Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 39–56

Formal verification of programs for abstract register machines

D. A. Chkliaev, V. A. Nepomniaschy

Abstract. Abstract register machines are an important formal model of computation widely used for the modeling of many classes of computational algorithms and the analysis of their complexity. Despite the significance of this model, formal verification of general-purpose programs for abstract register machines has not been properly covered in the existing literature. To fill this gap, we provide a formal specification for one version of abstract register machines – the random-access machine invented by Aho, Hopcroft and Ullman. Executions of this machine are formalized by a transition system in the language of the verification system PVS. We also specify in PVS a simple search program for this architecture and its correctness property. The property is proved using the interactive proof checker of PVS. We were able to prove not only the functional correctness of the program, but also its time complexity, which shows the novelty of our approach.

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

1. Introduction

Abstract register machines (ARMs) [9, 2], together with Turing machines and recursive functions, belong to the most important formal models of computation. However, unlike Turing machines and recursive functions, they are much less abstract and much closer to the operations of actual computers. Indeed, although ARMs are not directly based on any computer architecture used in practice, they are still able to imitate closely the structure of modern CPUs and their arithmetical, logical and input/output commands. For this reason, the formalism of ARMs has been used during several decades for the formal, mathematical study of computational algorithms and programs. Perhaps the best known example of this approach is shown in [7], where a highly detailed version of an abstract register machine is used to define rigorously many important classes of computational algorithms and to prove their complexity measures.

Although a lot of research has been dedicated to the verification of programs written in high-level languages such as C and Java (and also to the correctness of some register-based processor designs), we are not aware of any works that systematically study the formal verification of computational programs for ARMs. This is unfortunate, because formal verification can greatly improve our understanding of programs and algorithms, and help to

find and remove subtle errors in them. Such verification is especially rigorous and convincing when it uses some form of automated support, for example an interactive theorem prover. Since ARMs have been successfully used to analyze the complexity measures for many types of computer algorithms, it is an interesting challenge to create a formal framework that allows proving not only the functional correctness of programs for them, but also their time and space complexity.

In the dissertation [3], a promising general method was presented for the specification and verification of distributed protocols. In [3], it was used to verify several non-trivial examples from the field of databases, and in [4], it was successfully applied to the Sliding Window protocol. Here we show how a modification of our method can be used to provide a formal framework for the verification of programs for ARMs. We consider one particular version of ARMs – the random-access machine invented by Aho, Hopcroft and Ullman [1], which we call here RAM-AHU. In our method, we represent the data structures of RAM-AHU in the language of the verification system PVS [10] 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.

We use our method to verify a simple search program for RAM-AHU. A transition system is generated, and for that system the correctness property of the program is defined as some relation between the input sequence in the initial state and the output sequence in the final state. This allows us to verify formally the correctness property using the interactive theorem prover of PVS, which leads to a proof of correctness that is both rigorous and intuitively understandable. We were able to prove not only the functional correctness of the search program, but also its best-case and worst-case execution time. We hope that this fact demonstrates the novelty of our specification method, because the functional correctness of programs and their time complexity are usually analyzed and proved using different formal frameworks.

The rest of the paper is organized as follows. In Section 2, we give a brief introduction to the PVS system. Section 3 describes RAM-AHU on the basis of [1]. In Section 4, we present our formalization of the executions of abstract register machines which can also be applied to RAM-AHU. In Section 5, the RAM-AHU data structures are specified in PVS, and in Section 6 its commands are formalized in PVS. Section 7 presents a specification of a simple search program for RAM-AHU and verification of its correctness property. Finally, Section 8 gives some remarks on related works and possible future work.

2. The PVS verification system

The PVS system [10], created at the Stanford Research Institute about two decades ago, 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. PVS is able to combine an expressive specification language with powerful automated deduction, which allows it to handle many examples that present considerable difficulties for other verification systems.

The specification language of PVS is based on the classical higher-order logic. The main types include uninterpreted types, which may be introduced by a user, as well as built-in types such as Booleans, natural and real numbers. The constructors of types include functions, sets, records, tuples and enumerations, as well as recursively defined abstract datatypes (for example, lists and binary trees). It is also possible to define predicate subtypes such as the type of prime numbers. The specifications are organized into a hierarchy of parameterized theories which contain assumptions, definitions, axioms and theorems. Expressions of the PVS language provide usual arithmetical and logical operations, as well as application of functions, lambda abstraction and quantifiers, all with natural syntax. The predefined theories contain hundreds of useful definitions and lemmas.

