REUSE ENVIRONMENT BASED ON A MULTILEVEL PROGRAMMING PARADIGM

By

GARY WAYNE SMITH

Bachelor of Science

Texas A&M University

College Station, Texas

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1990

REUSE ENVIRONMENT BASED ON A

MULTILEVEL PROGRAMMING

PARADIGM

Thesis Approved:

Thesis Advisor

Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Dr. K. M. George for introducing me to the FP style of programming and serving as my thesis advisor. His advice and guidance made this thesis possible. I would also like to thank Drs. M. Samadzadeh and G. E. Hedrick for their support, flexibility, and comments while serving on my thesis committee.

I would like to give a special thanks to Dr. J. P. Chandler. Dr. Chandler has provided invaluable support and encouragement thoughout my stay at OSU, especially after that first day of teaching the 2113 theory section.

Thanks go to Drs. George (again) and B. Mayfield for serving as ACM and programming contest advisors this past year. I could not have made it through my first elected office without their help and support.

Finally, I would like to thank my parents, Dr. and Mrs. James W. Smith, my sister Teresa, and especially my wife, Lisa. Lisa's near infinite spelling abilities were thesis saving.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	SOFTWARE REUSABILITY	3
	Types of Reuse	3
	Motivation and Inhibitors	6
III.	FUNCTIONAL PROGRAMMING - FP LANGUAGE	9
	Description of FP System	9
	Sample FP Functions	13
	Advantages and Limitations	13
	Illinois FP	14
IV.	MULTILEVEL PROGRAMMING PARADIGM	17
	Function Level	17
	Class Level	18
	Procedure Level	21
V.	REUSE ENVIRONMENT BASED ON MPP - REMPP	23
	Goals	23
	Function and Class Level Syntax	24
	Sample Function and Class Level Routines	26
	Description of Environment	27
	Implementation Details	30
VI.	SUMMARY, CONCLUSIONS, AND FUTURE WORK	35
	Summary and Conclusions	35
	Future Work	36
BIBLIOGRAPHY		38
APPENDIX - SAMPLE REMPP SESSION		40

CHAPTER I

INTRODUCTION

It is widely believed that software reusability shows great promise in increasing programmer productivity. Utilizing preexisting programs instead of rewriting them every time they are needed could in general lead to tremendous savings. Unfortunately, most programming systems and languages provide only limited support for large scale reusability.

In 1978, Backus [BACK78] introduced the FP programming language. FP incorporates a method of programming in which large functions are built from smaller functions using program forming operations. Backus' primary purpose in designing FP was to provide an alternative to the conventional von Neumann languages. FP also provides strong support for code reusability. FP functions are free of side effects. They could be used in any position in a program with a guaranteed behavior.

FP does have its limitations. Its primary limitation is that it is not history sensitive. In its original presentation, FP could not be used in interactive applications, such as text editors. In an attempt to extend the capabilities of FP as well as provide even stronger architectural support for reuse, George designed the Multilevel Programming Paradigm (MPP) [GEOR90].

MPP is a three-level system. The first level is based on standard FP. The second level adds an object oriented class system to FP functions. The third level is a general purpose procedural level used to extend the capabilities of FP.

The primary purpose of this study has been to develop an environment supporting MPP program development at all three levels and specialized support for the first two levels: function and class levels. The environment is called REMPP for Reuse Environment based on a Multilevel Programming Paradigm.

Chapter II contains a more detailed description of the effect of reusability on programmer productivity along with some of the inhibitors to reusability. Chapters III and IV are devoted to descriptions of FP and MPP, respectively. Chapter V contains a description on the REMPP environment and details of its implementation.

CHAPTER II

SOFTWARE REUSABILITY

The cost of software development is rapidly increasing With the rising cost, there is a pressing need for increased productivity from software engineers. In addition, more and more time and effort is spent on post production activities such as maintenance. Reusability has emerged as a practical way to improve both programmer productivity and software quality. These improvements can lead to lower development and maintenance costs.

Types of Reuse

The broad area of approaches to software reusability can be divided into three main areas: generation, translation, and composition. While different in nature, all of these methods can provide significant productivity and quality improvements.

In code generation techniques, the emphasis is on having the system produce the code required. A compiler generator is an example of a code generation tool. When using compiler generators, the programmer supplies specification of the software to be developed in the form of a grammar. The generator then produces the compiler which satisfies the specification. Reuse at this level is less of a matter of using preexisting components and more of a matter of

execution (of the generator) [BIRI87].

In approaches based on transformations, the software development environment provides facilities to apply transformations to existing software. This is done to produce software to meet new demands and to use in new application areas. Goguen's theory morphism is an example of this type of approach [GOGU86].

The other major type of reuse is code composition. In composition, the programmer is concerned with finding and using existing modules, functions, objects, etc. Depending on the amount of software artifacts reused, the main effort could be spent tying the existing routines together.

A classic example of code composition is that used in the UNIX[1] pipe scheme [KERN84]. Complex procedures can be developed by tying the output of a smaller procedure to the input of the next, much like a pipe. This type of composition is described as horizontal composition.

A more powerful method of composition is that of vertical composition. In vertical composition, higher levels of abstraction are used to tie parts together. There are two avenues in vertical abstraction: that of the traditional software engineering techniques (specifications, design) and that of recent object-oriented approaches [KAGA87].

The potential for reuse is higher with each higher level of abstraction. In the case of design reuse, however, there are several problems to overcome. Foremost is the lack of a design standard. Another is the fact that even if a design is reused, the lower level code may still need to be

[1] UNIX is a trademark of Bell Laboratories.

originally developed [KZSG90]. The best example of reusable design is that of compiler construction [AHSU86]. Most all compilers contain a lexical analyzer, parser, etc.

