Chapter 6

**PFPM – AN ABSTRACT MACHINE**

6.1 Introduction

The parallel algorithms developed in Chapter 4 are being modified to fit into Parallel Architectures especially for SIMD architectures in Chapter 5. To implement the algorithms of Chapter-4 and Chapter-5 an abstract machine is developed in this chapter the method evolved in a functional programming machine is taken and a data-parallel evaluation mechanisms is added. This sat along side its ordinary sequential mechanism. The development involves Translation of a functional program expressed in an intermediate format into a sequence of low-level instruction and extension of same to SIMD architecture. The basic structure of a typical functional language compiler is illustrated in fig. 6.1

![Diagram](image)

**Fig. 6.1**
Chapter 6. PFPM – An Abstract Machine

Although figure 6.1 implies that each intermediate form exists explicitly during the computation of a program, in practice this may be relaxed. The second intermediate form can be generated directly from the abstract syntax tree, or even both intermediate forms can be omitted and the machine code can be generated directly. The final output of a compiler is a sequence of machine coded instructions which are represented as binary code inside the main memory of a target computer and so we could write ‘real machines’ in fig. 6.1 in place of ‘abstract machine’, but by that the compiler is tailored to one specific machine and not easily ported onto other machine. For this reason we talk about abstract machines rather than concrete machine and try to define an abstract machine architecture and instruction set which is sufficiently general to enable the same compiler to be used on a variety of target machines. This requires an extra layer of translation, which converts the individual abstract machine instructions into concrete machine instruction.

6.2 The PFPM System:

PFPM is an acronym for Parallel Functional Programming Machine. This abstract machine is POD based and can be viewed as an expanded version of FPM stack-based machine. The FPM will have a code generator which translates programs expressed in an intermediate form called FC into FPM
abstract Machine instructions (F Code). Source programs are translated into FC by first removing pattern matching and then Lambda lifting each user defined function to produce a set of combinators. The complete compilation route is shown in fig. 6.2

![Diagram of compilation process]

**Fig. 6.2**

### 6.2.1 The intermediate code FC:
An FC is represented as an S-expression.

\[
\begin{align*}
\text{preogram} & = ((\text{comblist} (\exp)) \\
\text{comblist} & = ((\text{combdef})^*) \\
\text{combdef} & = ((\text{argcount})(\exp)) \\
\exp & = (c(v(\con))(i(v(\argnum)))(m(t(\exp))^*)) \\
& \quad (t(v(\anty)(\exp))(i(f(\exp)(\exp)^*)) \\
& \quad (a(f(\funTAG)(\exp)^*)) \\
\con & = (i(n(\text{ger}))(\text{character}))(\conlist)(\contup) \\
\conlist & = ((\con)^*) \\
\contup & = [(\con)^*]
\end{align*}
\]
Chapter 6. PFPM – An Abstract Machine

(funtag) = (bi(code))(ud(funnum))(fe(exp))

(argcount),(argnum),(anty),(code),(funnum),(tag)

= integers 0,1,2,...

6.2.2 Constants(cv)

FC constants are either atoms, lists of constants represented as S-expressions or tuples of constants represented as S-expressions delimited by square brackets.

6.2.3 Local values(lv)

The expression(lv n) refers to the nth parameter of the combinator in which the expression occurs. For example, for the source function.

---f(x,y) <= .... X....y....;

the equivalent FC combinator will have an arity of 2 and x and y will be referred to by (lv 1) and (lv 2) respectively.

6.2.4 Temporary values(tv)

Tv expressions are used to introduce new local values and correspond to let statements.

For example, the FC expression

(tv k E1E2)

is equivalent to the expression

let (x1, x2, ... xk) = E1 in E2

153
Chapter 6. PFPM – An Abstract Machine

6.2.5 Conditional (if)

Conditional in FC are *multibranche* conditionals, meaning that a single conditional expression can have an arbitrary number of outcomes, rather than just two. For a conditional expression.

\[(\text{if } S \ E_1 \ E_2 \ldots \ E_n)\]

The selector expression $S$ is first evaluated to yield an integer which must be in the range $0$ to $n-1$. If the result of evaluating $S$ is $k$, then the value of the expression is the value of $E_{k+1}$.

