Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 71–91
© 2014 NCC Publisher

Deductive formal verification of search programs in arrays of arbitrary size for abstract register machines

D.A. Chkliaev, V.A. Nepomniaschy

Abstract. The random-access machine invented by Aho, Hopcroft, and Ullman is one of the several known versions of abstract register machines, which are an important computation model. Using a formal framework developed for this architecture in our previous work, we consider a challenging example – a search program that computes the maximum of an integer array of an arbitrary size. Here we present its specification in PVS and complete deductive verification, which turned out to be rather difficult. We have also proved its best-case and worst-case complexity measures as a function on the size of the array. We hope that our use of deductive verification for this example is both novel and appropriate, because the fully automated techniques, such as model-checking, are not perfectly suitable either for the verification of infinite-state programs or for the analysis of their complexity.

Keywords: abstract register machines, random-access machines, search programs, formal specification, interactive theorem proving, verification system PVS.

1. Introduction

The formalism of ARMs (abstract register machines) [14, 3] has been used during several decades for the formal, mathematical study of computational algorithms and programs. However, we are not aware of any works that systematically study the formal verification of computational programs for ARMs.

The random-access machine suggested by Aho, Hopcroft, and Ullman (RAM-AHU) is one of the several known versions of abstract register machines. In our previous work [6], we presented a comprehensive framework that allows us to verify rigorously computational programs for RAM-AHU. We represent the data structures of RAM-AHU in the language of the verification system PVS [15] and define the effect of its commands on these data structures. After that, the executions of RAM-AHU are formalized as finite or infinite traces of a transition system generated by the effect predicate of its commands. The correctness properties of programs for RAM-AHU are defined by logical formulas on traces of this transition system.

As an application of our formal model, we have presented the verification of a simple search program for RAM-AHU that computes the larger of two integers. We have proved not only the functional correctness of the program,

but also its best-case and worst-case execution time. The simplicity of the program is indicated by the fact that it contained no loops.

The programming of more complex algorithms for RAM-AHU is a challenging task. Unlike the CPU's used in practice, this architecture has only one input tape and one set of registers; also, there is no additional memory that can hold the (preliminary) instructions or computational results. Moreover, the set of RAM-AHU commands allows only programs with very primitive control flow: there are no while- or for-statements, only unconditional jumps and conditional jumps that compare the first register to zero. This makes the programming of loops a rather tedious task.

In this paper, our goal is to develop a program computing the maximum of an integer array of arbitrary size. Another goal is the formal verification of our program, which is complicated by the arbitrary length of the input array. First, we prove that the program terminates for any input data, which is done by showing that the loop counter always decreases to zero for all possible paths. After that, the functional correctness of the program and its exact complexity bounds are proved within a single formal framework. We establish that when the program terminates, there is exactly one number written on its output tape equal to the maximum of all elements in the input array. The best-case and worst-case complexity measures are expressed as functions of the size of the array.

The rest of the paper is organized as follows. Section 2 describes RAM-AHU on the basis of [1]. In Section 3, we present our formalization of the executions of abstract register machines. In Section 4, the RAM-AHU data structures and its commands are specified in PVS. Section 5 presents a specification of a search program operating on arrays of arbitrary size, first in the RAM-AHU language and then in PVS. Specification of the functional correctness for our program is given in Section 6, as well as the main theorem expressing its complexity measures. Section 7 contains the verification of the main theorem with PVS. Finally, Section 8 gives some concluding remarks on related works and possible future work.

2. RAM-AHU machine

The random-access machine invented by Aho, Hopcroft, and Ullman [1], which we call here RAM-AHU, is a computing device with one adder in which the program cannot change itself (the so-called Harvard architecture). It consists of three parts: the input tape, the main (computational) part, and the output tape.

The *input tape* is a sequence of *cells* of an unlimited length. Each cell contains a *symbol*; it is only possible to read symbols from the input tape but not to write them. At any moment, the *reading head* of the tape points to some cell. After reading a symbol from that cell, the head moves one cell

right.

The *output tape* is also an unlimited sequence of cells, with each cell containing a symbol. It is only possible to write symbols to the output tape but not to read them. At any moment, the *writing head* of the tape points to some cell. After writing a symbol to that cell, the head moves one cell right. It is not possible to change symbols that have already been written to the output tape. For this version of the machine, all symbols that can appear on the input or output tape are integers.

The computational part of RAM-AHU consists of a program, a program counter, and memory. *The program* for RAM-AHU is a finite sequence of *commands*; each command can have a *label*. It is assumed that the program is not stored in the memory, so it cannot change itself during its execution (which corresponds to the so-called Harvard architecture). There are commands for arithmetical operations, conditional and unconditional jumps, input/output operations and some others.

At any moment of time during the program execution, *the program counter* points to some of its commands that should be executed at the next step of the computation. After the command with some index k is performed, the counter automatically moves to the command with the index $k+ 1$ (i.e. the next command). The only exception is made for conditional and unconditional jumps, as well as the command HALT which stops the computation. If the counter no longer points to any command (i.e. exceeds the length of the program), this means that there are no more commands to be executed, so the computation is over.

The memory of RAM-AHU is a sequence of *registers* $r_0, r_1, \ldots r_i, \ldots$; each register can store an arbitrary integer. It is assumed that there is no upper limit to the number of registers that can be used. This idealization is reasonable when the size of the task is small enough to fit in the main memory of the machine. The first register r_0,called *the adder*, participates in all arithmetical operations (it can also store an arbitrary integer).