Object-oriented languages provide a class/subclass mechanism to facilitate the use of existing software as a base and the adding of new modules on top of them. The class systems in most object-oriented languages (C++, Smalltalk, etc.) and the generics in Ada provide powerful abstraction mechanisms. The Multilevel Programming Paradigm described in Chapter IV is also a system designed for strong class-based reusability support.

Side-effect-free functional programs show great promise for reusability. Mathematically-based functions can be combined to form larger functions. Also, mathematical techniques can be used for consistency checks and correctness proofs on these larger functions. This is an important issue when developing software for applications which prohibit real life testing.

In any software reuse method, a major concern is how to manage the components of existing software. This means identification and adaption problems need to be addressed. Libraries or other suitable means need to be provided. The programmer needs to have simple and effective access to the library. Methods need to be developed to aid the programmer in selecting the correct routine to use. A large well-maintained library is of no use if programmers do not know it exists, or if it is largely inaccessible. In addition, if the effort needed to adapt existing software is greater than the effort needed to create it, there is no justification for

reuse. The accessibility issue is partially addressed in this thesis.

Motivation and Inhibitors

If 40% of a design and 70% of the code on a project can be taken from reusable parts, the net development productivity would increase by about 40%. While this may not be an order of magnitude increase, it is substantial. In addition, even higher payoffs come later when maintenance costs could decrease by as much as 90% [BIRI87].

Mary Shaw is quoted in [AGMG89] as giving several reasons for these improvements:

- Effort: Putting existing routines together with relatively minor amounts of glue (parameters, inheritance, instantiation, etc.) is a much smaller task than developing the routines from scratch (the glue is still needed).
- Reliability: The components available for reuse are generally stable and well-documented. The more a routine is used, the more reliable it becomes. Each use involves some amount of testing, even if the testing is in the form of normal usage.
- Consistency: In most situations, all components in a library would be developed in a standard format. Using these same components throughout the system increases the chance of a consistent design.
- Manageability: Using several smaller well-understood and already well-documented components helps to decrease the chances of cost and schedule overruns.

Most of the parts have already been developed and their behavior is understood.

- Standardization: The components available for reuse would generally have been developed with reuse and standards in mind. These components could provide the standardization base upon which the rest of the program could be fashioned.

Reusability seems to be a highly appealing technique and tools supporting reusability have been available for several years. Unfortunately, the sweeping effects envisioned by reusability prophets have not been fully realized [TRAC87]. There are several factors which have retarded wide scale reusability. Some are listed below:

- "Not Invented Here" syndrome: Programmers have a tendency not to want to use other people's code. They may not trust the quality of the code, they may feel a sense of job security if they develop the code themselves, or they may find it more fun to write it themselves. Whatever the reason, the outcome is a tendency not to use reusable code, even if it is available.
- Cost: It takes a substantial effort to start a reusability system. The library system must be obtained or developed. All the reusable components in the library must be obtained or developed. Before code can be reused, it must be used in the first place. The library must be made available to a large enough group to have a positive return. This can take up valuable computing resources. These and other

startup costs can be significant. Short-sighted deadlines might not allow for the time and resources necessary.

- Lack of standards: There are no clearly defined standards, either for developing reusable software or for systems based on reusable software. While the lack of standardization may foster research, bottom-line managers are not as likely to invest in technology that is changing so rapidly, especially considering the possibility of high startup costs.
- Lack of tools. Tools currently exist for software reuse (libraries, object-oriented languages), but they are largely unavailable and limited. As the field of reusability matures, more and better tools need to be developed.

The activity in the reusability related research (including this study) is mostly devoted to solving technical problems associated with the access and integration of software artifacts [TRAC89]. This study is concerned with the development of a environment based on the MPP paradigm. MPP was designed to combine and integrate important reusability concepts. Much of the paradigm is adapted from Backus' FP language.

The next chapter contains a description of the FP language. FP in its original form shows considerable potential for reusability. With the further development of FP into MPP and the development of the REMPP environment, reusability is even further supported.

CHAPTER III

FUNCTIONAL PROGRAMMING - FP LANGUAGE

John Backus introduced the FP programming language [BACK78] as an alternative to conventional von Neumann programming languages in his Turing Award lecture. FP is based on mathematical foundations: FP programs can be proven correct by using the associated algebra of functional programs [BACK78, HAWW90, GEHE86]. FP utilizes a revolutionary variable-free style of programming. As there are no variables or states, FP functions are free from side effects.

Description of FP System

An FP system contains the following five parts:

a) A set of objects where each object is either an atom or a sequence: $\langle x_1, \ldots, x_n \rangle$ whose elements are objects, or undefined ("bottom", represented by $\perp$). Examples of atoms are: numbers, strings, booleans T or F, and $\perp$. The atom ϕ is used to denote the empty sequence and is the only object that is both an atom and a sequence.

b) A set of primitive functions that are built into the FP system.

c) A set of functional forms used to tie functions

together.

d) A set of definitions that associate names to functions. A definition is of the form:

def l ≡ r, where the left side (l) is an unused function symbol and the right side (r) is a possibly recursive expression which is built from other functions using functional forms.

e) An application operation denoted by ":".

