A FUNCTIONAL ABSTRACT MACHINE FOR IMPLEMENTATION OF FUNCTIONAL PROGRAMMING LANGUAGES
PART I


KEYWORDS: Functional Programming, Programming Languages, Operational Semantics, Denotational Semantics

ABSTRACT
A formal system of describing a functional abstract machine used in the implementation of a purely functional programming language A_LispKit Lisp is described. The basis for this was a SECD machine [La64, St83a, St83b], which has been used in the implementation of various versions of functional programming languages [Sto84, St84, He80, BuIv89, BuIv90a, BuIv90b]. This machine is extended with some new possibilities such as:
- the different method of parameter passing mechanism based on demand evaluation,
- incorporating of some primitives for interactive input / output, and for reading and writing in files, and
- extending the machine for the usage of the external and resident functional libraries.

The approach based on denotational semantics for formal description of this new machine is presented. In Part I of this paper am given the mathematical background of this approach. Also, we give some examples of the operational semantics approach in the definition of an abstract functional machine.

1. INTRODUCTION
For those who are less familiar with functional programming and functional programming languages in general, some elementary notations and definitions will be given.

1.1. FUNCTIONAL PROGRAMMING LANGUAGES
Functional programming languages are special kind of applicative programming languages which are based on the mathematical definition of functions. The name - functional, is used because in these programming languages a function (definition and application) is dominant.

1This research was supported by Science Fund of Serbia
like an assignment statement in conventional - imperative programming languages.

Functional programming languages consist of four components: set of data, function call - applicative operation, set of primitive functions, or built-in functions, and set of functional forms.

The set of data for earlier versions of functional programming languages such as LISP [McC60] is very restrictive, but the structure of data is simple and regular in mathematical sense. Modern functional programming languages such as KRC, Miranda [Tu79, PaJo87], etc. overcome this disadvantage, and put into them various kinds of data and data structures. Function call is a mechanism for applying function definition to the arguments and evaluating the result. The essence of functional programming is to perform two operations:
- to combine the function definitions with functional forms, and
- to compose functions with the purpose of getting new - powerful functions.

In addition to four components described above, functional programming languages have the mechanism for corresponding function name with definition, which means that the function is defined only once, not each time when it is called.

A pure functional language is one which has no assignment statement, it is free from side effects, and is referentially transparent. Each purely functional programming language is essentially a version of the λ-calculus [Ch41] covered in syntactic sugar. Usually, the λ-calculus kernel is enriched in various ways, e.g. by introduction of local definitions for the purpose of a better understanding, shortness of programs and faster execution, or incorporating of some forms of polymorphic type discipline.

1.2. STRUCTURE OF A FUNCTIONAL PROGRAMMING LANGUAGES AND DATA

One of the possible characterizations of a functional programming language is that it has simple and regular data structure, and that the program and data have the same structure, which is a property of machine languages only. The structure of the program, data and of the result is a symbolic expression (S-expression), which has a simple definition:
- An atom is S-expression, and
- If $S_1$ and $S_2$ are S-expressions then a pair $(S_1,S_2)$ is S-expression.

An atom is a set of words from some alphabet. Some chosen words from this set are the numbers or numeric atoms. There are three special atoms: $T$ for truth value, $F$ for false and NIL for an empty pair or list. A list is a special pair which is defined with:
- An atom is not a list,
- If $S_1$ is a pair which is not a list, and $S_2$ is an S-expression, then $(S_1,S_2)$ is not a list, and
- If $S_1$ is an atom or a list, and $S_2$ is an atom which is not NIL or a pair which is not a list, then a pair $(S_1,S_2)$ is not a list.

A program in some functional programming language is a function $Fun$, which is applied on some arguments $Args$, i.e.:

$$Fun \in _, Fun : _ \times _ \times \ldots \times _ \rightarrow _.$$

where the set $\_$ represents a set of all valid functions in that language, and $\_$ denotes a set of all well formed values of that language, in our case $\_$ is a set of all well-formed S-expressions.

A remarkable property of almost all functional programming languages is that a program - function, and data have the same structure. That means, that the following statement holds:

$$\_ \subset \_$$

i.e., from the structure point of view a universe of all functions is in a subset of the set of all data.

2. MACHINE ARCHITECTURE FOR A FUNCTIONAL PROGRAMMING
An important property of a functional programming language (FPL) is the ease with which can be used to express its own translator in the same language. This translator could be either a compiler which translates a FPL into another language, while an interpreter for a particular function definition and particular arguments computes the value of that function by applying it to the arguments. For this reason, it is called an interpreter or interpreting function.