In the prover of PVS, every goal or subgoal is displayed in the following form:

```
{-1}  A1
{-2}  A2
 [-3]  A3
...
|-------------
 [1]   B1
{2}   B2
{3}   B3
...
```

This display is a *sequent*: formulas above the dashed line (A1, A1, A3...) are called *antecedents* and those below (B1, B1, B3...) are called *consequents*. The sequent is interpreted as follows: conjunction of the antecedents implies disjunction of the consequents. The lists of antecedents and consequents may both be empty (an empty antecedent is equivalent to *true*, and an empty consequent is equivalent to *false*).

The proof of every theorem in PVS begins with a single consequent (representing the theorem). The objective of the proof is to create a *proof tree* of

sequents in which all leaves are trivially true. The prover is always attempting to prove some unproved leaf in the tree. It can accomplish this task by invoking one of its commands, which either proves the current sequent (usually by applying some of the decision procedures) or splits it into several easier subgoals. When there are no more unproven branches in the tree, the prover notifies the user that the proof is complete. The resulting proof is automatically stored in a file and can be run again later. The PVS system has extensive facilities for managing the proofs and displaying information about them.

In our PVS specification of RAM-AHU, we mostly use natural numbers and integers to represent some variables in the data structures of the machine, and an abstract datatype to model the commands of the machine. Additional datatypes are constructed from these basic types by applying records, finite and infinite sequences and predicate subtypes. Many predicates and lambda functions are also used to generate the whole specification and a search program for RAM-AHU. Verification of the program relies on the PVS decision procedures for Boolean logic and arithmetical operations on natural numbers and integers.

3. RAM-AHU

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.

4. Our 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 [3] and [4]. 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 $\mathit{Effect}(s1, s2) = false$. 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 $\mathit{Effect}(s1, s2) = true$.

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):

$$isFinal(st) = (pCounter(st) \geq length(program(st)))$$

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. $isFinal(last(R)) = true$.

5. Data structures of the machine in PVS

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 the 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_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 #)
```

6. Commands of the machine in PVS

It was already said in the previous section 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 3. 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 4, 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 3. 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 *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 the 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 *typeop*1 = *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 *intop*1 < 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
```

7. Verification of a search program

7.1. The program and its specification in PVS

To illustrate our method for the verification of programs for RAM-AHU, we use it to verify a simple search program. The aim of the program is to compute the larger of two integers located in the beginning of the input tape. Even for such a simple task, the resulting program is not particularly short. It consists of 9 commands numbered from *com*0 to *com*8 which are given below.

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

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

com2 : Commands = (# label := 0, body := store(TRUE, 2) #)

com3 : Commands = (# label := 0, body := sub(dir, 1) #)

com4 : Commands = (# label := 0, body := jgtz(1) #)

com5 : Commands = (# label := 0, body := write(dir, 1) #)

com6 : Commands = (# label := 0, body := jump(2) #)

com7 : Commands = (# label := 1, body := write(dir, 2) #)

com8 : Commands = (# label := 2, body := halt #)
```

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

```
SearchProg : finite_sequence[Commands] =
   (# length := 9,
      seq := (LAMBDA (k : below[9]):
          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
          ENDCOND)
    #)
```

We also defined the input tape *SearchSeq* on which the computation of *SearchProg* should begin. It contains arbitrary integers $int0$ and $int1$ followed by a string of zeros. The definition of *SearchSeq* is given below.