Primitive Functions

The following examples of primitive functions are taken from [BACK78].

a) Selector functions (1,1r,2,2r,3,3r,...):

$i{:}x \equiv$ if $x = \langle x_1,\ldots,x_n\rangle$ and $i \leq n \rightarrow x_i$

else $\rightarrow \perp$

$ir{:}x \equiv$ if $x = \langle x_1,\ldots,x_n\rangle$ and $i \leq n \rightarrow x_{n-i+1}$

else $\rightarrow \perp$

b) Tail functions:

$tl{:}x \equiv$ if $x = \langle x_1\rangle \rightarrow \phi$

else if $x = \langle x_1,\ldots,x_n\rangle \rightarrow \langle x_2,\ldots,x_n\rangle$

else $\rightarrow \perp$

$tlr{:}x \equiv$ if $x = \langle x_1\rangle \rightarrow \phi$

else if $x = \langle x_1,\ldots,x_n\rangle \rightarrow \langle x_1,\ldots,x_{n-1}\rangle$

else $\rightarrow \perp$

c) Identity function:

$id{:}x \equiv \rightarrow x$

d) Equals function:

$eq{:}x \equiv$ if $x = \langle y,z\rangle$ & $y = z \rightarrow T$

else if $x = \langle y,z\rangle$ & $y \neq z \rightarrow F$

else $\rightarrow \perp$

e) Null function:

null:x $\equiv$ if x = $\phi \rightarrow$ T

else if x $\neq \perp \rightarrow$ F

else $\rightarrow \perp$

f) Add, Subtract, Multiply, Divide functions (+,-,*,÷):

+:x $\equiv$ if x = <y,z> & y,z are numbers $\rightarrow$ y + z

else $\rightarrow \perp$

g) And, Or, Not functions:

and:x $\equiv$ if x = <T,T> $\rightarrow$ T

else if x = <T,F> x = <F,T> x = <F,F> $\rightarrow$ F

else $\rightarrow \perp$

h) Append Left, Append Right functions:

appndl:x $\equiv$ if x = $\langle y,\phi\rangle \rightarrow \langle y\rangle$

else if x = $\langle y,\langle z_1,\ldots,z_n\rangle\rangle \rightarrow \langle y,z_1,\ldots,z_n\rangle$

else $\rightarrow \perp$

appndr:x $\equiv$ if x = $\langle\phi,z\rangle \rightarrow \langle z\rangle$

else if x = $\langle\langle y_1,\ldots,y_n\rangle,z\rangle \rightarrow \langle y_1,\ldots,y_n,z\rangle$

else $\rightarrow \perp$

i) Transpose function:

trans:x $\equiv$ if x = $\langle\phi, \ldots ,\phi\rangle \rightarrow \phi$

else if x = $\langle x_1,\ldots,x_n\rangle \rightarrow \langle y_1,\ldots,y_m\rangle$

where $x_i = \langle x_{i1},\ldots,x_{im}\rangle$ and

$y_j = \langle y_{1j},\ldots,y_{nj}\rangle$, $1\leq i\leq n$, $1\leq j\leq m$

else $\rightarrow \perp$

Functional Forms

Some functional forms given in [BACK78] are described below:

a) Composition

$(f \cdot g):x \equiv f:(g:x)$

b) Construction

$[f_1,\ldots,f_n]:x \equiv \langle f_1:x,\ldots,f_n:x\rangle$

c) Condition

$(p \rightarrow f;\ g):x \equiv$ if $(p:x) = T \rightarrow f:x$

else if $(p:x) = F \rightarrow g:x$

else $\rightarrow \perp$

d) Constant

$x:y \equiv$ if $y = \perp \rightarrow \perp$

else $\rightarrow x$

e) Insert

$/f:x \equiv$ if $x = \langle x\rangle \rightarrow x$

else if $x = \langle x_1,\ldots,x_n\rangle$ & $n \geq 2$

$\rightarrow f:\langle x_1,/f:\langle x_2,\ldots,x_n\rangle\rangle$

else $\rightarrow \perp$

f) Apply to all

$\alpha:x \equiv$ if $x = \phi \rightarrow \phi$

else if $x = \langle x_1,\ldots,x_n\rangle \rightarrow \langle f:x_1,\ldots,f:x_n\rangle$

else $\rightarrow \perp$

g) While

$(\text{while } p\ f):x \equiv$ if $p:x = T \rightarrow (\text{while } p\ f):(f:x)$

else if $p:x = F \rightarrow x$

else $\rightarrow \perp$

The functional forms and primitive functions determine each particular FP system. One implementation may have a small set of primitive functions and functional forms. Other systems may have larger and more powerful sets.

Sample FP Functions

The following sample FP functions are indicative of the style of FP programming. These small functions can be combined recursively or with other primitive or defined functions to form larger functions.

a) Inner Product

def INNER ≡ (/+) ∘ (α*) ∘ trans

b) Length

def LENGTH ≡ null? → *0*; (/+) ∘ (α1)

c) Add1

def ADD1 ≡ +∘[id,*1*]

Advantages and Limitations

Each function in FP takes one object as its only argument and maps the object to another object. The lack of variables or state guarantees that functions are free from side effects. FP functions are combined hierarchically to form larger FP functions. These three factors: the simple interface, process of combining functions to form larger functions, and guarantee of no side effects all contribute to reusability.

Also associated with FP is its own algebra of functional programs [BACK78]. Each variable in the algebra of programs represents an FP function while the manipulation rules represent the FP combining forms. All FP functions can be reasoned about mathematically using the laws of algebra. This can be of invaluable aid in verifying that a function

will perform as expected.

Although FP represents a powerful programming environment based on mathematics, it is not well suited for general purpose software development. The most noted shortcoming is its inability to handle interactive programming. There have been several attempts at enhancing FP, including the addition of infinite objects [HAWW90] and introduction of objects with state [GEOR88]. Backus et al. introduced the FL programming language [BAWW86] as a general purpose extension of FP. FL is a substantially more complex language than FP [GEOR90]. An alternate approach was taken in the MPP system described in Chapter IV.

Illinois FP

There have been several architectural designs for FP based systems [MAMI84, HUHH86]. At least two implementations of FP interpreters for use on general purpose machines have been developed: Berkeley FP [BADE83] and Illinois FP [ROBI87a, ROBI87b]. Berkeley FP utilizes FP syntax and semantics while Illinois FP (IFP) uses FP semantics but introduces a more conventional syntax. IFP has been shown to be significantly faster than Berkeley FP [ROBI87a].