The initial state of RAM-AHU is determined by the chosen program and its input data. In any initial state, there are some symbols on its input tape (i.e. the input data), all registers are empty, the output tape is also empty, and the program counter points to the first command of the program. After the execution of each command, the program counter changes as described above until it eventually exceeds the length of the program and the computation stops. It is also possible that this event never happens (i.e. there is always some command waiting to be executed), and this leads to a non-terminating computation.

Each command of RAM-AHU consists of two parts – its *operation code* and its *address*. The command's address is either *an operand* or a label of some command in the program; in some cases it can also be empty. An operand a can be of one of the three types:

1. The expression $= i$ means the integer i itself and is called a *literal*.
2. The expression i means the content of the register i(i cannot be negative).
3. The expression $*i$ means the use of indirect addressing, i.e. the value of this operand is the content of the register j, where j is the integer located in the register i. If $j < 0$, the program should stop.

If some command has an operand a, we can define *the value v(a)* of this operand. The definition of the function v uses another function c: for each natural number i, $c(i)$ is the content of the register i. Using the informal definition of the expressions $= i$, i and $*i$ given above, we define the value of an arbitrary operand a as follows:

- v(=i) = i,
- v(i) = c(i),
- v(*i) = c(c(i)).

There are 12 types of commands in the programs for RAM-AHU: LOAD, STORE, ADD, SUB, MULT, DIV, READ, WRITE, JUMP, JGTZ, JZERO and HALT. For the first eight commands (their meaning is clear from their names) the address is an operand. For the commands JUMP, JGTZ and JZERO, the address is a label, and for the command HALT, the address is empty. JUMP is an unconditional jump instruction, whereas JGTZ ("jump if greater than zero") and JZERO ("jump if equal to zero") are conditional jump instructions. The HALT command terminates the program execution.

The following list defines the effect of each command. Here the sign $\leftarrow$ denotes an assignment, and the function *floor(x)* gives the greatest integer that is less than or equal to x. Undefined commands and commands with an illegal value of the address are equivalent to the command HALT.

1. LOAD a. Effect: c(0) $\leftarrow$ v(a)
2. STORE i. Effect: c(i) $\leftarrow$ c(0)
   STORE $*i$. Effect: c(c(i)) $\leftarrow$ c(0)
3. ADD a. Effect: c(0) $\leftarrow$ c(0) + v(a)
4. SUB a. Effect: c(0) $\leftarrow$ c(0) - v(a)
5. MULT a. Effect: c(0) $\leftarrow$ c(0) * v(a)
6. DIV a. Effect: c(0) $\leftarrow$ floor(c(0) / v(a))
7. READ i. Effect: c(i) $\leftarrow$ the next input symbol.
   READ $*i$. Effect: c(c(i)) $\leftarrow$ the next input symbol. In both cases, the head of the input tape moves one cell right.

8. WRITE *a*. Effect: v(a) is printed in the cell of the output tape which is currently observed by its head. After that, the head moves one cell right.
9. JUMP *b*. Effect: the program counter moves to the command with the label b.
10. JGTZ *b*. Effect: If c(0) >0, the program counter moves to the command with the label b or otherwise to the next command.
11. JZERO *b*. Effect: If c(0) = 0, the program counter moves to the command with the label b or otherwise to the next command.
12. HALT. Effect: the execution of the program stops.

3. A formalization of abstract register machines

Our methods of specification are not specific to RAM-AHU but can be used in principle to model the behavior of any abstract register machine. They have significant similarities to the methods we previously used to specify and verify distributed protocols in [4] and [5]. In our approach, the behavior of an abstract register machine is defined by the notion of a *state*, representing a snapshot of the state-of-affairs during the execution of its program, and a set of *commands*. The state includes all information present in the machine at any time: its program, the value of its registers and the program counter, the input and output tapes. Each command changes the values of some variables in the state when it is executed; commands can have an arbitrary number of parameters. Commands are formally specified by an *effect predicate* which relates the states before and after the command execution. The process of computation on an abstract register machine begins in an *initial state* which includes a particular program and a particular input tape.

The computation on an abstract register machine (for an initial state) terminates if and only if it eventually reaches a *final* state, i.e. a state where it is no longer possible to execute any command. For RAM-AHU, it is easy to see that a state is final if and only if the value of the program counter exceeds the length of the program, so the counter no longer points to any command. The definitions of the final state and the effect predicate are closely related: if a state *s1* is final and *Effect* is the effect predicate for our machine, then for any other state *s2* we should have *Effect(s1, s2) = false*. On the other hand, if a state *s1* is not final, then the effect predicate should transform it into another state *s2*. RAM-AHU is fully deterministic, so for any non-final state *s1* there exists exactly one state *s2* such that *Effect(s1, s2) = true*.

The complete execution of an abstract register machine (for some initial state) is defined by the notion of a *complete run*. A complete run R is

either an infinite or a finite sequence of states which satisfies the following conditions:

- If R is infinite, then it is a sequence of the form $s_0 s_1 \ldots s_i s_{i+1} \ldots$, where s_i $(i \geq 0)$ are states, s_0 is the initial state of the machine, and each pair of states (s_i, s_{i+1}) is related by the effect predicate.
- If R is finite, then it is a sequence of the form $s_0 s_1 \ldots s_i s_{i+1} \ldots s_n$, where s_i $(0 \leq i \leq n)$ are states, s_0 is the initial state of the machine, each pair of states (s_i, s_{i+1}) is related by the effect predicate, and the state s_n is final.

