Computers and Algorithms in the Mizar System

Artur Korniłowicz[1] and Christoph Schwarzweller[2]*

[1] Institute of Computer Science, University of Białystok, Białystok, Poland
arturk@math.uwb.edu.pl
[2] Department of Computer Science, University of Gdańsk, Gdańsk, Poland
schwarzw@math.univ.gda.pl

Abstract. We review the general model of a computer as defined in the Mizar system. The main emphasize is on specializing and using this model to formulate algorithms and prove their correctness. We give examples for a concrete machine over the integers derived from the general model. The further development of this model – especially towards the complexity of algorithms – is discussed.

1 Introduction

The development of the Mizar system [Miz04] started 30 years ago with the proposition of a formal language intended to match mathematical vernacular. The initial goal of this language – also called Mizar [Try78] – was not to write or check mathematical proofs but rather to assist mathematicians in writing and preparing their papers. Since then, however, Mizar has evolved to a proof checker [RT01] with one of the largest known repository of checked mathematical knowledge covering a variety of topics such as analysis, algebra, topology, graph theory, category theory, and others. The most ambiguous projects have been the formalization of a text book on continuous lattices [GHK+80] and the formalization of Jordan's curve theorem [Jor87].

Not widely known outside the Mizar community, however, are the efforts concerning programs and program verification in Mizar. Started already in the early 90th of the last century a theory of abstract computers has been developed and formalized in Mizar. The basic Mizar model of a computer [NT92] is quite general giving a conceptual framework in which computations take place. On top of this a number of so-called macro instructions resembling the structure of familiar procedural programming languages has been introduced and used to show the correctness of several algorithms, for e.g. of the Euclidean algorithm [TN93].

In this paper we give an overview of abstract computers and their algorithms as defined in Mizar. Based on the general abstract computer we discuss the possibilities of specializing this model to different – concrete and generic – machines. After that we concentrate on the **SCM-FSA** machine [TNR96], a "Simple Concrete Model" of a computer over the integers. We present the mechanism of

* The work of the second author was partially supported by grant BW 5100-5-0147-4.

macro instructions which allows to formulate algorithms for this machine in a programming language fashion. Finally we consider the question of describing and proving the complexity of algorithms. The basics of complexity theory have been formalized in Mizar [KRS01b], however not been applied to algorithms for abstract computers, yet.

2 The Mathematical Model of a CPU

The basic model of a computer has been defined in a quite general matter [NT92]. It does not resemble a special kind of programming language nor even a particular kind of machine. The idea was to capture the general structure of computations on a machine in this way providing a framework which can be instantiated to realize different models of computation. Consequently, the Mizar computer in fact models a central processing unit by introducing among others the concept of instructions and instruction counter. This means that memory and memory management is explicitely addressed. Here is the definition of the structure `AMI-Struct` [NT92].

```
definition let N be set;
struct (1-sorted) AMI-Struct over N
  (# carrier -> set,
     Instruction-Counter -> Element of the carrier,
     Instruction-Locations -> Subset of the carrier,
     Instruction-Codes -> non empty set,
     Instructions -> non empty Subset of
                   [:the Instruction-Codes,((union N) \/ the carrier)*:],
     Object-Kind -> Function of the carrier,
                       N \/ {the Instructions,the Instruction-Locations},
     Execution -> Function of the Instructions,
              Funcs(product the Object-Kind, product the Object-Kind) #);
end;
```

The `carrier` plays the role of the memory and the set `N` gives the data that can be stored within it. Note that both the instruction counter and the locations of instructions are placed in the same memory. The functor `Object-Kind` indicates which kind of information – about data or instructions – can be stored in a given memory location. A state of the computer is thus a function from the memory – the `carrier` – which respects these kind limitations, hence an element of `product Object-Kind`. Finally, an instruction is given by an instruction code and a data or memory element. Thus for example [J,<*a*>] can be read as the instruction "jump to a", that is the instruction counter is set to a. The interpretation of an instruction, however, is not fixed. Its meaning is given by the functor `Execution` which maps each instruction to a function from states to states.