The IFP interpreter is a public domain programming system running under UNIX. The user must enter the IFP environment before running any functions. The user can organize IFP functions in a tree structure using the UNIX subdirectory scheme.

As previously stated, IFP incorporates FP semantics with a different syntax. The syntactic differences are in the

way the functional forms are constructed. All primitive functions described in the previous sections are incorporated into IFP. The IFP syntax was designed to facilitate indentation and comments. Parentheses are neither needed nor allowed since all functional forms bracket their parameters. IFP functional forms follow a left to right syntax instead of the right to left order of FP functions. A table showing the FP and corresponding IFP syntax follows [ROBI87b]:

FP	IFP
C∘B∘A	A | B| C
[F,G,H]	[F,G,H]
p → f;g	IF p THEN f ELSE g END
p → f; q → g; h	IF p THEN f ELSEIF q THEN g ELSE h END
αf	EACH f END
/f	INSERT f END
(while p f)	WHILE p DO f END
def f ≡ x	DEF f AS x;
ϕ	<>
⊥ (bottom)	?

The following IFP program corresponds to the FP inner product function given in the previous section:

```
DEF Inner AS
    trans |
    EACH * END |
    INSERT + END;
```

The user first enters the IFP environment and can then execute a function by using the 'show' command. The final results are displayed back on the screen. For example, the following is a sample session from IFP in which the IFP environment is entered from UNIX and the inner product of two vectors is calculated. The user supplied input is shown in bold.

```
% ifp
ILLINOIS FUNCTIONAL PROGRAMMING
ifp> show < <2 3> <4 5> > : Inner
     23
ifp> exit
%
```

As IFP offers no extensions to standard FP, it still suffers from the same limitation of not being able to handle interactive applications. All input to a function must be known before executing a function. The MPP paradigm described in the next chapter does not suffer from this limitation. MPP extends the capabilities of FP to interactive applications and also provides architectural support for reusee.

CHAPTER IV

MULTILEVEL PROGRAMMING PARADIGM

In 1990, George [GEOR90] introduced the Multilevel Programming Paradigm (MPP). MPP is a three-level hierarchical system that blends the advantages of FP, object-oriented programming, and conventional programming. The first level is based on standard FP. The second level adds an object-oriented class mechanism to group the FP functions together and provide encapsulation. The third level is a procedural level based on conventional programming. The FP algebra of functional programs can be used at the function and class levels while the procedural level is used to extend the paradigm to general purpose situations.

MPP was developed with reusability as a primary goal by providing support for reusable software artifacts of different granularities and support for verification and testing. Support for reusability is especially strong at the function and class levels.

The three levels of MPP, namely function, class, and procedure, are briefly described in the sections that follow.

Function Level

The lowest level is the function level. It is directly based on FP. The set of primitive functions and functional forms given for MPP are similar to the set given for FP. A

full description of these functions and functional forms can be found in [GEOR90]. User defined functions are created by hierarchically combining the basic operations to form new functions. Any function developed at this level can be used by other functions as well as class and procedure level routines.

Class Level

The second level of MPP is the class level. The class level is a merging of object-oriented programming techniques with FP functions. The class level provides a template for grouping FP functions. The template is much like those used in object-oriented programming except that there are no class or instance variables. This is in keeping with the variable-free nature of FP. The class level provides encapsulation and inheritance mechanisms. The basic syntax for the class level is given below:

```
class name
      signature = {fn_name:arg_structure:res_structure,
                   .. }
      reuse information
      inherits class cl_name
      def
         fn_name = function definition
         ..
      end def
   end class name
```

The signature provides the interface information for class functions visible outside the class. The reuse information

is descriptive in nature and given in the form of a comment. The function definitions are FP function definitions. Some examples follow [GEOR90]:

```
class STACK
  signature = { PUSH:<obj,<*>>:<obj,*>,
                POP:<*>:<*>,
                TOP:<*>:obj }
  /* This class implements a stack object. An FP
     object is represented by "obj" and a list is
     represented by <*>. The functions PUSH and POP
     change the stack. The function TOP returns the
     first object in the list. */
  def
     PUSH = eq ∘ [length, 2] → appndl ∘ [1, 2]; ⊥
     POP = not ∘ null? → t1; ⊥
     TOP = not ∘ null? → 1 ; ⊥
  end def
end class STACK

class INPUT
  signature = { READ:<*>:obj,
                ADVANCE:<*>:<*> }
  /* These functions treat a stream as a bounded
     list. */
  def
     READ = null? → ϕ ; 1
     ADVANCE = null? → ⊥ ; t1
  end def
end class INPUT
```

```
class DEQUE
   signature = { PUSH:<obj,<*>>:<obj,*>,
                 POP:<*>:<*>,
                 TOP:<*>:obj,
                 DPUSH:<obj,<*>>:<*,obj>,
                 DPOP:<*>:<*>,
                 DTOP:<*>:obj }
   /* This class implements a double ended stack object.
      The functions PUSH, POP, and TOP are inherited
      from another class.  The functions DPUSH, DPOP,
      and DTOP work on the opposite end from the
      inherited stack functions. */
   inherits class STACK
   def
      DPUSH = eq ∘ [length, 2] → appndr ∘ [1, 2]; ⊥
      DPOP = not ∘ null? → dlast; ⊥
      DTOP = not ∘ null? → last ; ⊥
   end def
end class DEQUE
```

In the above examples, the class DEQUE inherits the class STACK. The signature of a class consists of all the functions that can be accessed through that class including the functions defined through inheritance. Therefore, a procedure using the class DEQUE has access to all three functions defined in the class STACK.