In our PVS specification of the states of RAM-AHU, the program is represented by the variable *program*, and the program counter by the variable *pCounter*. For an arbitrary state *st*, this allows us to define formally the notion of a final state by the following predicate *isFinal* (note that in PVS the elements of a sequence are enumerated starting with 0, not with 1):

isFinal(st) = (pCounter(st) ≥ length(program(st)))

In the PVS specification, the definition of complete runs is implemented by giving the initial state *Ini* (for a particular program and a particular input tape) and the effect predicate *Effect*, i.e. the Boolean predicate on pairs (s_i, s_{i+1}). We define the abstract datatype *Runs*, which includes both the infinite and finite sequences of states. Suppose R is a variable of the type *Runs*. If R is infinite, then it is a complete run if the following two properties are met:

1) *R(0) = Ini*;

2) for each natural index i, we have *Effect(R(i), R(i + 1)) = true*.

If R is finite and is of length *Len*, then R is a complete run if *Len >0* and the following three properties are satisfied (where the function *last* gives the last element of a sequence):

1) *R(0) = Ini*;

2) for each natural index *i*such that *i <Len - 1*, we have *Effect(R(i), R(i + 1)) = true*;

3) *isFinal(last(R)) = true*.

4. Implementation of RAM-AHU machine in PVS

4.1. Data structures of the machine in PVS

The PVS system [15], created at the Stanford Research Institute, is widely used for formal specification and verification of complex computer protocols and systems, especially in the area of fault-tolerant computing. It consists of a specification language, a large number of predefined theories and an

interactive prover, as well as documentation, tutorials and examples illustrating the use of PVS in several domains. In our previous work [6], we already used PVS to verify a simple search program.

To model RAM-AHU in PVS, we need to define the structure of its states. The state should include the program of the machine, the value of its registers and of the program counter, the input and output tapes. Since any program is a sequence of commands, we need to specify the structure of the machine commands.

In the informal definition of a program, only some of its commands have labels, and these labels are represented by words in a natural language. In PVS, it is much more convenient to have a label for every command, and to represent labels by natural numbers. A label equal to 0 is interpreted as absence of a label, and "real" labels are modeled by positive natural numbers. For this reason, any command is represented by a record with two fields: its *label* and its *body*. The body of a command belongs to the abstract datatype *CommandBody*, which is rather complex and is presented in the next section. Assuming that the type *CommandBody* is already defined, we can define the type *Commands* as follows:

```
Commands: TYPE = [# label : nat,
                    body : CommandBody #]
```

The complete data structure for the states of RAM-AHU is given by the PVS type *RAMstates*, which is defined as follows:

```
RAMstates : TYPE =
   [# program : finite\textunderscore sequence[Commands],
      pCounter : nat,
      registers : sequence[int],
      inputTape : sequence[int],
      inputHead : nat,
      outputTape : sequence[int],
      outputHead : nat #]
```

The meaning of the fields in the type *RAMstates* is rather obvious: the program is represented by a finite sequence of commands, the field *pCounter* models the program counter, the field *registers* represents the infinite sequence of registers, where each register can hold an integer. The input tape and the output tape are also modeled as infinite sequences of integers. The field *inputHead* points to the cell of the input tape that should be read during the next read command, and the field *outputHead* points to the cell of the output tape that is due to be written during the next write command.

Suppose that we have a program *SomeProg* (i.e. a finite sequence of the type *Commands*) and an input tape *SomeInputTape* for it (i.e. a sequence

of integers of unlimited length). The initial state for *SomeProg* and *SomeInputTape* is defined in a rather obvious way: they are included in the state, an empty sequence *EmptyIntSeq* (i.e. a sequence consisting of only zeros) is assigned to the fields *registers* and *outputTape*, and 0 is assigned to the program counter and the variables *inputHead* and *outputHead*. So the initial state for *SomeProg* and *SomeInputTape* is represented by the following constant *SomeIniState* of the type *RAMstates*:

```
SomeIniState : RAMstates =
   (# program :$=$ SomeProg,
      pCounter := 0,
      registers := EmptyIntSeq,
      inputTape := SomeInputTape
      inputHead := 0,
      outputTape := EmptyIntSeq,
      outputHead := 0 #)
```

4.2. Commands of the machine in PVS

It was already said in the previous subsection that any command of RAM-AHU is represented by a record with two fields: its label and its body. The body belongs to the abstract datatype *CommandBody* shown below.

```
CommandBody [ IntOpType : TYPE] : DATATYPE
BEGIN

load(typeop : IntOpType, intop : int) : load?
store(dir : bool, natop : nat) : store?
add(typeop : IntOpType, intop : int) : add?
sub(typeop : IntOpType, intop : int) : sub?
mult(typeop : IntOpType, intop : int) : mult?
div(typeop : IntOpType, intop : int) : div?
read(dir : bool, natop : nat) : read?
write(typeop : IntOpType, intop : int) : write?
jump(labop : posnat) : jump?
jgtz(labop : posnat) : jgtz?
jzero(labop : posnat) : jzero?
halt : halt?

END CommandBody
```

The type *CommandBody* has another type *IntOpType* as a parameter. A variable *typeop* ("type of operand") of the type *IntOpType* indicates the meaning of the integer operand *intop* in some commands. It can have one

of three values: *lit*, *dir*, or *indir*. If *typeop* is equal to *lit*, then the integer operand in the corresponding command should be interpreted as a literal. If *typeop* = *dir*, then *intop* means the index of a register with direct addressing, and if *typeop* = *indir*, then *intop* means the index of a register with indirect addressing.