6.2.6 Apply function (af)

Function applications have the following format

\[
\begin{align*}
\text{(bi } c) \\
\text{(af(udm) } E_1 E_2 \ldots E_n) \\
\text{(fe } E)
\end{align*}
\]

$E_1 E_2 \ldots E_n$ denote argument expressions to which the function in the second field is applied. If the function is built in (bi) then $c$ denotes the function code. The built in functions of FC are given in table 6.1. If the function is user-defined (ud) then $m$ denotes the $m$th combinator in the list of combinators defined in the program. If the function is neither built in nor user defined, then the expression $E$ denotes a function-valued expression (fe) which must always evaluate to a closure. A closure is generated as the
result of partially applying a function, that is by applying a function to fewer arguments
than its purity.

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Arity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>plus</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mult</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>greater</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>empty</td>
<td>0</td>
<td>Returns the empty (nil) list</td>
</tr>
<tr>
<td>5</td>
<td>cons</td>
<td>2</td>
<td>List constructor function</td>
</tr>
<tr>
<td>6</td>
<td>head</td>
<td>1</td>
<td>Returns the tail of a given list</td>
</tr>
<tr>
<td>7</td>
<td>tail</td>
<td>1</td>
<td>Returns the tail of a given list</td>
</tr>
<tr>
<td>8</td>
<td>isempty</td>
<td>1</td>
<td>Returns true(1) if given list is empty; false(0) otherwise</td>
</tr>
<tr>
<td>9</td>
<td>index</td>
<td>2</td>
<td>Returns a tuple component. The first parameter is the tuple; the second is the index of the component</td>
</tr>
</tbody>
</table>

Table 6.1 FC built-in functions.

The rules for translating from the standard intermediate code into FC are given in
table 6.2 Cp denotes the Fc built in function code for the primitive P, N_F denotes the
number of the combinator corresponding to F and L_x denotes the local value number
associated with the identifier x.
# Summary of intermediate code–FC translation rules

<table>
<thead>
<tr>
<th>Intermediate code expression</th>
<th>Equivalent FC expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>(cv C')</td>
</tr>
<tr>
<td>x</td>
<td>(lv L_x)</td>
</tr>
<tr>
<td>$\lambda x_1. \lambda x_2 \ldots \lambda x_n. E$</td>
<td>lambda-lifted to FC combinator</td>
</tr>
<tr>
<td>let x=e1 in E2</td>
<td>(tv [E1'E2'])</td>
</tr>
<tr>
<td>letrec x₁ = E₁,x₂=E₂,...,xₙ=Eₙ in E</td>
<td>E'</td>
</tr>
<tr>
<td>TUPLE.n E₁, E₂,...Eₙ</td>
<td>(mt E₁'E₂'...Eₙ')</td>
</tr>
<tr>
<td>IF E₁ E₂ E₃</td>
<td>(mt E₁'E₂'E₃')</td>
</tr>
<tr>
<td>P E₁E₂...Eₙ</td>
<td>(af (bi C P) E₁'E₂'...Eₙ')</td>
</tr>
<tr>
<td>(not TUPLE.n)</td>
<td></td>
</tr>
<tr>
<td>F E₁ E₂...Eₙ</td>
<td>(af (ud Nₚ) E₁'E₂'...Eₙ')</td>
</tr>
<tr>
<td>E E₁E₂...Eₙ</td>
<td>(af(fe E') E₁'E₂'...Eₙ)</td>
</tr>
</tbody>
</table>
6.3 The FPM abstract Machine:

The FPM abstract machine has four components; a program store which contains the compiled code for each combinator, an evaluation stack which is need to hold the agreements to a combinator any temporary values introduced, a call stack which holds return addresses for a combinator and a heap which stores the data structures built by the program and any closures generated by partially applying a function.