This definition just establishes the basic concepts of a central processing unit, thus does not give a machine with usually expected properties. Such properties – if desired – are enforced in a second step using Mizar attributes. A natural

restriction, for example, is that only instruction locations can be put into the instruction counter, which gives a *von-Neumann*-like machine. Note, for a second example, that in principle the execution of an instruction can change the given program by overwriting the value of an instruction location. A machine, in which this does not occur, is called *steady programmed*.

Note also that the memory is an unordered set, which means in particular that it is not possible to speak of the k-th instruction (location) of a machine. To overcome this, an ordering of instruction locations has been defined in [TRK01]. The ordering is given by the sequence of instruction locations that can be put into the instruction counter during its execution. A machine such that there exists a bijection between the natural numbers and the instruction locations respecting this order is called a *standard* machine.

In addition the parameterization by the set `N` allows to consider special purpose machines: `N` gives the data that can be stored, so one could for example define a machine storing integer numbers or finite sequences of integers (the **SCM**, **SCM-FSA** computer respectively, see [NT93,TNR96]). Note that by instantiating the parameter `N` also operations of the data become available, in our example addition of integer numbers or concatenating finite sequences. These operations can be considered as the primitives of the machine, and be used when defining the functor `Execution`, that is the effect of the instructions. It is even possible to define generic machines that work over arbitrary algebraic structures: `N` is instantiated with the (algebraic) structure – in Mizar realized by structure definitions. Then the elements of the structure become the data and the structure's operations become the machine's primitives. So for example a machine **SCM**-Ring working over arbitrary rings has been defined in [Kor98].

Programming such a machine is not an easy task, basically it is the same as programming in an assembler language. So an obvious goal is to extend the model by introducing higher-level programming constructs that resemble ordinary programming languages. This has been done for the **SCM-FSA**-machine [TNR96], an extension of the **SCM**-computer over the integers, which in addition allows to store finite sequences of integers. Therefore in the following only this machine is considered.

3 Macros

To facilitate writing of programs and algorithms and making them more readable (more similar to programs written in languages like Pascal, C, etc.) a number of macros have been introduced in MML. They allow writing of programs in terms of Mizar terms. In this section we present two different definitions of macros and discuss the impact of these definitions on composing the macros.

3.1 General Definition

As a very basic definition of a macro one can take the finite partial state of a computer such that the domain of it is a subset of instruction locations. In fact,

it has been done so in MML, but one more condition has been added. Because it was done for **SCM-FSA** [TNA97], and because the instruction locations of **SCM-FSA** are constituted by natural odd numbers the authors decided to assume that a macro must be *initial*, in the sense that a macro must occupy the contiguous part of the memory starting from the first location. Having such a condition one can assume that, when one executes a macro, we always can (must) start the execution from the first instruction location. So, we do not put any conditions on the structure of the macro. Then, the composition of two macros (let's say f and g) can be done in the following steps [TN96a,TN96b]:

- incrementation of all jumps of g by the length of f,
- shifting of incremented g by the length of f,
- changing all halts of f by jumps to the first location of incremented and shifted g,
- concatenation of new f with new g.

3.2 More Specific Definition

From some point of view, the previous definition of a macro is not comfortable. Why? When one wants to compose two macros, one must replace all halts of the first macro by jumps to the beginning location of the second one. Therefore it was decided to introduce a new definition of a macro, which is more suitable for concatenation. Namely, this new definition additionally requires that a macro is finished by a *halt* instruction, and only one *halt* instruction is allowed in a macro, in contrast to the previous definition [TRK01]. Of course, this definition is more restrictive than the original one, but to compose two macros it is enough to remove the last instruction of the first macro and concatenate the second one, instead of changing all halts by jumps. Then, the composition of two macros (let's say f and g) can be done in the following steps [Kor01]:

- incrementation of all jumps of g by the length of f minus 1,
- shifting of incremented g by the length of f minus 1,
- removing of the last instruction of f,
- concatenation of new f with new g.

The shortness of the second way of composition is well visible in the case, when one concatenates macro instructions made of one "non halt" and one "halt" instruction. Table 1 presents the composition of four such macro instructions. Following the first way of concatenation we obtain as a result the macroinstruction containing 3 superfluous jumps. They do not appear in the case of the second definition. (In Table 1, the notion il.n stands for the n-th instruction location, and dl.n for the n-th data location.)