The meaning of the commands and their parameters in the type *CommandBody* should be rather obvious, because it completely corresponds to their informal definition in Section 2. We have already explained the parameters *typeop* and *intop* of the commands LOAD, ADD, SUB, MULT, DIV and WRITE. The commands STORE and READ have a natural parameter *natop*, and a Boolean parameter *dir* that indicates the meaning of *natop*. If *dir* = *true*, *natop* means the index of a register with direct addressing, and if *dir* = *false*, *natop* means the index of a register with indirect addressing. The commands JUMP, JGTZ and JZERO have a positive natural parameter *labop* indicating the label of the command to which the program counter should jump if some condition is satisfied. The command HALT has no parameters.

To obtain the complete runs for RAM-AHU according to the method presented in Section 3, we need to define the effect predicate *Effect*, i.e. a Boolean predicate on pairs of states. This was done separately for each of the 12 commands of the machine. The effect predicates for most commands are rather large and cumbersome, and we see no need to present all of them here, because they correspond very closely to the intuitive meaning of the commands given in Section 2. To illustrate our approach, we only show the effect of commands HALT and LOAD.

The effect of the HALT command is very simple: the program counter becomes equal to the length of the program, so it no longer points to any command of the program (note that if the program length is *Len*, then its elements are enumerated from 0 to *Len – 1*). So if *s0* and *s1* are arbitrary states, the effect is defined as follows:

```
haltEffect(s0, s1) : bool =
    s1 = s0 WITH [ pCounter := length(program(s0)) ]
```

The LOAD command has two parameters: an integer operand *intop* and its type *typeop* with possible values *lit*, *dir* or *indir*. If if *s0* and *s1* are arbitrary states, then the effect of the LOAD command with arbitrary parameters *intop1* and *typeop1* is defined as follows:

```
loadEffect(s0, typeop1, intop1, s1) : bool =
      CASES typeop1 OF
   lit: loadLitEffect(s0, intop1, s1),
   dir: IF intop1 >= 0 THEN loadDirEffect(s0, intop1, s1)
        ELSE haltEffect(s0, s1) ENDIF,
```

```
   indir: IF intop1 >= 0 THEN loadIndirEffect(s0, intop1, s1)
          ELSE haltEffect(s0, s1) ENDIF
      ENDCASES
```

So, it is clear from this definition that the effect is defined according to three possible values of the parameter *typeop1*. If *typeop1* = *lit*, then *intop1* is a literal that should be loaded into the adder. This is defined by the predicate *loadLitEffect*:

```
loadLitEffect(s0, intop1, s1) : bool =
   s1 = s0 WITH
      [ registers := registers(s0) WITH [ (0) := intop1 ],
        pCounter := pCounter(s0) + 1 ]
```

If *typeop1* = *dir*, then *intop1* is the index of the register that should be loaded into the adder. This is defined by the predicate *loadDirEffect* given below; it uses the predicate *loadLitEffect* for loading a literal. If *intop1* <0, the LOAD command has illegal parameters and should have the same effect as the HALT command.

```
loadDirEffect(s0, natop1, s1) : bool =
   loadLitEffect(s0, registers(s0)(natop1), s1)
```

Finally, if *typeop1* = *indir*, then *intop1* is the index of the register that should be loaded into the adder via indirect addressing. This is defined by the predicate *loadIndirEffect* which is given below; it uses the predicate *loadDirEffect* for loading based on direct addressing. Again, if *intop1* <0, the LOAD command has illegal parameters and should have the same effect as the HALT command.

```
loadIndirEffect(s0, natop1, s1) : bool =
   IF registers(s0)(natop1) >= 0
      THEN loadDirEffect(s0, registers(s0)(natop1), s1)
   ELSE haltEffect(s0, s1) ENDIF
```

5. Search program and its specification in PVS

5.1. Our search program for an array of arbitrary size

The aim of our search program is to compute the maximum of an arbitrary number of integers located in the beginning of the input tape; they are preceded by the size of the array N. Initially, the input tape contains the size of the array N and all its elements. The program starts by reading N and the first element of the array, which becomes a potential maximum. After that, it has a loop to examine all remaining elements of the array.

The first statement of the loop uses the value of N - 1 to check whether there are still items to be read. During each iteration of the loop, we read one remaining element and compare it to the maximum of the elements that have already been read; the larger of these numbers becomes the new potential maximum. The number of the remaining elements serves as a loop counter, and it is decreased by one before each subsequent iteration of the loop. After there are no more elements to be read, we conclude that the potential maximum is the largest element of the array, and we write it to the output tape. Below we list the commands of the program. Three of them have a label, indicating a beginning of the loop, a jump to a value greater than zero, and the first command after the loop, respectively.

```
          READ 0
          READ 1
          SUB =1
beginl:   JZERO leavel
          STORE 3
          READ 2
          LOAD 1
          SUB 2
          JGTZ gzero
          LOAD 2
          STORE 1
gzero:    LOAD 3
          SUB =1
          JUMP beginl
leave1:   WRITE 1
          HALT
```

It is easy to see how our program computes the largest of all numbers in the input array. The first command reads the size of the array N and places it into the adder. After that, the second command reads the first integer from the array and places it into the register with index 1. The third command subtracts 1 from the adder, so it now contains $N - 1$.