A code generator for FC expressions which produces sequences of F-code instructions is generated by using instructions given in table 6.2 this is a stripped down version of full F -code, this subset may be called ‘SIMPLE’

6.3.1 SIMPLE INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>#N</td>
<td>Pushes integer literal N onto the evaluation stack</td>
</tr>
<tr>
<td></td>
<td>$C</td>
<td>Pushes character literal C onto the evaluation stack</td>
</tr>
<tr>
<td></td>
<td>@A</td>
<td>Pushes address A onto the evaluation stack</td>
</tr>
<tr>
<td></td>
<td>%N</td>
<td>Pushes value N words from top of stack onto stack</td>
</tr>
<tr>
<td></td>
<td>*n</td>
<td>Pushes the nth entry in the table of constants onto the evaluation stack</td>
</tr>
<tr>
<td>TABLE</td>
<td>k</td>
<td>Pops the stack and puts the results in position k of the Constants table</td>
</tr>
<tr>
<td>COPY</td>
<td>n</td>
<td>Builds a heap cell by copying the top n elements of the stack onto the heap together with the integer n+1 which</td>
</tr>
</tbody>
</table>
forms the size field of the cell. A pointer to the heap cell replaces the items on the stack.

**DROP** \( m,n \) Moves the \( m \) values on top of the stack down \( n \) words and updates the top of stack pointer accordingly.

**UNPACK** Unpacks the components of the tuple pointed to from the top of the stack which is popped and pushes them onto the stack. The number of elements in the tuple is held in the size field of the tuple.

**CASE** \( n \) Pops the stack and uses the value as an index into a table of addresses (a jump table) located after the CASE instruction. Instruction execution continues at the indexed address.

**JUMP** \( A \) Forces instruction execution to continue at address \( A \).

**CALL** \( A \) Calls function at address \( A \). (Return address is saved on the call stack)

**RET** Return from function call (resume address is popped from the call stack)

**APPLY** \( n \) Applies the closure pointed to by the top of stack to the \( n \) arguments beneath that pointer on the stack.

**ADD, SUB, EMPTY etc.** Built-in operations which take their argument(s) from the top of the stack. The result replaces the arguments on the stack.

---

**Table 6.3 SIMPLE instructions**
Chapter 6. PFPM — An Abstract Machine

6.3.2 Compilation Schemes

The bulk of the FC-SIMPLE compiler is described in terms of five compilation schemes D,E,F,C and S which respectively compile function definitions, expressions, function calls, constants and structured constants. The D compilation scheme compiles a user-defined function definition into a sequence of SIMPLE INSTRUCTIONS. The generated code must adhere to the function-calling conventions of FPM: when a function is entered the arguments to that function will always be at the top of the evaluation stack and the result computed by the function must always replace these arguments. To express what each of the compilation schemes does in a concise manner we shall use a semifunctional shorthand notation. To compile the kth user-defined function we write.

\[ D(\text{(ne)};k) = \text{LABEL FUN}_k : (\text{compiled body expression, e}); \text{DROP 1}, n, \text{RET} \]

(The double brackets are used to delimit syntactic objects,) D is a function which given an FC function definition and the function number, k, returns the compiled code for that function as a sequence of SIMPLE instructions (the; between each instruction can be construed as a form of append function on instruction sequences). The D rule states that to compile the kth function definition we first label the function code FUN_k; we then compile the body of the function using the E-scheme, which generates code that leaves the value computed by on top of the stack; we then generate code to overwrite the argument frame (the local environment) with this result using the DROP instruction and finally return to the caller of the function using RET.
Chapter 6. PFPM – An Abstract Machine

To compile the body expression we use the E-scheme. This is complicated slightly by the fact that the local environment in which the expression is being evaluated is accessed by indexing from the top of the evaluation stack. Whenever we push an item onto the stack (for instance, when we are building the local environment for a nested function call) the ‘distance’ from the top of the stack to the local environment frame increases by one word. For this reason we must parameterize E by d, which is the number of words between the top of stack and the base of the local environment (Figure 6.4(a)). Furthermore, the local environment may be extended by using a tv expression, and this may result in the local environment entries being scattered over the stack. For this reason we must also parameterize C by an index map which indicates the number of words between the base of the local environment and

![Diagram](image-url)
Chapter 6. PFPM – An Abstract Machine

frame and each local value; an example with three local values is shown in Figure 6.4 (b).
Whenever we add to the local environment using tv this map is updated to include the
offsets of the new local value(s) using the parameter d. If maps n local values then
$$\{++[k, m]\}$$ will be used to denote the extension of $$I$$ which also maps local value $$n+1$$ to the
offset $$m+1$$ (from the base of the local environment), local value $$n+2$$ to the offset $$m+2$$ and
so on up to local value $$n + k$$ which is mapped to the offset $$m + 2$$ and so on up to locala
value $$n + k$$ which is mapped to the offset $$m + k$$. Figure 5.4 (c) shown the state of 6.4c the
parameter map after two new local values have been added to the stack of Figure 6.4 (b).
It is important to understand that the parameter map shown is a compile time structure, not
a run-time structure.