Procedure Level

The third level is the procedure level. It is in this level that the concept of state or variables are incorporated. For this reason, the procedure level extends the paradigm to general purpose situations but it lacks the underlying mathematical base of FP. The syntax of the procedure level is a combination of FP and Pascal syntax. What follows is a sample procedure which simulates a push-down automata [GEOR90].

```
procedure PDA
   environment
      S :: LIST,
      X :: OBJECT,
      I :: STREAM,
   objects
      STACK_OBJECT :: STACK,
      INPUT_OBJECT :: INPUT,
   process
      let X = READ:I,
      while not ∘ null? : X
         eq ∘ [X, TOP:S] ->
            [let I = ADVANCE:I, let S = POP:S];
            [let I = ADVANCE:I, let S = PUSH:<X,S>]
      end while,
      eq ∘ [TOP:I, NULL] -> T ; F
end  procedure PDA
```

The environment section creates a set of named FP objects

similar to the variables used in conventional languages. The variables may represent different values at different points in the execution of the program. This changing of values is performed by the let operation. The objects section provides instances of classes. Any function defined in the signature of these classes may be used in the processing section.

The major objective in this paradigm was to provide architectural support for reuse. While the entire paradigm incorporates reuse support, the support is especially strong at the function and class levels. The REMPP system described in the next chapter provides an environment for MPP program development.

CHAPTER V

REUSE ENVIRONMENT BASED ON MPP - REMPP

The primary purpose of this study has been to develop an environment supporting MPP program development at all three levels with specialized support for the function and class levels. The environment is called REMPP for Reuse Environment based on a Multilevel Programming Paradigm.

Goals

There are several goals that were central to the design of REMPP. These included goals to:

- Promote Reusability. It must be easy and convenient to reuse (inherit) a function or class.
- Provide library support. The user should be able to easily review and access existing classes and functions. Mechanisms are needed to assist the user in deciding exactly which routines to use.
- Use IFP as an underlying FP interpreter. Although the syntax of IFP is different than FP, the semantics are the same. By using IFP as a base, the conversion process is alleviated.
- Provide a clear, readable syntax. The syntax of the function and class levels needs to be easy to understand and should be extended if necessary to provide library support.

- Anticipate code development at all three levels. The development of the procedure level is not a goal of this thesis. However, the environment and the first two levels must be designed with the realization that the procedure level would be added in the future.

Function and Class Level Syntax

Function Level

Function level routines can be developed and added to the library in REMPP. They can then be used in classes or procedures in the same manner as primitive functions. The function level syntax is nearly identical to the function syntax in IFP. The main difference is the incorporation of reuse information. The reuse information consists of a short description (truncated to a maximum of 60 characters for display purposes) of the function. The reuse information immediately follows the function name and is placed in parentheses. The general syntax for the function level is given below. The keywords specified in bold face are required, other information is user supplied:

function fname **(** reuse information **)**
= function definition;

The reuse information is associated with the function name in the library. REMPP provides commands to list each function's reuse information or to search for strings. The reuse information is intended as an aid in selecting a function for reuse. It generally consists of input and

output arguments along with a short English description. New-line characters and preceding blanks are ignored in the reuse information.

Class Level

The class level corresponds to the class level given in MPP with some syntax changes. As with functions, classes must be compiled to be added to the library. They may be inherited by other classes or used in procedures. The general class level syntax is given below:

```
class name ( class reuse information )
signature
  ftn ( function reuse information )
   ...
inherits
  class
   ...
begin
  ftn = function definition ;
   ...
end
```

Excluding minor cosmetic changes, there are three main differences from the class level in MPP. The first is the addition of class reuse information in parentheses immediately after the class name. The second is the replacement of a rigid argument structure in the signature with function reuse information. Both class and function reuse information, which are descriptive in nature, are

intended to assist in choosing the correct routines from the library. The third difference is in the use of IFP syntax rather than FP syntax for the functions. IFP syntax was used since IFP would be used as an underlying base.

Sample Function and Class Level Routines

The following REMPP function is equivalent to the FP and IFP inner product functions given in Chapter III.

```
function Inner ( <<*><*>>:obj Computes inner product )
    =  trans |
       EACH * END |
       INSERT + END;
```

A sample REMPP class which corresponds to the MPP DEQUE class given in Chapter IV is listed below:

```
class DEQUE ( implements a double ended stack )
signature
   PUSH  ( <obj,<*>>:<obj,*>  Push object on top of
           stack )
   POP   ( <*>:<*>  Delete object from top of stack )
   TOP   ( <*>:obj  Return object on top of stack )
   DPUSH ( <obj,<*>>:<*,obj>  Push object on bottom of
           stack )
   DPOP  ( <*>:<*>  Delete object from bottom of stack )
   DTOP  ( <*>:obj  Return object on bottom of stack )
inherits
   STACK
begin
  DPUSH = IF [len,#2] THEN [1,2] | apndr ELSE ?;
  DPOP = IF null | not THEN dlast ELSE ?;
  DTOP = IF null | not THEN last ELSE ?;
end
```

Description of Environment

Before invoking REMPP, the user must set the UNIX environment variable 'REMPP_LIB_DIR' to the name of an existing directory to be used as the library directory. After doing this, the user invokes the REMPP environment from UNIX and enters any of a number of commands specific to the environment.

The rest of this section contains detailed descriptions of REMPP commands. In the commands, optional information is included inside brackets '[ ]'. Options are preceded by a dash '-'. The options may be combined after a single dash or they may be listed separately. They may appear at any place in the command line after the command itself. A sample session showing the use of all commands is given in the Appendix.

- help

  When 'help' is entered at the REMPP prompt, an on-line listing of all available REMPP commands, their options, and a description are given. A question mark '?' may be entered instead of 'help'.

- cmp filename

  Compiles a function or class. The primary purpose of the compile command is to add a function or class to the library. The file can consist of any number of functions or classes; all will be added to the library. Checking is done at compile time for the existence of each inherited class. All inherited classes must have been previously

compiled. If the same function (same function name) is inherited from two or more classes, the last is used.