3.3 Examples of Macros

There is a number of macros already introduced in MML. Let us list some of them. We present macros defined only for **SCM-FSA** [Asa97a,Asa97b,Asa97c].

					general definition	specific definition
il.n	M_1	M_2	M_3	M_4	$M_1;M_2;M_3;M_4$	$M_1;M_2;M_3;M_4$
il.0	dl.1:=dl.0	dl.2:=dl.0	dl.3:=dl.0	dl.4:=dl.0	dl.1:=dl.0	dl.1:=dl.0
il.1	**halt**	**halt**	**halt**	**halt**	**goto** il.2	dl.2:=dl.0
il.2					dl.2:=dl.0	dl.3:=dl.0
il.3					**goto** il.4	dl.4:=dl.0
il.4					dl.3:=dl.0	**halt**
il.5					**goto** il.6	
il.6					dl.4:=dl.0	
il.7					**halt**	

Table 1. An example of the concatenation of macro instructions

Then, they are macros in the sense of the general definition, but not always in the sense of the second definition (not all of them are ended by *halt*).

1. Unconditional jump:

```
definition
 let l be Instruction-Location of SCM+FSA;
 func Goto l -> Macro-Instruction equals :: SCMFSA8A:def 2
  insloc 0 .--> goto l;
end;
```

2. Conditional macro:

```
definition
 let a be Int-Location;
 let I, J be Macro-Instruction;
 func if=0(a,I,J) -> Macro-Instruction equals :: SCMFSA8B:def 1
 a =0_goto insloc (card J + 3) ';' J ';' Goto insloc (card I + 1) ';'
     I ';' SCM+FSA-Stop;
end;
```

3. The below repeats a times I :

```
definition
 let a be Int-Location;
 let I be Macro-Instruction;
 func Times(a,I) -> Macro-Instruction equals :: SCMFSA8C:def 5
  if>0(a,loop if=0(a,Goto insloc 2,I ';'
                    SubFrom(a,intloc 0)),
     SCM+FSA-Stop);
end;
```

4 Algorithms in Mizar

Developing of the theory of computers, and especially the theory of macro instructions in MML, has been started and it is being still continued with deep confidence that such computers would be (are) suitable for verification of algorithms. A typical way of saying that an algorithm is correct relies on the comparison of the algorithm with a function that describes the semantics of the algorithm. The comparison should be done in terms of predicate `computes` [NT92], defined as:

```
definition
  let N be with_non-empty_elements set;
  let S be realistic halting IC-Ins-separated definite
        (non empty non void AMI-Struct over N);
  let p be FinPartState of S, F be Function;
  pred p computes F means :: AMI_1:def 29
  for x being set st x in dom F ex s being FinPartState of S st x = s &
   p +* s is pre-program of S & F.s c= Result(p +* s);
 end;
```

The function mentioned above is usually a partial function from the sets of all finite partial states to the sets of all finite partial states of the computer such that elements of the domain are related to inputs to the algorithm and elements of the codomain to results of the algorithm. What is important is that it is not necessary to define a function for each algorithm. If different algorithms solve exactly the same problem, it means that they compute the same function.

As an example let us take into account Euclid's algorithm defined and verified in [TN93]. The algorithm is just the assignment of appropriate instructions to the consecutive instruction locations.

```
definition
func Euclide-Algorithm -> programmed FinPartState of SCM
   equals :: AMI_4:def 1
   (il.0 .--> (dl.2 := dl.1)) +*
  ((il.1 .--> Divide(dl.0,dl.1)) +*
  ((il.2 .--> (dl.0 := dl.2)) +*
  ((il.3 .--> (dl.1 >0_goto il.0)) +*
   (il.4 .--> halt SCM))));
end;
```

Then the related function can be defined as:

```
definition
 func Euclide-Function -> PartFunc of FinPartSt SCM, FinPartSt SCM
    means :: AMI_4:def 2
  for p, q being FinPartState of SCM holds [p,q] in it
   iff ex x, y being Integer st x > 0 & y > 0 &
    p = (dl.0,dl.1) --> (x,y) & q = dl.0 .--> (x gcd y);
end;
```

It assigns partial states `q` of **SCM** that store the result (`x gcd y`) to relevant partial states `p` containing inputs (`x,y`).