When we invoke the E rule to compile the body of an expression we must initialize
the index map with the offsets of the initial local values (function arguments). Thus in the
initial index map, which we write $$I^k$$, local value $$I$$ has an offset of $$k-1$$ from the base of the
local environment frame (1 ≤ k). The full rule for D now becomes

$$D\{[(ne)]k} = \text{LABEL \ FUN}_{k}; E\{[e]|n-1\}; \text{DROP 1, n; RET}$$

There is one E rule for each type of FC expression. If the expression is a constant then we
use the C-scheme to compile it. If the constant is an atom then code is generated to push
the value of the constant onto the stack. Structured constants are lifted out and compiled
into initialization code which is executed before the code for the top-level expression is
entered i.e., before the mainstream program execution begins; this ensures
Chapter 6. PFPM – An Abstract Machine

that they are evaluated only once. The initialization code evaluates the constant and leaves
a pointer to it in a constants table which forms part of the program store – see Fig. 6.3. The
kth table entry points to the kth constant lifted from the program. The S-scheme which
generates this initialization code is described later on. Hence we have

\[ E || (cvk) || d || = C || k || \]

\[ C ||'c''\] || = \text{PUSH $c$} \]

\[ C ||n \ || = \text{PUSH # n (if n is an integer)} \]

\[ (C ||"abcd\ldots") || = C ||\{a"b"b"c"d\ldots\}II \]

\[ C ||(E_1 E_2 E_n) || = C ||[E_1' E_2' E_n'] || \]

\[ = \text{(lift out the constant to be compiled later)}; \]

\[ \text{PUSH}^*k \ (\text{if this is the kth constant lifted}) \]

Notice that the final rule suggests the need for an additional parameter to \( C \) (the current
constant number, \( k \)) and additional result to be delivered by \( C \) (the list of constants to be
compiled later); however, these are omitted for clarity.

If the expression being compile under the \( E \)-s name is a local value reference then
we use the parameters \( d \) and \( l \) to compute the offset of the required local value from the top
of the stack:

\[ E ||(lv n) || d || = \text{PUSH} \% (d=1_n) \]

(the \( n \)th entry in the index map \( l \) is written \( n_l \)).

If the expression is an \( mt \) expression, then the tuple elements are first evaluated and
placed on the stack. This creates copies of the required tuple cell elements on the stack;
these are then transferred to the heap using the \text{COPY} instruction. Code is
Chapter 6. PFPM – An Abstract Machine

generated to check whether there is sufficient space to store the tuple and to add the size field to the tuple cell automatically if so. The elements on the stack are replaced by a pointer to the tuple in the heap and we have

\[ E||mt \ E_1 \ E_2 \ldots \ E_n|| d| = E||E_1|| d |;E||E_1|d+1|;;\ldots \]

\[ E||D_1||d+n-1|; \]

COPY \ n

is insufficient space on the heap to store the tuple then a garbage collector is invoked to reclaim any unused heap space.

If the expression is a \ lv \ expression then one or more new local values are added to the local environment; this involves updating | for the evaluation of the resultant expression as follows:

\[ E||(tv \ nE_1 \ E_2) || d | = E|| E_1|| d |; \]

If \ n > 1 \ then UNPACK;

\[ E||E_1|| d + n 1++ [n,d]; DROP 1, n \]

Notice that if the number of local values being defined is greater than 1, then the result of evaluating \ E \ will be a tuple and so must be unpacked onto the stack using the UNPACK instruction. Because the original language is strongly typed we are certain that \ n \ matches the size field of the tuple. The arity test is performed by the compiler and is not part of the generated code.
Chapter 6. PFPM — An Abstract Machine