After that, the command labeled *beginl* is the first command in a loop. It checks whether $N - 1 = 0$. If it is so, then the array has only one element. Therefore, it is the maximum, so we can immediately exit the loop. The program counter moves to the command with label *leavel*, which writes the first element of the array from the register with index 1 to the output tape, and finally the HALT command terminates the computation.

However, if $N - 1 > 0$ (remember that $N > 0$), this means that there are still elements of the input array waiting to be read from the tape. So we proceed with the loop in order to compare the first element with these remaining elements. Firstly, the STORE command stores the value of $N - 1$ into the register with index 3 for later use as a loop counter. Next, the

READ command reads the second integer from the array, and places it into the register with index 2.

In the remaining part of the loop, we compute the maximum of the first two elements of the array, and place it into the register with index 1. To achieve that, the LOAD command loads the value of the first element into the adder. After that, the SUB command subtracts the second element from the first one and places the result into the adder. If its value is greater than 0, then the larger of the two elements is already in the register with index 1. Therefore, as a result of the JGTZ command, the program counter moves to the command labeled *gzero*. However, if the opposite is true, then the LOAD and STORE commands are executed, which change the value in the register with index 1 from the first element of the array to the second one. In both cases, when the program counter arrives at the command LOAD 3, the larger of the first two elements is located in the register with index 1. If there are still array elements waiting to be read from the tape, we must compare this maximum with these remaining elements; otherwise we must exit the loop and write the maximum value to the output tape. In order to check that, at the end of the loop the LOAD and SUB commands compute the value of $N - 2$ and place it into the adder. After that, the JUMP command unconditionally moves the program counter to the beginning of the loop, where we determine whether to start the next iteration of the loop or to exit the loop. The subsequent iterations of the loop proceed in a completely similar manner.

5.2. The implementation of our search program in PVS

It is fairly straightforward to implement our search program in PVS. The resulting PVS version consists of 16 commands numbered from *com0* to *com15* which are given below. The text labels *beginl*, *gzero* and *leavel* are replaced here by the natural numbers 5, 10 and 15.

```
com0 : Commands = (# label := 0, body := read(TRUE, 0) #)

com1 : Commands = (# label := 0, body := read(TRUE, 1) #)

com2 : Commands = (# label := 0, body := sub(lit, 1) #)

com3 : Commands = (# label := 5, body := jzero(15) #)

com4 : Commands = (# label := 0, body := store(TRUE, 3) #)

com5 : Commands = (# label := 0, body := read(TRUE, 2) #)

com6 : Commands = (# label := 0, body := load(dir, 1) #)
```

```
com7 : Commands = (# label := 0, body := sub(dir, 2) #)

com8 : Commands = (# label := 0, body := jgtz(10) #)

com9 : Commands = (# label := 0, body := load(dir, 2) #)

com10 : Commands = (# label := 0, body := store(TRUE, 1) #)

com11 : Commands = (# label := 10, body := load(dir, 3) #)

com12 : Commands = (# label := 0, body := sub(lit, 1) #)

com13 : Commands = (# label := 0, body := jump(5) #)

com14 : Commands = (# label := 15, body := write(dir, 1) #)

com15 : Commands = (# label := 0, body := halt #)
```

We constructed a PVS program *SearchProg* consisting of these 16 commands which looks as follows.

```
SearchProg : finite_sequence[Commands] =
   (# length := 16,
      seq := (LAMBDA (k : below[16]):
         COND
             k = 0 -> com0, k = 1 -> com1, k = 2 -> com2,
             k = 3 -> com3, k = 4 -> com4, k = 5 -> com5,
             k = 6 -> com6, k = 7 -> com7, k = 8 -> com8,
             k = 9 -> com9, k = 10 -> com10, k = 11 -> com11,
             k = 12 -> com12, k = 13 -> com13, k = 14 -> com14,
             k = 15 -> com15
         ENDCOND)
    #)
```

We also defined the input tape *InputSeq* on which the computation of *SearchProg* should begin. The size of an integer array which will be searched by the program is defined by an arbitrary natural number N that must be positive.

```
N : posnat
```

The array itself is defined as a finite sequence of integers of length N. To achieve that, we define the PVS type *FinArrayN* that contains all possible finite sequences of integers of length N.

```
finseq : VAR finite_sequence[int]

FinArrayGZ : TYPE = { finseq | length(finseq) > 0 }

finarr : VAR FinArrayGZ

FinArrayN : TYPE = { finarr | length(finarr) = N }
```

After that, an array *InputArray* is defined as an arbitrary constant of the type *FinArrayN*.

```
InputArray : FinArrayN
```

Finally, the input tape *InputSeq* is defined as a sequence which begins with the size of the array N. After that, it contains all elements from *InputArray* followed by a string of zeros. The definition of *InputSeq* is given below.

```
InputSeq : sequence[int] =
   LAMBDA k : IF k = 0 THEN N
              ELSE IF k < N + 1 THEN seq(InputArray)(k - 1)
                 ELSE 0 ENDIF
              ENDIF
```

Since N and *InputArray* are arbitrary constants, it is clear from this definition of *InputSeq* that it models any possible input tape for the program *SearchProg*.