```
EmptyIntSeq : sequence[int] = LAMBDA n : 0

int0, int1 : int

SearchSeq : sequence[int] =
    EmptyIntSeq WITH [(0) := int0, (1) := int1]
```

Since $int0$ and $int1$ are arbitrary constants of the type integer, it is clear from this definition of *SearchSeq* that it models any possible input tape for the program *SearchProg*.

It is easy to see how the program *SearchProg* computes the larger of $int0$ and $int1$. The command $com0$ reads the integer $int0$ and places it into the register with index 0 (the adder). After that, the command $com1$ reads the integer $int1$ and places it into the register with index 1. Since another copy of $int0$ will be needed shortly, the command $com2$ stores $int0$ into the register with index 2.

After that, the command $com3$ subtracts $int1$ from $int0$ and places the result into the adder. If it is greater than 0, then $int0 > int1$, so the command $com4$ moves the program counter to the command with label 1, i.e. the command $com7$. The command $com7$ writes $int0$ from the register with index 2 to the output tape, and the command $com8$ terminates the computation. However, if $int0 \leq int1$, the program counter moves to the command $com5$. The command $com5$ writes $int1$ from the register with index 1 to the output tape. After that, the command $com6$ unconditionally moves the program counter to the command $com8$. Again, the command $com8$ terminates the computation.

The initial state *SearchIni* of RAM-AHU for *SearchProg* and *SearchSeq* is defined in the same way as was presented in Section 5: 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 4 and obtain the set of complete runs for *SearchIni*.

7.2. Specification and verification of the correctness property

The correctness property for the program *SearchProg* is as follows: it terminates for any values of $int0$ and $int1$ (i.e. the numbers in the beginning of its input tape in the initial state) and, in the last state of its complete run, there is exactly one number written on its output tape equal to the maximum of $int0$ and $int1$. If *crun* is an arbitrary complete run, the correctness property for it is defined as follows (here the function *last* gives the last element of a finite sequence):

$$
\begin{aligned}
&Correct(crun) = \\
&\quad fin?(crun)\ \&\\
&\quad outputHead(last(crun)) = 1\ \&\\
&\quad outputTape(last(crun))(0) = max(inputTape(SearchIni)(0),\\
&\qquad\qquad\qquad\qquad\qquad\qquad inputTape(SearchIni)(1))
\end{aligned}
$$

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

$$\forall crun : Correct(crun)\ \&\ (length(crun) = 8\ OR\ length(crun) = 9) \tag{Main}$$

It is clear from the theorem Main that all executions of our search program consist of exactly 7 or 8 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 7 commands, and its worst-case execution time is 8 commands.

The proof of the theorem Main consists of about 40 PVS theorems and lemmas. Checking the proof takes less than 1 minute on a regular PC. Below we present the proof itself.

Proof of the theorem Main. Like all PVS proofs, our proof 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 4. 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.

$$\forall crun : inf?(crun) \Rightarrow crun(0) = SearchIni \qquad \text{(InfIniLem)}$$

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.

$$\forall crun : inf?(crun) \Rightarrow \forall i : \mathit{Effect}(crun(i), crun(i+1)) \qquad \text{(InfEffLem)}$$

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.

$$\forall crun : fin?(crun) \Rightarrow length(crun) > 0 \ \&\ crun(0) = SearchIni \qquad \text{(FinIniLem)}$$

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.

$$\forall crun : fin?(crun) \Rightarrow \forall i : i < length(crun) - 1 \Rightarrow \mathit{Effect}(crun(i), crun(i+1)) \qquad \text{(FinEffLem)}$$

Finally, the elementary lemma FinLastLem expresses that the last state of any finite complete run should be final (in the sense defined in Section 4). It follows from clause 3 in the definition of finite complete runs.

$$\forall crun : fin?(crun) \Rightarrow isFinal(last(crun)) \qquad \text{(FinLastLem)}$$

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 lemma *BadCounter* which expresses that the program counter will reach the value of 9 either in the state with index 7 or in the state with index 8:

$$\forall crun : inf?(crun) \Rightarrow$$
$$(pCounter(crun(7)) = 9 \; OR \; pCounter(crun(8)) = 9) \quad \text{(BadCounter)}$$

The proof of lemma *BadCounter* is rather long and complex; we do not present it here. Using this lemma, we can easily obtain a contradiction with the elementary lemma InfEffLem. Indeed, applying InfEffLem we obtain $Effect(crun(7), crun(8)) = true$ and $Effect(crun(8), crun(9)) = true$. If $pCounter(crun(7)) = 9$, this leads to a contradiction with $Effect(crun(7), crun(8)) = true$ and 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. For the same reason, $pCounter(crun(8)) = 9$ creates a contradiction with $Effect(crun(8), crun(9)) = true$ and the definition of the effect predicate. Since both cases lead to a contradiction, this means that *crun* cannot be infinite. This result establishes the termination of our program for any input data.

The case of a finite complete run. If *crun* is finite, our aim is to prove that eventually the final state will be reached in which the output tape contains either $int0$ or $int1$, depending on which of these numbers is larger. If $int0 > int1$, such a state will be reached after executing exactly 7 commands, and if $int0 \leq int1$, it will be reached after exactly 8 commands. We proved the following lemmas ShortPathLem and LongPathLem which describe both possible cases:

$$\forall crun : fin?(crun) \;\&\; int0 > int1 \Rightarrow$$
$$length(crun) > 7 \;\&\; pCounter(crun(7)) = 9 \;\&$$
$$outputHead(crun(7)) = 1 \;\&\; outputTape(crun(7))(0) = int0 \quad \text{(ShortPathLem)}$$

$$\forall crun : fin?(crun) \;\&\; int0 \leq int1 \Rightarrow$$
$$length(crun) > 8 \;\&\; pCounter(crun(8)) = 9 \;\&$$
$$outputHead(crun(8)) = 1 \;\&\; outputTape(crun(8))(0) = int1 \quad \text{(LongPathLem)}$$

We do not discuss here the proofs of lemmas ShortPathLem and LongPathLem because of their large size and complexity. Using these lemmas, we can easily prove the theorem Main. Indeed, if $int0 > int1$, we apply the lemma ShortPathLem and obtain: $pCounter(crun(7)) = 9$, $outputHead(crun(7)) = 1$ and $outputTape(crun(7))(0) = int0$. If we assume that the state with index 7 is not the last state of *crun*, we can obtain a contradiction with the elementary lemma FinEffLem and the definition of the effect predicate. This means that $length(crun) = 8$ and

$crun(7) = last(crun)$. We can easily see that the theorem Main is satisfied for $crun$. If $int0 \leq int1$, we apply the lemma LongPathLem and obtain: $pCounter(crun(8)) = 9$, $outputHead(crun(8)) = 1$ and $outputTape(crun(8))(0) = int1$. As in the previous case, we prove that the state with index 8 is the last state of $crun$. Therefore, $length(crun) = 9$ and $crun(8) = last(crun)$. Again, the theorem Main is satisfied for $crun$. This completes the proof of Main.

8. Conclusion

Abstract register machines (ARMs), which include counter machines [8] and pointer machines [11], as well as more realistic models of hardware such as random-access machines [5] and random-access stored-program machines [6], are an important model aimed at rigorous analysis of computer algorithms. We presented here a formal framework for the specification and verification of computational programs for ARMs – something not presented in [5, 6, 8, 11] and subsequent works on this model. As we already mentioned, our framework allows proving not only the functional correctness of such programs, but also their best-case and worst-case time complexity.

The version of ARMs considered here (based on the book [1]) has not only significant similarities to the random-access machine from [5], but also some differences. For example, the only primitive arithmetic commands in [5] are addition and subtraction, but there is a mechanism that allows creating arrays. However, our version has little in common with pointer machines from [11]. The computational part of pointer machines is so primitive that they cannot perform arbitrary arithmetic operations, and this makes them unsuitable for programming of complex algorithms.

To illustrate our method of verification, we used it to verify a search program which computes the larger of two arbitrary integers. Despite the apparent simplicity of this example, we believe that it is far from trivial. Indeed, since the initial data for our program belong to an infinite domain $\mathbf{Z} \times \mathbf{Z}$, we verified its correctness for an unlimited number of possible executions. This is something that is rather challenging for fully automated techniques such as model-checking, but can be done using deductive verification. An additional advantage of our approach is the fact that we were able to prove the exact time complexity of the program: its executions consist of at least 7 and at most 8 commands.

Naturally, the search program considered in this paper is only the first step in our investigation of programs for abstract register machines and their formal verification. In our future work, we would like to verify a program for RAM-AHU that performs a search in an array of an arbitrary size. Another interesting possibility is to investigate how to efficiently sort large arrays on this architecture, and also to verify formally such sorting programs. It

is well-known that programs that process data structures of arbitrary sizes usually cannot be verified fully automatically. Therefore, it seems completely appropriate to use deductive verification in order to ensure their correctness. We also plan to extend our framework so that it would allow us to prove not only the time complexity of programs, but also their space complexity.

References

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

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

[3] Chkliaev D.A. Mechanical Verification of Concurrency Control and Recovery Protocols: PhD thes. – Eindhoven University of Technology, 2001. – Available at http://alexandria.tue.nl/extra2/200112908.pdf

[4] Chkliaev D.A., Nepomniaschy V.A. Deductive verification of the sliding window protocol // Modeling and Analysis of Information Systems. – 2012. – Vol. 19, N 6. – P. 57–68 (In Russian).

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

[6] Hartmanis J. Computational complexity of random access stored program machines // Mathematical Systems Theory. – 1971. – Vol. 5, N 3. – P. 232–245.

[7] Knuth D.E. The Art of Computer Programming (Vol. 1, 2, 3). – Addison-Wesley Publishing Company, 1968, 1969, 1973.

[8] Lambek J. How to program an infinite abacus // Mathematical Bulletin. – 1961. – Vol. 4, N 3. – P. 295–302.

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

[10] Owre S., Rushby J.M., Shankar N. PVS: a prototype verification system // Lect. Notes Comput. Sci. – 1992. – Vol. 607. – P. 748–752.

[11] Schonhage A. Storage modification machines // SIAM J. on Computing. – 1980. – Vol. 9, N 3. – P. 490–508.