An interpreter for a variant of FPL can be written in the form of a function \( \text{Apply}(\text{Fun}, \text{Args}) \). Here, \( \text{Fun} \) is the S-expression representation of a function - program, and \( \text{Args} \) the representation of its list of arguments by an S-expression. Thus, we can take \( \text{Apply}(\text{Fun}, \text{Args}) \) to be a formal specification of semantics of FPL, since it determines for us the value of any program for any given set of arguments. The principal criticism of this approach is that the meaning of the language being defined depends on our understanding of the language used to write the interpreter. For example, suppose we wish to determine whether the actual parameters are evaluated before function entry or only when needed, i.e., are they called by value or by name. Suppose we consider the function defined by \( \text{Fun} = \lambda(x)\text{Mother} \). This function, if called with an actual parameter which is undefined, such as \( \text{Args} = \text{Tail}(\text{Father}) \), would be undefined in one case and yet return the value \( \text{Mother} \) in the other. The problem arises because the level of the defining language is too high and therefore it opens to many different interpretations.

A much better way to overcome this is to make a compiler for that FPL. Firstly, we must define an abstract machine which essentially operates a stack in the same way we used it to evaluate functional programs. Then, we define a translator - compiler, which translates the programs written in FPL into the programs which can be run on this machine.

2.1. VARIOUS MODELS OF FUNCTIONAL ABSTRACT MACHINES

In some implementations, the abstract machine called the SECD machine, has been used. It was invented by Landin [La64], and used in various implementations of functional programming languages [Sto84, St84, He80, BuIv89, BuIv90a, and BuIv91]. Some other authors used some different machines: FAM by Cardelli [Ca83], or Graph machine (G-machine) [Tu79, PaJo87].

The FAM is based on delayed substitution in which the function application is carried out not by constructing an instance of the body of the function, but rather by evaluating the body of the function in an environment in which the formal parameters are bound to their actual values. The environment (context) is a data structure which holds the values of all variables currently in the scope. If the result of evaluating the function is itself a function, then a closure is returned, which is a pair consisting of:
- the code of the function;
- the environment in which it should subsequently be executed.

This is the approach very similar to the SECD machine strategy, and FAM can be considered as an optimized SECD machine:

- The SECD machine code is often implemented by direct interpretation of the abstract machine code. The FAM has a more powerful abstract machine code, and is compiled to a target machine code (VAX).
- The SECD machine environment is often implemented as a linked list, and the closures as a pair of pointers to the code and to the environment. The FAM constructs closures as an (N+1)-tuple, in which the first element points to the code of the function, and the other N elements are the values of only those variables that occur free in the function definition.
- The SECD machine stack and dump are often implemented as a linked list. The FAM uses the target machine stacks, called **AS** (Argument Stack) and **RS** (Return Stack) respectively [Ca84].

The G-machine [PaJo87], shows a remarkable similarity and correspondence to the environment based machines, and the FAM machine technique of implementation of **FPL**. Having said this, there is a close correspondence between the FAM and the G-machine:

- The G-machine equivalent to a FAM closure is a piece of graph consisting of a supercombinator applied to a few arguments. The arguments give the values of the variables used in the supercombinator body. It is an easy consequence of the \( \lambda \)-lifting algorithm that all the extra arguments to a function produced by \( \lambda \)-lifting are used somewhere in the supercombinator body. This corresponds to the fact that FAM closures only contain variables which may be required in the function.

- Execution is stack based for much of the time. Arguments to the current function are found on the stack. The difference here is that the FAM may also access free variables in the environment, whereas supercombinators have no free variables.

- Arguments to be passed to a function are placed on the stack before calling the function. This is always the case in the FAM and the optimization of the G-machine [PaJo87] mean that it will be the case in the realization of a G-machine.

There are two major differences between the FAM and the G-machine:

- The FAM is not lazy, i.e. the method of parameters passing is not based on demand evaluation.

- The G-machine is simply an efficient implementation of graph reduction. Graph reduction is a much more natural model to support parallel execution, so that a parallel G-machine is probably much easier to build then a parallel FAM.

In this paper, we introduced a formal specification of a new machine, called **A_SECD** machine [Je92a, Je92b], which is the natural extension of the SECD machine. Some of supplementations were taken from the FAM’s realization, and the concept of demand evaluations were used from the model of a G-machine.