- lsc [-f] [-r] [classname]

  Without options, 'lsc' lists the names of all classes which have been added to the library (compiled). The '-f' option causes the function names available through each class's signature to be listed. The '-r' option causes the class reuse information to be listed. If both options are used (i.e., '-fr', '-r -f', etc.), class names, function names, class reuse information, and function reuse information are all listed. If a specific class name is given, the list is narrowed down to only that class (if it exists). For example, 'lsc -f -r DEQUE' would list DEQUE's reuse information, all of its available functions (PUSH, POP, TOP, DPUSH, DPOP, DTOP), and their reuse information.

- lsf [-r] [functionname]

  This command lists all function level routines which have been added to the library (compiled). The '-r' options causes reuse information to be listed with each function. If a function name is given, only that function is listed.

- sch [-c] [-f] string

  Search for any occurrence of 'string' in the library. In the default format (no options), the search is made in all class names, their reuse information, the function names in each class,

their reuse information, the function level names, and their reuse information. The '-c' option causes only the class information (class name, class reuse information, function names in each class, and their reuse information) to be searched. The '-f' option causes only the function level information (names and reuse information) to be searched.

- rmc classname

  Remove a class from the library. This command removes a class from the library but the file in which the original source came from is untouched. If it is still available, the class could be recompiled and added back to the library.

- rmf functionname

  Remove an independent function level routine from the library. Works in the same way as 'rmc'.

- exit

  Exits REMPP and returns to the UNIX shell. Ctrl-D may be used instead of 'exit'.

If a command is not recognized by REMPP, it is passed to the UNIX shell: 'sh'. In this way, most UNIX commands can be accessed within REMPP. Vi or another suitable UNIX editor must be used to create and edit the function, class or procedure files.

Two special commands are 'ls' and 'cd' (list files in the directory and change directory, respectively). In the case of 'ls', REMPP checks for options. If there are any requested options, the command is sent to the shell exactly

as entered. If there are no options, REMPP adds an option to list the files in columns and then sends it to the shell. This presents a more readable listing.

A different situation occurs with the 'cd' command. When any UNIX command is passed to the shell, a sub-shell is created and the command is executed in the sub-shell. Commands which do not change the environment ('ls', 'vi', etc.) work as expected. However, commands which change the environment (e.g., 'cd') do not work as expected. They change the sub-shell's environment but not REMPP's. To correct this specific case, REMPP recognizes 'cd' and uses other system calls to change REMPP's working directory rather than a sub-shell's working directory.

Implementation Details

REMPP was developed in C on the Perkin-Elmer running XELOS. It consists of 22 functions in 5 files, a LEX input, and a YACC input. A high level description of REMPP is shown below:

```
initialization
loop
  read command
  call correct routine to handle command
  or pass command to UNIX shell
end loop
```

The UNIX LEX and YACC tools were used to generate the code to compile functions and classes. This decision was made both to simplify the design of REMPP and to allow future

syntax changes to be easily incorporated.

Compilation of part or all of the environment is facilitated by the UNIX make facility. A makefile has been created to compile the necessary parts and put the executable code in the file: REMPP. To execute the environment, the user types REMPP at the UNIX command line.

The library directory is essential to REMPP. As stated previously, the UNIX environment variable 'REMPP_LIB_DIR' must be set equal to a directory name prior to execution. It is in this directory that REMPP stores the functions and classes available for use. In other words, the environment for REMPP is REMPP_LIB_DIR. Any function or class outside this directory is not visible to it.

There are four types of files in the library: files for each class level function, an information file for each compiled class, files for each function level routine, and top level files containing each function and class in the library. These four types of files are described in the following sections.

Class Level Function Files

The IFP interpreter expects each function to be in a separate file with the name of the file matching the name of the function. Because IFP is used as a base, each function in a compiled class must be put in a separate file. In addition, because two compiled classes may have functions of the same name, a special naming convention that ensures unique filenames is required. This is accomplished by changing the name of the function and its resulting file name

to one that refers to the class name in which it is included (two compiled classes cannot have the same name).

A new function name is obtained by counting the functions in each class and appending the current number of functions after the class name. The mapping from the original function name to the new function name is stored in a class information file (to be described next). For the DEQUE class, three class level function files would be created: DEQUE.1, DEQUE.2, and DEQUE.3, representing DPUSH, DPOP, and DTOP, respectively.

Class Information Files

After compiling a class, a file is created to store the information necessary for using that class. This information includes a list of functions in the signature, a list of inherited classes, and a list of all functions defined in the class. The information is needed for the class to be inherited by another class and will also be needed in the procedure level.

The first section of this file contains an integer representing the number of functions in the signature, followed by a listing of each of these functions, the file that the function is stored in, and its reuse information. Following the signature information is a number representing the total number of functions defined in the class followed by a listing of each function and its new function name. The last section in the file consists of a number representing the number of inherited classes and a listing of each inherited class along with the date and time, in seconds,

that the inherited class was last compiled.

For example, compiling class DEQUE would create a class information file with the following information:

```
6
PUSH STACK.1 <obj,<*>>:<obj,*> Push object on top of stack
POP STACK.2 <*>:<*> Delete object from top of stack
TOP STACK.3 <*>:obj Return object on top of stack
DPUSH DEQUE.1 <obj,<*>>:<*,obj> Push object on bottom
DPOP DEQUE.2 <*>:<*> Delete object from bottom of stack
DTOP DEQUE.3 <*>:obj Return object on bottom of stack
3
DPUSH DEQUE.1
DPOP DEQUE.2
DTOP DEQUE.3
1
STACK 651403302
```

The listings of DPUSH, DPOP, DTOP, and their filenames in the two sections of the class information file is redundant. This redundancy is necessary because some functions defined in the class may not be included in the signature, and some functions in the signature may not be defined in this class (inherited).