The initial state *SearchIni* of RAM-AHU for *SearchProg* and *InputSeq* is defined in the same way as was presented in Section 4: they are included in the state, an empty sequence *EmptyIntSeq* (i.e. a sequence consisting of only zeros) is assigned to the fields *registers* and *outputTape*, and 0 is assigned to the program counter and the variables *inputHead* and *outputHead*. After that, we can use our definitions from Section 3 and obtain the set of complete runs for *SearchIni*.

6. Specification of the correctness property and the complexity bounds

The correctness property for the program *SearchProg* is as follows: it terminates for any N and any elements of the array *InputArray* and, in the last state of its complete run, there is exactly one number written on its output tape equal to the maximum of all elements in *InputArray*. Since *InputArray* has an arbitrary number of elements, namely N, it is not trivial to define its maximum. To achieve that, we use a powerful mechanism available in

PVS – *recursive functions*. Firstly, if *finarr* is some finite sequence of integers with at least 2 elements, we define an auxiliary function *SubSeq(finarr)* (not shown here) which removes from *finarr* its last element. After that, for an arbitrary nonempty sequence of integers *finarr*, we recursively define the function *MaxValue(finarr)* as follows. If *finarr* has exactly one element, then *MaxValue(finarr)* is equal to that element. Otherwise, *MaxValue(finarr)* is defined as a maximum of *MaxValue(SubSeq(finarr))* and the last element of *finarr*. The complete definition in PVS looks as follows (the measure function is needed to guarantee that recursion terminates):

```
MaxValue(finarr) : RECURSIVE int =
   IF length(finarr) = 1 THEN seq(finarr)(0)
   ELSE max(MaxValue(SubSeq(finarr)), seq(finarr)(length(finarr) - 1))
      ENDIF
   MEASURE length(finarr)
```

If *crun* is an arbitrary complete run, we can use the function *MaxValue* and define the correctness property for it as follows (here the function *last* gives the last element of a finite sequence):

Correct(crun) =
fin?(crun) &
outputHead(last(crun)) = 1 &
outputTape(last(crun))(0) = MaxValue(InputArray)

We proved in PVS the following theorem called Main which establishes not only the functional correctness of any complete run for our program, but also the lower and upper bounds on the number of states in it:

$\forall$ *crun : Correct(crun)* &
length(crun) $\geq$ *9*(N – 1) + 7* &
length(crun) $\leq$ *11*(N – 1) + 7* Main

It is clear from the theorem Main that all executions of our search program consist of at least *9*(N – 1) + 6* and at most *11*(N – 1) + 6* commands, because the number of states in any finite run exceeds the number of commands by 1. For example, if some run is a sequence of states *s0 s1 s2 s3*, then there are exactly 3 commands leading from *s0* to *s3*. So the theorem Main implies that the best-case execution time of the program *SearchProg* is *9*(N – 1) + 6* commands, and its worst-case execution time is *11*(N – 1) + 6* commands. For example, if $N = 1$, all executions of the program consist of exactly 6 commands, and if $N = 2$, they consist of at least 15 and at most 17 commands.

The proof of the theorem Main consists of about 175 PVS theorems and lemmas. Checking the proof takes less than 4 minutes on a regular PC. In the next section, we present the proof itself.

7. Proof of the theorem Main

Like all PVS proofs, the proof of the theorem Main is structured as a tree. The root of our tree is the theorem Main, and most of its leaves are lemmas InfIniLem, InfEffLem, FinIniLem, FinEffLem and FinLastLem, which will be given below. These lemmas, which we call *elementary lemmas*, follow directly from the definition of complete runs as it was given in Section 3. We only need to replace in that general definition the initial state *Ini* by its instance *SearchIni* for the program *SearchProg*.

The lemmas InfIniLem and InfEffLem describe the basic properties of infinite complete runs. The lemma InfIniLem expresses that the first state in any infinite complete run must be equal to the initial state. It follows directly from clause 1 in the definition of infinite complete runs.

$\forall$ *crun : inf?(crun)* $\Rightarrow$ *crun(0) = SearchIni* InfIniLem

The elementary lemma InfEffLem means that in any infinite complete run each state should be obtained from the previous state according to the effect predicate. It follows directly from clause 2 in the definition of infinite complete runs.

$\forall$ *crun : inf?(crun)* $\Rightarrow \forall$ *i : Effect(crun(i), crun(i + 1))* InfEffLem

The lemmas FinIniLem, FinEffLem and FinLastLem describe the elementary properties of finite complete runs. The lemma FinIniLem expresses that any finite complete run must have at least one state and its first state must be equal to the initial state. It follows directly from clause 1 in the definition of finite complete runs.

$\forall$ *crun : fin?(crun)* $\Rightarrow$ *length(crun)* >0 & *crun(0) = SearchIni* FinIniLem

The lemma FinEffLem means that in any finite complete run each state should be obtained from the previous state according to the effect predicate. It follows directly from clause 2 in the definition of finite complete runs.

$\forall$ *crun : fin?(crun)* $\Rightarrow$
$\forall$ *i : i* $<$*length(crun) – 1* $\Rightarrow$ *Effect(crun(i), crun(i + 1))* FinEffLem

Finally, the elementary lemma FinLastLem expresses that the last state of any finite complete run should be final (in the sense defined in Section 3). It follows from clause 3 in the definition of finite complete runs.