3. SEMANTIC DEFINITION OF **A_SECD** MACHINE

**A_SECD** machine can be defined as a general function \( \text{Exec} \) which takes a compiled version of a function \( \text{Fun} \), denoted with \( \text{Fun}^* \), and the S-expression representation of the arguments \( \text{Args} \). Thus, it produces an S-expression representation of the result of applying \( \text{Fun} \) to \( \text{Args} \). The formal definition of **A_SECD** machine, the function \( \text{Exec} \), given in terms of denotational semantics approach, is:

\[
\text{Exec} : \__{A\_SECD} \times \_ \rightarrow \_ \quad \text{and} \\
\text{Eval} [ \, \text{Exec}(\text{Fun}^* \, , \, \text{Args}) \, | \, \rho = \text{Eval}_{A\_SECD} [ \, \text{Fun}^* (\, \text{Args} \, ) \, ] \, ] \, \rho = \text{Res},
\]

where \( \text{Fun}^* \in \__{A\_SECD}, \, \text{Args} \in \_ \) and \( \text{Res} \in \_ \). The set \( \_ \) represents a set of all programs - functions of A_LispKit Lisp language, \( \_ \) is a set of all S-expressions, and the set \( \__{A\_SECD} \) is a set of all possible, executable, programs in the machine language of **A_SECD** machine. The general denotational function \( \text{Eval} \) is defined by:

\[
\text{Eval} : \_ \rightarrow \_,
\]

where \( \_ \) is a set of all expressions, and \( \_ \) is a set of all values of a language. The denotational function \( \text{Eval}_{A\_SECD} \) describes semantics of the **A_SECD** machine, and the complete definition of the function \( \text{Eval}_{A\_SECD} \) will be given in Part II of this paper. Now, we will give only the mappings of the function \( \text{Eval}_{A\_SECD} \), with:

\[
\text{Eval}_{A\_SECD} : \__{A\_SECD} \rightarrow \_.
\]

where \( \__{A\_SECD} \) is a set of all valid functions written in the machine languages of **A_SECD**
machine.

From the point of view of operation semantics, formal definition of A_SECD machine, given by the function $\text{Exec}$ is:

$$\text{Exec}( \text{Fun}^*, \text{Args}) = \text{Apply}( \text{Fun}, \text{Args})$$

$$\text{Compile}( \text{Fun}) = \text{Fun}^*,$$

where the function $\text{Compile}$ translates a source code of a program function $\text{Fun}$ into the machine language of A_SECD machine. That is, in some way the A_SECD machine, given the S-expression representations of the compiled function (a machine language program) and its arguments, executes the machine language program to compute the result of applying that function to these arguments.

After defining the machine, we should write a compiler. That is, we will define a function $\text{Compile}( \text{Fun})$ with the denotational property that:

$$\text{Compile} : \_ \rightarrow \_\text{A_SECD},$$

$$\text{Eval}[\text{Compile}( \text{Fun}) \circ \text{Args}] \rho = \text{Eval}[\text{Exec}( \text{Fun}^*, \text{Args})] \rho,$$

where $\text{Fun}^* \in \_\text{A_SECD}$, $\text{Fun} \in \_ \text{and} \text{Args} \in \_$, $\text{Res} \in \_$, and the operator $\circ$ is a function applying operator. The context $\rho$ represents some environment in which all bindings of variables to their values are performed.

In fact, both $\text{Exec}( \text{Fun}^*, \text{Args})$ and $\text{Compile}( \text{Fun})$, will be operationally given by a set of rules of mapping, rather than by usual kind of function definition. However, by turning the rules for $\text{Compile}( \text{Fun})$, into the program $\text{Prog}$ in purely functional language, we are then in a position to transliterate that into the FPL program, for which a function $\text{Compile}$ is defined by:

$$\text{Compile}(\text{Prog}) = \text{Compile}^*.$$  

The function $\text{Compile}^*$, is now a machine language program with the property that:

$$\text{Exec}(\text{Compile}^* ,\text{Fun}) = \text{Fun}^*, \forall \text{Fun} \in \_. $$

That means, if we defined A_SECD machine by the function $\text{Exec}( \text{Fun}^*, \text{Args})$ and by the compiler $\text{Compile}( \text{Fun})$ (that compiler is obtained in an arbitrary way), we could made the new and new compilers in the source language, without changing the implementation code of the either A_SECD machine simulator $\text{Exec}$, or the compiler $\text{Compile}$.

The function $\text{Exec}( \text{Fun}^*, \text{Args})$ is implemented in such a way that it operates a stack for the evaluation of function calls, much as the process described. Since the program $\text{Fun}^*$ is an S-expression and since the data with which it operates are S-expressions, the natural notation for expressing the state of this stack machine is the S-expression notation. Thus, if we wish to denote the stack in such a way that its top item is an S-expression of $X$ we will write $(X.s)$, where $s$ represents the remaining items.

The form of the function $\text{Exec}( \text{Fun}^*, \text{Args})$ is such that the values of $\text{Fun}^*$ and $\text{Args}$ are loaded into the registers of the A_SECD machine in order to initialize it. Then, by analyzing the program $\text{Fun}^*$, the A_SECD machine operates in such a way that the state of the machine is transformed according to the semantics of each machine instruction which is encountered in the program. Ultimately, the value of $\text{Exec}( \text{Fun}^*, \text{Args})$ is computed and because of the property similar the net-effect, is to be found on the top of one of the stacks. The net-effect property is the property that the sequence of statements derived from a well-formed expression (in any language) will, if executed, leave a single value on the top of the stack, and otherwise leave unchanged that part of the stack in use before these statements were executed. The single value thus pushed onto the stack will be the value computed for the well-formed expression.