Function Level Files

Support is provided in REMPP for developing and compiling functions at the function level. When a function level routine is compiled, a file is created in the library and the function is stored in it. Since IFP requires that the file name match the function name, the name of the file is the

same as the name of the function.

Top Level Files

There are two top level files in the library: one for functions and one for classes. The top level function file contains a listing of each function level routine in the library along with its reuse information. In the same manner, the top level class file contains a listing of each class in the library along with its reuse information. These files are used by the 'lsc', 'lsf', and 'sch' commands to list classes, list functions, and search for a string, respectively.

IFP Communication Prototype

The design of REMPP has incorporated IFP interpreter as a functional base. In order to assure that this was feasible, I have developed a prototype which shows that communication between REMPP and IFP is possible. The prototype forks the REMPP routine into an IFP process and continues along both paths. Pipes are then used for two-way communication between the two processes. This forking mechanism is not included in the current REMPP environment but can easily be added with the further development of the procedure level.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND FUTURE WORK

Summary and Conclusions

Software reusability is becoming an increasing factor in software development. As software projects grow larger, more and more developers are realizing the advantages of having a library of existing routines available for reuse. Several language and systems have been developed to facilitate reusability. In this project, I have implemented a new reuse environment based on the MPP paradigm.

FP was originally designed as a replacement for bulky von Neumann languages. Its side-effect-free nature, function-combining nature, and other features afford it great potential for promoting reusability. It does, however, lack support for some general purpose computing situations.

The MPP system was designed as a means of extending standard FP to include stronger support for reusability and to include provisions for general purpose situations. MPP does this by including three levels. The first level is based on standard FP. The second level adds an object-oriented class system to FP functions. The third level is a procedure level used to extend FP to general purpose situations.

The REMPP system which I have designed and implemented

includes features supporting reuse. The function and class levels have been fully developed. Commands in the environment to list classes and functions along with the information useful in deciding which class or function to reuse will enhance the effectiveness of the environment. Once the implementation is complete (with the further development of the procedure level relegated to future work), REMPP should demonstrate the overwhelming advantages of reuse.

Future Work

REMPP currently does not support development at the procedure level. Routines need to be added to either compile and run procedures or to run procedures interpretively. To add compilation, the YACC specification can be expanded to include the necessary rules with a run command added to the list of recognized REMPP commands. However, since IFP is an interpretive system, the procedures may need to be interpreted rather than compiled. For either case, compiling or interpreting, the IFP interpreter needs to be executed. Since IFP is a stand-alone application, one possible scenario could be to have REMPP fork into an IFP process and continue execution along both paths. Pipes can be used to communicate between the processes.

Another possible enhancement to REMPP is the incorporation of more syntax checking when compiling a class. Currently, the compile routines check class level syntax (signature, inherits, etc.) but do not check to ensure that each of the defined functions includes correct IFP syntax. The function is merely copied to the library file until a

semicolon is reached. This design decision was made to simplify the compile process. This leads to the problem that syntax errors in a class may go undetected until the function is actually run by IFP. Thus, an improved method would be to enhance the YACC specification.

BIBLIOGRAPHY

[AHSU86] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AGMG89] W.W. Agresti and F.E. McGarry, "The Minnowbrook Workshop on Software Reuse: A Summary Report," in *Software Reuse: Emerging Technology*. W. Tracz, Ed. Washington, D.C.: IEEE Computer Society Press, 1989.

[BACK78] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613-641, August 1978.

[BAWW86] J. Backus, J. H. Williams, and E. L. Wimmers, "FL Language Manual," *IBM Almaden Research Center*, Technical Report, 1986.

[BIRI87] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions", *IEEE Software*, vol. 4, pp. 41-49, March 1987.

[GEHE86] K. M. Geroge and G. E. Hedrick, "Transformations in the Algebra of Functional Programs," in *Proceedings of the IEEE international Conference on Computer Languages*, pp.40-46, October 1986.

[GEOR88] K. M. George, "Objects and Data Structures in the FP Paradigm," in *Proceedings of the Seventh International Pheonix Conference on Computers and Communications*, pp. 256-260, March 1988.

[GEOR90] K. M. George, "A Multilevel Programming Paradigm," in *Proceedings of the Ninth International Pheonix Conference on Computers and Communications*, pp. 340-346, March 1990.

[GOGU86] J. Goguen, "Reusing and Interconnecting Software Components," *IEEE Computer*, vol. 19, pp. 16-28, February 1986.

[HAWW90] J. Halpern, J. Williams, and E. Wimmers, "Completeness of Rewrite Rules and Rewrite Strategies for FP," *Journal of the ACM*, vol. 37,

pp. 86-143, January 1990.

[HUHH86] T. Huynh, B. Halpern, and L. Hoevel, "An Execution Architecture for FP," *IBM Journal of Research and Development*, vol. 30, pp. 609-615, November 1986.

[KAGA87] G. Kaiser and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software*, vol. 4, pp. 17-24, July 1987.

[KZSG90] M. Kazerooni-Zand, M.H. Samadzadeh, and K.M. Geroge, "ROPCO: An Environment for Micro-Incremental Reuse," in *Proceedings of the Ninth International Pheonix Conference on Computers and Communications*, pp. 347-354, March 1990.

[KERN84] B. Kernighan, "The Unix System and Software Reusability," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 513-518, September 1984.

[MAMI84] G. Mago and D. Middleton, "The FFP Machine - A Progress Report," in *Proceedings of the International Workshop on High-Level Computer Architectures*, pp. 5.13-5.25, May 1984.

[ROBI87a] A. Robison, "The Illinois Functional Programming Interpreter," in *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pp. 64-73, 1987.

[ROBI87b] A.D. Robison, "Illinois FP User's Manual," Professional Workstation Research Group, University of Illinois at Urbana-Champaign, Technical Report, February 1987.