$\forall$ *crun : fin?(crun)* $\Rightarrow$ *isFinal(last(crun))* FinLastLem

Now we continue with the proof. Let *crun* be an arbitrary complete run which can be either infinite or finite. Below we consider both possible cases.

The case of an infinite complete run. If *crun* is infinite, our goal is to prove that this is impossible, i.e. obtain a contradiction. This is done by showing that the program counter in an infinite run will eventually exceed the length of the program. We proved the following lemmas *BeforeLoop*

and *BadCounter* which describe how the program counter changes before reaching the first command in the loop and after that, respectively. The lemma *BeforeLoop* expresses that in the state with index 3 (i.e. the state that is obtained after the execution of the first 3 commands) the counter points to the command *com3*, i.e. the first command in the loop, and the value of the adder is equal to *N – 1*.

$$\forall\ crun : inf?(crun) \Rightarrow \\ (pCounter(crun(3)) = 3\ \&\ registers(crun(3))(0) = N - 1)$$ BeforeLoop

We do not give here the proof of the lemma *BeforeLoop*, but it is easy: it is sufficient to examine the effect of the commands *com0*, *com1* and *com2* (which was explained in Section 4). Next, the lemma *BadCounter* expresses that if there is a state with some index j in which the program counter is equal to 3 (i.e. it points to the first command in the loop) and the value of the adder is equal to some arbitrary natural number k, then there exists some index $n$such that in the state with index n, the program counter is equal to 16.

$$\forall\ crun, j, k : inf?(crun)\ \&\ pCounter(crun(j)) = 3\ \& \\ registers(crun(j))(0) = k \Rightarrow \\ \exists n : pCounter(crun(n)) = 16$$ BadCounter

The proof of the lemma *BadCounter* will be given later. Using the lemmas *BeforeLoop* and *BadCounter*, we can prove the termination of our program. Indeed, the lemma *BeforeLoop* gives us $pCounter(crun(3)) = 3$ and $registers(crun(3))(0) = N - 1$. If in the lemma *BadCounter* we now take $j = 3$ and $k = N - 1$, we obtain that there exists some index $n$such that $pCounter(crun(n)) = 16$. Applying the lemma InfEffLem to the state with index n, we obtain $Effect(crun(n), crun(n + 1)) = true$. However, this leads to a contradiction with the definition of the effect predicate, because a state with such a large value of the program counter cannot be related to any other state by the effect predicate. This means that *crun* cannot be infinite, which establishes the termination of our program for any input data.

Proof of the lemma *BadCounter*. In the proof of the lemma *BadCounter*, we first prove an additional lemma *AdderZero*, which expresses that after the program counter reaches the first command in the loop, the value of the adder will eventually decrease to 0 from an arbitrary natural number k (after some iterations of the loop).

$$\forall\ crun, j, k : inf?(crun)\ \&\ pCounter(crun(j)) = 3\ \& \\ registers(crun(j))(0) = k \Rightarrow \\ \exists m : pCounter(crun(m)) = 3\ \&\ registers(crun(m))(0) = 0$$ AdderZero

The lemma *AdderZero* is proved by induction on the index k. The basis

of the induction is obvious. However, the proof of the induction step is quite complex, so we can only explain its general idea here. Suppose we are in the beginning of the loop, and the adder is equal to $k+1$. By examining all actions in the loop, including the jump instructions, we prove that the adder decreases exactly by one during a single iteration of the loop. So it will decrease to k after a single iteration. We can now apply the induction hypothesis and conclude that the adder will decrease to 0 after a few more iterations of the loop. This completes the proof of *AdderZero*.

Using the lemma *AdderZero*, it is easy to prove the lemma *BadCounter*. Indeed, if we are currently in the beginning of the loop and the adder is equal to zero, it will reach the value of 16 after the actions *com3*, *com14* and *com15* are executed.

The case of a finite complete run. If *crun* is finite, our aim is to prove that eventually, after at least *9*(N – 1) + 6* and at most *11*(N – 1) +6* commands, the final state will be reached in which the output tape contains exactly one value *MaxValue(InputArray)*. To show this, we must first prov the following complicated lemma *EndLoopReached*.

$\forall$ *crun : fin?(crun)* $\Rightarrow$ EndLoopReached
$\exists i : i \geq$ *9*(N – 1) + 3* & $i \leq$ *11*(N – 1) + 3* & *length(crun)* $> i$ &
pCounter(crun)(i) = 3 &
registers(crun)(0) = 0 & *registers(crun)(1) = MaxValue(InputArray)* &
outputHead(crun)(0) = 0

Despite its complex definition, the intuitive meaning of the lemma *EndLoopReached* is easy to explain. Indeed, it expresses that eventually, after at least *9*(N – 1) + 3* and at most *11*(N – 1) +3* commands, the state will be reached where the program is ready to leave the loop after computing the correct result. More precisely, it is the state where the program counter points to the first command of the loop, the value of the adder is 0, and the value of the next register is *MaxValue(InputArray)*.