Strictly speaking, a pure stack is a data structure which has only two operations, the pushing of a new element onto the stack, and conversely, the operation of popping an element from the stack. It is said to be used in last-in-first-out discipline. The detonational semantics of a stack is:

$$\text{Stack} : \_ \rightarrow \_.$$
\[
\text{Stack}_{\text{Pop}} : _{-} \rightarrow _{-}, \\
\text{Stack}_{\text{Push}} : _{-} \times _{-} \rightarrow _{-}, \\
\text{Eval} [\text{Stack}_{\text{Pop}}((X,s))] \rho = \text{Eval} [X] \rho, \\
\text{Eval} [\text{Stack}_{\text{Push}}(X, (Y,s))] \rho = \text{Eval} [(X.(Y,s))] \rho,
\]
for (\ \forall X \in _{-}), (\ \forall Y \in _{-}) and (\ \forall s \in _{-}).

### 3.1. THE OPERATIONAL SEMANTICS OF A_SECD MACHINE

The A_SECD machine consists of five registers and each of them holds an S-expression. These registers derive their names from the purpose they have in dealing with S-expressions:

- **S** - the stack, used to hold the intermediate results during computation. At the end of the program execution, the top of the stack S contains the final result,
- **E** - the environment, holds the values which are bound to variables during evaluation,
- **C** - the control list, used to hold the machine-language program which is currently executed. In each moment of the evaluation process, the first element of the control list is the command which will be processed next,
- **D** - the dump, which saves the values of all other registers S, E, and C during a new function call,
- **L** - the resident library

manager, the stack which contains the resident libraries, i.e. the programs in an executable code written in the machine language of A_SECD machine, which are consulted during the evaluation of a program.

The machine language of the A_SECD machine consists of a certain number of commands. The execution of a command forces the machine to change its state, i.e. the contents of its registers. We call this a **machine transition**, and it can be denoted, from the point of operational semantics, in the following way:

\[
S \ E \ C \ D \ L \Rightarrow S' \ E' \ C' \ D' \ L',
\]

where S, E, C, D and L are the contexts of the registers before the next command execution, and S', E', C', D', and L' denote the new contexts of the all registers after that.

The initial state of the machine is:

\[
S \equiv (\text{Args}) the \ input \ data \ for \ program, \\
E \equiv () the \ empty \ list, \\
C \equiv \text{Fun}^* \text{A_SECD machine language program which will be evaluated}, \\
D \equiv () the \ empty \ list, \ and \\
L \equiv @\text{Lib}_1 \oplus \ldots \oplus @\text{Lib}_n \ \text{the \ list \ which \ is \ concatenation \ of \ resident \ libraries} \quad @\text{Lib}_i, \ 0 \leq i \leq n, \ n \in \mathbb{N}.
\]

For example, a machine transition of arithmetic operations of A_SECD machine is:

\[
(a \ b.S) \ E \quad (\text{OpA}.C) \ D \ L \Rightarrow (b \ \text{SiA} \ a.S) \ E \ C \ D \ L
\]

where OpA \in \{ \text{ADD, SUB, MUL, DIV} \ldots \} and SiA \in \{ \oplus, \ _, \ \odot, \ _, \ldots \}.

For another example, a machine transition of relation between the data of A_SECD machine is:

\[
(a \ b.S) \ E \quad (\text{ReA}.C) \ D \ L \Rightarrow (b \ \text{RiA} \ a.S) \ E \ C \ D \ L
\]
where \( \text{ReA} \in \{ \text{GT, GE, LE, NE} \ldots \} \) and \( \text{SiA} \in \{ >, \geq, \leq, \neq, \ldots \} \).

As all registers of A_SECD machine, according to the rules of machine transition, perform some operations on S-expressions, the simulator of A_SECD machine is naturally implemented by mapping these rules in some procedures of the implementation language.

On the other hand, the meaning of the instructions of A_SECD machine, in a mathematical sense is not clear. A much better way to describing the meaning of A_SECD machine, i.e. of its semantics, is to use denotational semantics approach.

4. CONCLUSION

The semantics of an abstract functional machine which is used in supporting the implementation of a functional programming language, can be described in two ways:
- using an operational approach, which is a convenient way for describing the implementation, or
- denotational approach, which is more suitable from the rigorous mathematical viewpoint of language definition.

The denotational approach in the definition of semantics of an abstract functional machine will be described in Part II of this article.

REFERENCES