A more advanced algorithm that has been already defined and verified using Mizar is bubble sorting of a finite sequence of numbers [CN98]. It involves macros in its body and looks like:

```
definition
  set a0 = intloc 0, a1 = intloc 1, a2 = intloc 2, a3 = intloc 3;
  set a4 = intloc 4, a5 = intloc 5, a6 = intloc 6;
  set initializeWorkMem =
      (a2:=a0) ';' (a3:=a0) ';' (a4:=a0) ';' (a5:=a0) ';' (a6:=a0);
 let f be FinSeq-Location;
 func bubble-sort f -> Macro-Instruction equals :: SCMBSORT:def 1
   initializeWorkMem ';' (a1:=len f) ';'
   Times(a1, (a2:=a1) ';' SubFrom(a2,a0) ';' (a3:=len f) ';'
        Times(a2, (a4:=a3) ';' SubFrom(a3,a0) ';'
             (a5:=(f,a3)) ';' (a6:=(f,a4)) ';' SubFrom(a6,a5) ';'
             if>0(a6, (a6:=(f,a4)) ';' ((f,a3):=a6) ';' ((f,a4):=a5),
                      SCM+FSA-Stop)));
end;
```

To verify the correctness of it the following function has been introduced:

```
definition
 func Sorting-Function -> PartFunc of FinPartSt SCM+FSA,FinPartSt SCM+FSA
     means :: SCMBSORT:def 3
  for p, q being FinPartState of SCM+FSA holds [p,q] in it
   iff ex t being FinSequence of INT, u being FinSequence of REAL
    st t,u are_fiberwise_equipotent & u is FinSequence of INT &
       u is non-increasing & p = fsloc 0 .--> t & q = fsloc 0 .--> u;
end;
```

It assigns partial states `q` containing sorted sequences `u` to partial states `p` containing original sequences `t`.

5 Outlook: Complexity of Algorithms

Formalizing the complexity of algorithms on a machine is a complex task. One needs a formalization of both the theoretical basis of complexity and a notion that connects formalized algorithms with this basis. The usual approach is to define step counting functions where often not all but only steps of main interest – e.g. multiplications or comparisons – are considered.

The beginnings of complexity theory has already been formalized in Mizar [KRS01a,KRS01b]. Here the basic notations such as domination and complexity classes are introduced. In addition a number of examples and problems from a textbook has been addressed. Thus [KRS01a,KRS01b] gives a framework for defining the complexity of algorithms. This framework, however, has not been

applied to describe the complexity of algorithms, yet. In the following we briefly outline how this can be done.

Mizar computers provide a number of possible "one-step" operations on states, from which programs, that is algorithms are built. It is therefore natural to consider sequences of execution steps to capture the length of a computation. In [NT92] the concept of a computation sequence – to identify halting computations – has been already introduced. It has been defined using a functor `Following` computing the immediate successor of a state `s`. Thus `Computation s` is the sequence of states the machine is running through when initialized with state `s`. Using this functor it is now straightforward to define a step function, that counts the number of executions of the machine beginning with a state `s` [BR93]. `CurInstr` gives the instruction of a state, that is the instruction to be executed next.

```
definition let N be with_non-empty_elements set;
 let S be halting IC-Ins-separated definite
                        (non empty non void AMI-Struct over N);
 let s be State of S such that s is halting;
 func Complexity s -> Nat means :: SCM_1:def 2
  CurInstr((Computation s).it) = halt S &
  for k being Nat
         st CurInstr((Computation s).k) = halt S holds it <= k;
 end;
```

Note that a computation is in principle infinite. Therefore the attribute `halting` saying that an instruction does not change a state is used to indicate the "end" of a computation. The functor can be easily modified to counting only the execution steps that involve primitive operations given by the parameter `N`. This means basically summing up the elements in `Computation s` with a certain instruction code. Analogously counting functions for only one operation such as multiplication or comparison can be constructed. Thus based on the general machine model defined in Mizar it is possible to define step-counting and complexity functions following the literature.

6 Conclusion

We have presented the general computing machine as defined in Mizar. The macro mechanism allows to use this machine to formulate algorithms and prove them correct by showing which function the algorithm computes. In principle, this approach works for arbitrary programs though it seems hard to apply it to algorithms occuring in everyday life. We believe, however, that this approach is suitable to formalize the principles of algorithms – as for examples presented in textbooks on this topic – in order to store in a mathematical repository.