The proof of the lemma *EndLoopReached* is very long and complex; we do not show it here. We can now use that lemma to prove the main theorem. Indeed, suppose that after at least *9*(N – 1) + 3* and at most *11*(N – 1) +3* commands, the state has been reached where the program counter points to the command *com3*, the value of the adder is 0, the value of the register with index 1 is *MaxValue(InputArray)*, and there are no elements on the output tape. We now consider the effect of all subsequent commands according to the lemma FinEffLem. Applying the effect of the command *com3*, we obtain that in the next state, i.e. after at least *9*(N – 1) + 4* and at most *11*(N – 1) +4* commands, we are in the state where the program counter points to the command *com14*, the value of the register with index 1 is *MaxValue(InputArray)*, and there are still no elements on the output tape. Next considering the effect of the command

com14, we conclude that in the next state, i.e. after at least *9*(N – 1) + 5* and at most *11*(N – 1) +5* commands, we are in the state where the program counter points to the command *com15*, and there is a single element in the beginning of the output tape equal to *MaxValue(InputArray)*.

Finally, we apply the effect of the halt command *com15*, and obtain that after at least *9*(N – 1) + 6* and at most *11*(N – 1) +6* commands, we have reached the state where there are no more commands to perform, and there is a single element on the output tape equal to *MaxValue(InputArray)*. Therefore, we have reached the final state of *crun* after computing the maximum of *InputArray*, and after the number of steps specified by the theorem Main. This concludes the proof of the theorem Main.

8. Conclusion

In this paper, we use a method for the verification of RAM-AHU programs which allows proving their functional correctness and complexity measures within a single formal framework. Our method has been applied to a non-trivial example – a program that has a loop to compute the maximum of an integer array of arbitrary size. The verification was done using the interactive proof checker of PVS. We believe that this example would have presented considerable difficulties for the fully automated techniques such as model-checking [7], which justifies the use of deductive verification.

Unlike most works on complexity theory, we do not use the $O(n)$ notation, but express the best-case and worst-case complexity measures for our program with exact constants.

For example, for the particular program studied here, we prove that all its executions consist of at least *9*(N – 1) + 6* and at most *11*(N – 1) + 6* commands, where N is the size of the input array.

Finite-memory automata, also called register automata (for example, [11, 2, 9]) are an important class of automata-based models which has significant similarities to ARMs. In register automata, in addition to their control state, a finite number of registers can be used to store and compare letters from the input word. The only test available for letters is equality. There are operations for managing registers like store, move and delete (similar to ARM), but no arbitrary arithmetical operations. Some important problems are decidable for register automata; for example, the emptiness problem is PSPACE-complete. Register automata are very well suited for the modeling of communication protocols and the definition of formal languages, but not for numerical computations in arbitrary domains or processing of complex data structures.

In our future work, we would like to continue the study of more complex algorithms for the register-based architectures such as RAM-AHU and their formal verification. In particular, we are interested in sorting algorithms.

While attempting to program them on the existing RAM-AHU architecture, we discovered the difficulty of that task. It became apparent that the basic data structure of RAM-AHU, in particular its set of registers, is too limited to conveniently store the preliminary results of computations. This problem can be solved by adding extra register sets, together with commands that copy or move data between the main set of registers and these extra sets.

References

[1] Aho A.V., Hopcroft J.E., Ullman J.D. The Design and Analysis of Computer Algorithms. – Addison-Wesley Publishing Company, 1976.

[2] Benedikt Michael, Ley Clemens. Automata vs. logics on data words // Lect. Notes Comput. Sci. – 2010. – 6247. – P. 110–124.

[3] Boolos G.S., Burgess J.P., Jeffrey R.C. Computability and Logic (Fourth Edition). – Cambridge University Press, Cambridge, England, 2002.

[4] Chkliaev D.A. Mechanical Verification of Concurrency Control and Recovery Protocols: PhD thes. / Eindhoven University of Technology. – 2001.

[5] Chkliaev D.A., Nepomniaschy V.A. Deductive verification of the sliding window protocol // Modeling and Analysis of Information Systems. – 2012. – Vol. 19, No. 6. – P. 57–68 (In Russian).

[6] Chkliaev D.A., Nepomniaschy V.A. Formal verification of programs for abstract register machines // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2013. – IIS Special Iss. 35. – P. 39–56.

[7] Edmund A. Clarke, Jr., Orna Grumberg & Doron A. Peled. Model Checking. – MIT Press, 1999.

[8] Cook S.A., Reckhow R.A. Time-bounded random access machines // J. of Computer Systems Science. – 1973. – Vol. 7. – P. 354–375.

[9] Stephane Demri, Ranko Lazic. LTL with the freeze quantifier and register automata // LICS'06. – 2006. – P. 17–26.

[10] Hartmanis J. Computational complexity of random access stored program machines // Mathematical Systems Theory. – 1971. – Vol. 5, No. 3. – P. 232–245.

[11] Kaminsky M., Francez N. Finite-memory automata // Theor. Comput. Sci. – 1994. – Vol. 134(2). – P. 329–363.

[12] Knuth D.E. The Art of Computer Programming.– Addison-Wesley V.1, 1968.

[13] Lambek J. How to program an infinite Abacus // Mathematical Bulletin. – 1961. – Vol. 4, No. 3. – P. 295–302.

[14] Minsky M.L. Computation: Finite and Infinite Machines. – Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

[15] Owre S., Rushby J.M., Shankar N. PVS: a Prototype Verification System // Lect. Notes Comput. Sci. – 1992. - Vol. 607. - P. 748–752.

[16] Schonhage A. Storage modification machines. // SIAM J. on Computing. – 1980. – Vol. 9, No. 3. – P. 490–508.