If the expression is a multibranch conditional, then we must generate code to
evaluate the predicate, build the jump table and then compile each ‘arm’ of the conditional.
The last instruction in each arm (except the last arm) must be a JUMP instruction which
skips over any remaining arms to an ‘exit label’. To ensure that the label numbers
generated by the compiler are unique we must allocate sufficient label numbers to compile
the expression before beginning the compilation. In this way we can be sure that any
labels issued in the compilation of the code for each arm will also be unique. The
statement (Claim n) reserves n unique label numbers for this purpose. Labels have the
form a) (label number). We therefore define.

\[ E[(\text{if } P E_1 E_2 \ldots E_n)] \ |	ext{d} | = E \parallel P \parallel \text{d} |, \text{CASE } n; \]

(Claim \(n + 1\)) \hspace{1cm} (assume \(b, b + 1, \ldots, b + n\) allocated)

L\(_b\) + 1; ADDR L\(_b\)+2;\ldots;

ADDR L\(_b\) + n;

LABEL L\(_b\) + 1; E||E\(_1\)|| d |; JUMP L\(_b\);

LABEL L\(_b\) + 2; E||E\(_2\)|| d |; JUMP L\(_b\);\ldots;

LABEL L\(_b\) + n ;E||E\(_n\)|| d |;

LABEL L\(_b\)

Finally, if the expression is a function application, we first generate code to
evaluate all the function’s arguments and then generate code to call the function using the
A-scheme. If the function being applied is a closure then we generate an APPLY
instruction. If the number of arguments in the argument list is less than the arity of the
function, we form a closure by pushing the function address and the pending argument
count onto the stack and then using COPY to transfer the closure to the heap; otherwise we call the specified function. In order to test whether we must form a closure form the application. A must be parameterized by the number of arguments supplied to the function. We denote the arity of a function \( f \) by \( A \), and the instruction mnemonic corresponding to a built-in function \( c \) by \( M_c \). For example, \( M_1 \) delivers SUB and so on.

Hence we define.

\[
E \langle (af F \ E_1 \ E_2 \ldots \ E_n) \parallel d \parallel E \parallel E_n \parallel d \parallel E \parallel E_{n-1} \parallel d+1 \parallel \ldots; \\
E \parallel E_1 \parallel d + n - 1 \parallel; \\
A \parallel F \parallel d + n \parallel n
\]

\[
A \langle (feE) \parallel d \parallel n = E \parallel E \parallel d \parallel; \text{APPLY} \ n
\]

\[
A \langle (bi c) \parallel d \parallel n = \text{if} \ n = A_0
\]

Then \( M_0 \)

Else \( \text{PUSH}@B|c; \text{PUSH} \ A_c - n \); \( \text{COPY} \ n+2 \)

\[
A \langle (udk) \parallel d \parallel n = \text{if} \ n = A,
\]

Then \( \text{CALL} \ \text{FUN}_k \)

Else \( \text{PUSH}@\text{FUN}_k \); \( \text{PUSH}'A_k - n \); \( \text{COPY} \ n + 2 \)

If the function being partially applied is built in, the function address stored in the resulting closure is the address associated with the label \( B|c \) where is the numeric FC code for the built-in function. So, in addition to there being an instruction for each built-in function \( c \), there must also be a piece of code at label \( c \) which performs the same operation as that single instruction. For example.
These code sequences are the same for each program compiled and so may be viewed as a library of definitions. In fact they are stored in the kernel of the FPM system, which will be described later on. It is important to understand that these code sequences will only be referred to when a built-in function is partially applied; if the function is provided with all of its arguments at once then the single instruction associated with it will be invoked. This all follows from the fact that the function address field of a closure (i.e., \texttt{FUN@\texttt{must}} be an address; we cannot place an instruction in that field directly.

To complete the code generator we have only to define the S-scheme which generates (initialization) code to build a constant list or tuple and which leaves a pointer to it in the constants table. The initialization code is preceded by the constants table itself, the space for which is reserved by the \texttt{RESERVE} assembler directive. If a total of \(N\) constants is lifted from the program then the directive \texttt{RESERVE} \(N\) will be inserted in to the code sequence prior to the initialization code.

