window) '?',                  if rL=1
```

```
>     = lSide++window++","++shT (init right)++
>       show (last right)++"...\n"++
>       spaces (#lSide+lj)++rep (#window) '?',
>                                          otherwise
>       where lSide  = "..."++shT (reverse left)
>             rL = #right
>             window = show w
>   shT :: [symbol] ->  string
>   shT = concat.map ((++",").show)
```

APPENDIX 2

The following type declarations are used by the model. They are available in the file types.m, which constitutes an integral part of this paper.

```
> string == [char]
> symbol ::= Zero | One | Blank |
>            ZERO | ONE | ZerO | OnE |
>            ZERo | ONe |
>            D | ACC | PC | MAR | IR | OP | MM
> state ::= Ahead |
>           Done |  LD | ST | ADD | SUBT | INCR |
>                   HLT | JMP | JZ |
>           Eval |
>             E0 | E1 | E00 | E01 | E10 | E11 |
>           Incr | Lneg |
>           Ifz | Ifz_1 | Ifz_2 | Ifz_3 | Ifz_4 |
>           Add | A0 | A1 | A00 | A10 |
>                 D00 | D01 | D10 | D11 |
>                 S000 | S001 | S010 | S011 |
>                 S100 | S101 | S110 | S111 |
>                 Carry0 | Carry1 | R00 | R01 |
>                 R10 | R11 |
>                 Add_cleanup | Add_back |
>           Dlim | On | Acc | Pc | Mar | Ir | Op |
>           Mm | Mm0 |
>           Rtr | Rtr_1 | Rtr_2 | Rtr_3 | Rtr_4 |
>                 Rtr_5 | Rtr_6 | Rtr_7 |
>           Ltr | Ltr_1 | Ltr_2 | Ltr_3 | Ltr_4 |
>                 Ltr_5 | Ltr_6 | Ltr_7 | Ltr_8 |
>                 Ltr_9 | Ltr_10 | Ltr_11 |
>           Nw | Nw_1 | Nw_2 | Nw_3 | Nw_4 |
>                Nw_5 | Nw_6 | Nw_7 | Nw_8 | Nw_9 |
>           AP | AP1  | AP2  | AP3  | AP4  | AP5 |
>                AP6 | AP7 | AP8 | AP9 |
>                AP10 | AP11 | AP12 | AP13 |
>                AP_2 | AP_3 | AP_4 | AP_5 | AP_8 |
>                AP_9 | AP_10 | AP_11 |
>           BU | BU1  | BU2  | BU3  | BU4  | BU5  |
>                BU6  | BU7  |
>                BU_2 | BU_3 | BU_4 | BU_5 |
>           ShL| ShL1 | SHl  | SHl1 | SHl2 |
>           SHL | SHL0 | SHL1 | SHL2
>  move ::= Left | Right | None
>  tape       == ( lside_tape, window, rside_tape )
>  lside_tape == [symbol]
```

```
> window     ==  symbol
> rside_tape == [symbol]
> configuration ::= Config state tape
```

The following definitions constitute the model. They are available in the file body.m. This file is also an integral part of this paper.

```
> final :: [state]
> final =  [Done,LD,ST,ADD,SUBT,INCR,HLT,JMP,JZ]
> hleft, hright :: tape -> tape
> hleft  ([]   , c, rs) = ([], Blank, c:rs)
> hleft  (c':ls, c, rs) = (ls, c'   , c:rs)
> hright (ls, c, []   ) = (c:ls, Blank, [])
> hright (ls, c, c':rs) = (c:ls, c'   , rs)
> driver :: configuration -> configuration
> driver (Config s t)
>   = (Config s t),       if member final s
>   = driver (Config newS newT), otherwise
>     where (l, w, r) = t
>           (newS, neW, dir) = tr s w
>           newT' = (l, neW, r)
>           newT  = hleft  newT', if dir = Left
>                 = hright newT', if dir = Right
>                 =        newT', if dir = None
> tm :: state -> configuration -> configuration
> tm newS (Config s t) = driver (Config newS t)
> decoding c
>   = (tm opCode.rewind) (Config opCode t)
>     where (Config opCode t) = tm Eval c
> bigCycle :: (configuration -> configuration) ->
>             configuration -> configuration
> bigCycle step c = newC,             if newS=Done
>                 = bigCycle step newC, otherwise
>                   where
>                   newC = (anyProgress.step) c
>                   (Config newS newT) = newC
> anyProgress = tm AP
> cycle c = c,          if t = newT
>         = cycle newC, otherwise
>           where newC = compute c
>                 (Config s t) = c
>                 (Config newS newT) = newC
> load  = rewind.left_transfer.acc
> store = rewind.right_transfer.acc
> fetch = rewind.left_transfer.ir.argument.
>         rewind.right_transfer.pc.mar
> defOp  = op.rewind.left_transfer.op.ir
> defArg = rewind.right_transfer.ir.mar
> add   = tm Add
> adder = add.acc
> ar_negate = increment.on.lg_negate
> lg_negate = ahead.(tm Lneg)
> increment = ahead.(tm Incr)
> decrement = ar_negate.on.increment.on.ar_negate
> subtr = ar_negate.acc.adder.ar_negate.acc
> ifz = tm Ifz
```

```
>  compute = rewind.execute.decoding.defOp.defArg.
>            update_pc.fetch
>  update_pc = rewind.increment.pc
>  execute c = (       load.argument) c, if cd=LD
>            = (      store.argument) c, if cd=ST
>            = (      adder.argument) c, if cd=ADD
>            = (      subtr.argument) c, if cd=SUBT
>            = (  increment.argument) c, if cd=INCR
>            = (        decrement.pc) c, if cd=HLT
>            = (left_transfer.pc.mar) c, if cd=JMP
>            = (          ifz.acc.pc) c, if cd=JZ
>              where (Config cd t) = c
```

SECTIONS IN MEMORY AND BASIC OPERATIONS

```
> [rewind, ahead, on] = map tm [Dlim, Ahead, On]
> mark :: state -> configuration -> configuration
> mark s = on.tm s.rewind
> [acc, pc, op, ir, mar]
>       = map mark [Acc, Pc, Op, Ir, Mar]
> mm = on.ahead.tm Mm.rewind
> argument = tm Nw.mm.mar
> left_transfer  = tm Ltr
> right_transfer = tm Rtr
```

REFERENCES

1. J.A. Piotrowski, Sequential and data flow models of processing, *Computers Math. Applic.* **27** (1), 81–96 (1994).
2. J.A. Piotrowski, A functional model of a simplified sequential machine, *Information Processing Letters* **35** (3), 161–166 (1990).
3. J.A. Piotrowski, Playing with abstract machines, *Computers & Education* **17** (3), 181–193 (1991).
4. D. Turner, An overview of Miranda, *ACM SIGPLAN Notices* **21** (12), 158–166 (1986).
5. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.* **2** (42), 230–265 (1936).
6. E.L. Post, Finite combinatory processes—Formulation I, *J. Symbolic Logic* **1**, 103–105 (1936).