The development of a complexity theory for algorithms as mentioned above is still in its beginning phase. The complexity function presented in [BR93], for example, only considers a single state `s`. To describe the complexity of algorithms

it should be generalized to a function from all states of a machine `S` – or some kind of input for `S`.[3] To summarize the approach formalized so far has still to be extended and worked out in order to become suitable for proving the complexity of algorithms.

References

[Asa97a] N. Asamoto, Conditional branch macro instructions of SCM+FSA, Part I (preliminary); Formalized Mathematics, 6(1), pp. 65–72, 1997.

[Asa97b] N. Asamoto, Conditional branch macro instructions of SCM+FSA, Part II; Formalized Mathematics, 6(1), pp. 73–80, 1997.

[Asa97c] N. Asamoto, The loop and times macroinstruction for SCM+FSA; Formalized Mathematics, 6(4), pp. 483–497, 1997.

[BR93] G. Bancerek and P. Rudnicki, Development of Terminology for SCM; Formalized Mathematics, 4(1), pp. 61–67, 1993.

[GHK+80] G. Gierz, K.H. Hofmann, K. Keimel and J.D. Lawson, M. Mislove and D.S. Scott, A Compendium of Continuous Lattices, Springer-Verlag, 1980.

[CN98] J. Chen and Y. Nakamura, Bubble Sort on SCM+FSA; Formalized Mathematics, 7(1), pp. 153–161, 1998.

[Jor87] C. Jordan, Cours d'Analyse de l'École Polytechnique, 1887.

[Kor98] A. Korniłowicz, The Construction of SCM over Ring; Formalized Mathematics, 7(2), pp. 295–300, 1998.

[Kor01] A. Korniłowicz, On the Composition of Macro Instructions of Standard Computers; Formalized Mathematics, 9(2), pp. 303–316, 2001.

[KRS01a] R. Krueger, P. Rudnicki and P. Shelley, Asymptotic notation. Part I: Theory; Formalized Mathematics, 9(1), pp. 135–142, 2001.

[KRS01b] R. Krueger, P. Rudnicki and P. Shelley, Asymptotic notation. Part II: Examples and Problems; Formalized Mathematics, 9(1), pp. 143–154, 2001.

[Miz04] The Mizar Home Page, `http://mizar.org`.

[NT92] Y. Nakamura and A. Trybulec, A Mathematical Model of CPU; Formalized Mathematics, 3(2), pp. 151–160, 1992.

[NT93] Y. Nakamura and A. Trybulec, Some Remarks on Simple Concrete Model of Computer; Formalized Mathematics, 4(1), pp. 51–56, 1993.

[RT01] P. Rudnicki and A. Trybulec, Mathematical Knowledge Management in Mizar; in: B. Buchberger, O. Caprotti (eds.), Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), Linz, Austria, 2001.

[TN96a] A. Trybulec and Y. Nakamura, Modifying addresses of instructions of SCM+FSA; Formalized Mathematics, 5(4), pp. 571–576, 1996.

[TN96b] A. Trybulec and Y. Nakamura, Relocability for SCM+FSA; Formalized Mathematics, 5(4), pp. 583–586, 1996.

[TN93] A. Trybulec and Y. Nakamura, Euclid Algorithm; Formalized Mathematics, 4(1), pp. 57–60, 1993.

[TNA97] A. Trybulec, Y. Nakamura and N. Asamoto, On the compositions of macro instructions; Formalized Mathematics, 6(1), pp. 21–27, 1997.

[TNR96] A. Trybulec, Y. Nakamura, and P. Rudnicki, The SCM+FSA computer; Formalized Mathematics, 5(4), pp. 519–528, 1996.

[3] This has been pointed out by one of the referees.

[TRK01] A. Trybulec , P. Rudnicki, and A. Korniłowicz, Standard Ordering of Instruction Locations; Formalized Mathematics, 9(2), pp. 291–301, 2001.

[Try78] A. Trybulec, The Mizar-QC/6000 Logic Information Language, ALLC Bulletin, Vol.6, No 2, 1978.