\(S\) is applied to each lifted constant in turn. To compile the \(k\)th constant, \(C_k\), we first generate code to build the constant and then place the pointer to the constant in the constants table at position \(k\);
Chapter 6. PFPM – An Abstract Machine

\[
S||c\|;\] TABLEk
\]

\(S\) is given by

\[
\begin{align*}
S||'c'|| &= \text{PUSH} \; S_c \\
S||n|| &= \text{PUSH} \#n \quad (n \text{ is an integer}) \\
(S||"abcd..."||) &= S||( 'a' 'b' 'c' 'd' )|| \\
S||[E_1...E_{n-1}; E_n]|| &= \text{EMPTY}; S||E_n||; \text{CONS}, \\
&S||E_{n-1}||; \text{CONS}; ... \\
&S||E_1||; \text{CONS} \\
S||[E_1 E_2...E_n]|| &= S||E_n||; S||E_{n-1}||; ...; S||E_1||; \text{COPY} \; n
\end{align*}
\]

(Note the \texttt{EMPTY} has the effect of pushing the empty list (nil) onto the stack.)

To compile the top-level expression we have only to invoke \(E\) suitably parameterized to indicate that the local environment for the top-level expression is empty;

There are clearly a great many optimizations which can be built into the code generator to improve the quality of the resulting code. Typically, these optimizations are concerned with minimizing data movement to and from the stack and rely on making the F-code instruction set orthogonal in order that certain function arguments can be compiled directly into instruction operands rather than into stack pushes. In this respect they are rather low-level optimizations more akin to peephole optimizations and are rather machine specific in that they assume certain properties of the underlying concrete machine instruction set. However, there are some rather higher-level optimizations which can be applied. One of the simpler ones, which is the subject of an exercise at the
Chapter 6. PFPM – An Abstract Machine

end of the chapter, is to avoid building redundant heap cells. This attempts to deduce when it is safe to leave a tuple’s components on the stack after the execution of an mt instruction.

Another commonly used optimization is one which handles tail-recursion. Recall that a tail-recursive function is one whose result (in the non-base case) is determined by a recursive call to itself as in

\[ \text{-----} f(x, a) = \begin{cases} \text{if } x = 0 \text{ then } a \text{ else } f(x - 1, a + x); \end{cases} \]

In fact the optimization is more general in that it can be applied regardless of whether the recursive call is to for some other function, \( g \) say.

The first thing we need to know is whether the expression being compiled constitutes the result of the function whose code we are in the process of generating. To this end we add an extra parameter to the E-scheme which is \( Y \) if this is the case and \( N \) otherwise. In the case of function applications this extra parameter must be inspected: for user-defined functions if the extra parameter is \( Y \) then a jump to the called function can be made instead of the usual call. Before this can be done, however, the arguments of the function being called must be dropped down over the arguments of the current function, which are no longer required. This prevents the stack from growing on each recursive call;

\[ \text{A||ud k)||d | ny = if } n = A_k \]

then \( \text{DROP } n, \ d - n + 1 \); \( \text{JUMP \ FUN}\_k \)

else \( \text{PUSH@FUN}\_k, \text{PUSH } A_k - 1 \),

\( \text{COPY } n + 2 \); \( \text{DROP } 1, \ d - n \)
Chapter 6. PFPM – An Abstract Machine

The extra parameter must be appropriately propagated through each of the compilation schemes. Each of the rules for \( E \) must ensure that when the parameter is \( Y \) the compiled code for the given expression replaces the current stack frame by the result and returns to the calling function. This obviates the need for the DROP and RET instructions in the rule for \( D \) which now looks like this:

\[
D \| (n e) \| k = \text{LABEL FUN}_k; E \| e \| n - 1 \| Y
\]

6.4 Extension to Parallel Architectures:

As the code for SISD and SIMD will be the same, the algorithms designed in Chapter 5 will take care about the implementation of Functional Programs on SIMD architecture. To implement the same on MIMD machines the data dependency techniques should be used. That part has not been included in this thesis.
6.5 Summary:

- Functional programs expressed in combinator form has been translated into code for an abstract Machine.

- The abstract machine PFPM can be viewed as an optimized version of the SECD Machine.

- Source Programs are first compiled into a low-level functional code called FC, then translated into FPM machine code using translation scheme developed.

- The abstract machine is implemented on any SIMD machine by interpreting the instructions as macron as applying the algorithms developed in Chapter 5.