[TRAC87] W. Tracz, "Software Reuse: Motivators and Inhibitors," in *Proceedings of IEEE COMPCON '87*, pp 358-363, 1987.

[TRAC89] W. Tracz, *Software Reuse: Emerging Technology*, W. Tracz, Ed. Washington, D.C.: IEEE Computer Society Press, 1989.

APPENDIX

SAMPLE REMPP SESSION

```
$ rempp

-> ls
 deque.c      outfile      powers.f     square.f
 ifpstuff.c   powers.c     rempp*       stack.c

-> lsc
 List: No compiled classes currently exist

-> cat deque.c

(* this is a class to implement a double-ended stack *)
(* if inherits 3 functions from the class: stack     *)
class deque ( Implements a double-ended stack )
signature
   push    ( <obj,<*>>:<obj,*> Push object on top of stack )
   pop     ( <*>:<*>           Deletes object from top of stack )
   top     ( <*>:obj           Returns object on top of stack )
   dpush   ( <obj,<*>>:<*,obj> Push object on bottom of stack )
   dpop    ( <*>:<*>           Deletes object from bottom of stack )
   dtop    ( <*>:obj           Returns object on bottom of stack )
inherits
   stack
begin
   dpush := IF [len,#2] THEN [1,2] | apndr ELSE ?;
   dpop :=  IF null | not THEN dlast ELSE ?;
   dtop :=  IF null | not THEN last ELSE ?;
end


-> cmp deque.c
 Line 12: Error: inherited class (stack) not found

-> cmp stack.c

-> cmp deque.c

-> lsc
      deque
      stack
```

```
-> lsc -f
      deque
         push
         pop
         top
         dpush
         dpop
         dtop
      stack
         push
         pop
         top

-> lsc -r
      deque       Implements a double-ended stack
      stack       Implements a one-end stack

-> lsc -fr
      deque       Implements a double-ended stack
         push        <obj,<*>>:<obj,*> Push object on top of stack
         pop         <*>:<*>           Deletes object from top of stack
         top         <*>:obj           Returns object on top of stack
         dpush       <obj,<*>>:<*,obj> Push object on bottom of stack
         dpop        <*>:<*>           Deletes object from bottom of stack
         dtop        <*>:obj           Returns object on bottom of stack
      stack       Implements a one-end stack
         push        <obj,<*>> <obj,*> push object on top of stack
         pop         <*>:<*>           delete object from top of stack
         top         <*>:obj           return object on top of stack

-> lsc -fr deque
      deque       Implements a double-ended stack
         push        <obj,<*>>:<obj,*> Push object on top of stack
         pop         <*>:<*>           Deletes object from top of stack
         top         <*>:obj           Returns object on top of stack
         dpush       <obj,<*>>:<*,obj> Push object on bottom of stack
         dpop        <*>:<*>           Deletes object from bottom of stack
         dtop        <*>:obj           Returns object on bottom of stack

-> lsf
 List: No compiled classes currently exist

-> cat powers.f
```

```
(* this function returns the first object to the second power
       <2,4> :power  -> 16 *)
function power ( <obj,obj>:obj  first object to the second power )
  := IF [2,#1]|> THEN [1,[1,2|sub1]|power] | *
     ELSE 1 END;

(* this function returns the square of a number   3:square -> 9 *)
function square ( obj:obj  return the square of a number )
    := [id,id]|*;


-> cmp powers.f

-> lsf
      power
      square

-> lsf -r
      power       <obj,obj>:obj  first object to the second power
      square      obj:obj  return the square of a number

-> sch object
  classes:
      deque       Implements a double-ended stack
         push        <obj,<*>>:<obj,*> Push object on top of stack
         pop         <*>:<*>           Deletes object from top of stack
         top         <*>:obj           Returns object on top of stack
         dpush       <obj,<*>>:<*,obj> Push object on bottom of stack
         dpop        <*>:<*>           Deletes object from bottom of stack
         dtop        <*>:obj           Returns object on bottom of stack
      stack       Implements a one-end stack
         push        <obj,<*>> <obj,*> push object on top of stack
         pop         <*>:<*>           delete object from top of stack
         top         <*>:obj           return object on top of stack
  functions:
      power       <obj,obj>:obj  first object to the second power

-> sch -f object
  functions:
      power       <obj,obj>:obj  first object to the second power
```

```
-> sch bottom
  classes:
      deque        Implements a double-ended stack
         dpush        <obj,<*>>:<*,obj> Push object on bottom of stack
         dpop         <*>:<*>           Deletes object from bottom of stack
         dtop         <*>:obj           Returns object on bottom of stack
  functions:

-> lsc
      deque
      stack

-> rmc deque

-> lsc
      stack

-> exit

$
```

VITA

Gary Wayne Smith

Candidate for the Degree of

Master of Science

Thesis: REUSE ENVIRONMENT BASED ON A MULTILEVEL PROGRAMMING PARADIGM

Major Field: Computer Science

Biographical Data:

Personal: Born in Thomas, Oklahoma, July 19, 1964, the son of Dr. and Mrs. James W. Smith. Raised in Texas. Married to Lisa Min-yi Chen on August 5, 1989.

Education: Graduated from Como-Pickton Senior High School, Como, Texas in May, 1982; received Bachelor of Science Degree from Texas A&M University, College Station, Texas, in December, 1986; completed requirements for the Master of Science degree at Oklahoma State University in December, 1990.

Professional Experience: Engineering Coop, General Dynamics Corporation, Fort Worth, Texas, Spring 1985, Fall 1985, and Summer 1986. Engineer, General Dynamics Corporation, Fort Worth, Texas, January, 1987 to August, 1988. Graduate Teaching Assistant, Department of Computer Science, Oklahoma State University, August 1988 to August 1990.